

**ECSEL Research and Innovation actions (RIA)**



**AMASS**

**Architecture-driven, Multi-concern and Seamless Assurance and  
Certification of Cyber-Physical Systems**

**Design of the AMASS tools and methods for  
architecture-driven assurance (b)  
D3.3**

<b>Work Package:</b>	WP3 Architecture-Driven Assurance
<b>Dissemination level:</b>	PU = Public
<b>Status:</b>	Final
<b>Date:</b>	30 March 2018
<b>Responsible partner:</b>	Stefano Puri (INT)
<b>Contact information:</b>	{stefano.puri } AT intecs.it
<b>Document reference:</b>	AMASS_D3.3_WP3_INT_V1.0

**PROPRIETARY RIGHTS STATEMENT**

This document contains information, which is proprietary to the AMASS Consortium. Permission to reproduce any content for non-commercial purposes is granted, provided that this document and the AMASS project are credited as source.

---

*This deliverable is part of a project that has received funding from the ECSEL JU under grant agreement No 692474. This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme and from Spain, Czech Republic, Germany, Sweden, Italy, United Kingdom and France.*

---

## Contributors

Names	Organisation
Stefano Puri	Intecs (INT)
Ramiro Demasi, Stefano Tonetta, Alberto Debiasi	Fondazione Bruno Kessler (FBK)
Jaroslav Bendik	Masaryk University (UOM)
Petr Bauch	Honeywell International (HON)
Michael Soden, Sascha Baumgart	Ansys Medini Technologies (KMT)
Bernhard Winkler, Helmut Martin	Virtual Vehicle (VIF)
Barbara Gallina, Irfan Sljivo	Mälardalen University (MDH)
Bernhard Kaiser, Behrang Monajemi, Peter Kruse	Assystem Germany (B&M)
Garazi Juez, Estibaliz Amparan	Tecnalia Research & Innovation (TEC)
Eugenio Parra, Jose Luis de la Vara, Gonzalo Génova, Valentín Moreno, Elena Gallego	Universidad Carlos III de Madrid (UC3)
Luis M. Alonso, Borja López, Julio Encinas	The REUSE Company (TRC)

## Reviewers

Names	Organisation
Morayo Adedjouma (Peer review)	Commissariat a l'énergie atomique et aux Energies Alternatives (CEA)
Thierry Lecomte (Peer review)	Clearys (CLS)
Jose Luis de la Vara (TC review)	Universidad Carlos III de Madrid (UC3)
Cristina Martinez (Quality Manager)	Tecnalia Research & Innovation (TEC)

# TABLE OF CONTENTS

<b>Executive Summary.....</b>	<b>7</b>
<b>1. Introduction (*) .....</b>	<b>8</b>
<b>2. Conceptual level .....</b>	<b>9</b>
2.1 System Architecture Modelling for Assurance .....	9
2.1.1 Extended modelling of system architecture with safety aspects.....	9
2.1.2 Tracing CACM with results from external safety analysis tools (*) .....	15
2.1.3 Arguments, Architectures and Tools .....	19
2.1.4 System Modelling Importer .....	24
2.2 Architectural Patterns for Assurance (*).....	24
2.2.1 Library of Architectural Patterns .....	25
2.2.2 Parametrized architectures for architectural patterns.....	33
2.3 Contract-Based Assurance Composition .....	33
2.3.1 Contracts Specification .....	33
2.3.2 Reuse of Components (*).....	34
2.3.3 Contract-Based Assurance Argument Generation (*) .....	34
2.4 Activities Supporting Assurance Case (*) .....	35
2.4.1 Requirements Formalization with Ontologies .....	35
2.4.2 Requirements Formalization with Temporal Logics .....	36
2.4.3 Semantic Requirements Analysis .....	40
2.4.4 Metrics .....	43
2.4.5 Verifying Requirements against System Design.....	54
2.4.6 Design Space Exploration (*).....	56
2.4.7 Simulation-Based Fault Injection Framework (*) .....	57
2.4.8 Model-Based Safety Analysis .....	62
2.5 Assurance Patterns for Contract-Based Design (*).....	62
2.5.1 Assurance of Architectural Patterns .....	62
2.5.2 Assuring requirements based on OCRA results.....	65
<b>3. Design Level (*).....</b>	<b>67</b>
3.1 Functional Architecture for Architecture Driven Assurance .....	67
3.2 System Component Metamodel for Architecture-driven Assurance .....	73
3.2.1 Elaborations .....	74
3.2.2 CMMA Metamodel specification.....	75
3.3 CHES Modelling Language .....	84
<b>4. Way forward for the implementation (*) .....</b>	<b>86</b>
4.1 Feedback from Core/P1 prototype evaluation.....	90
<b>5. Conclusions (*) .....</b>	<b>91</b>
<b>Abbreviations and Definitions.....</b>	<b>92</b>
<b>References .....</b>	<b>94</b>
<b>Appendix A: LTL/MTL.....</b>	<b>97</b>
<b>Appendix B: Architecture-driven Assurance logical architecture (*) .....</b>	<b>99</b>
<b>Appendix C: SafeConcert metamodel.....</b>	<b>105</b>
<b>Appendix D: Design patterns for fault tolerance applied to technology according to ISO 26262 .....</b>	<b>110</b>
<b>Appendix E: Massif Metamodel .....</b>	<b>117</b>
<b>Appendix F: Document changes respect to D3.2 (*) .....</b>	<b>119</b>

## List of Figures

Figure 1.	Meta-model of System Architecture Modelling .....	10
Figure 2.	System Architecture Modelling integrated with Safety Analysis .....	11
Figure 3.	System Architecture Modelling integrated with Safety Analysis and Safety Aspect .....	12
Figure 4.	Work Products of Safety Aspects .....	13
Figure 5.	Overview of the meta-models .....	14
Figure 6.	Safety Core model from Medini Analyze .....	16
Figure 7.	Fault Tree Analysis package from Medini Analyze .....	17
Figure 8.	Diagnostic Coverage Worksheet metamodel from Medini Analyze .....	18
Figure 9.	Tracing metamodel from Medini Analyze .....	19
Figure 10.	GSN illustration of assurance links .....	21
Figure 11.	Relationship between architectural patterns, AMASS System Component and architecture driven assurance objectives.....	25
Figure 12.	The Acceptance Voting Pattern .....	26
Figure 13.	Safety architecture pattern system from [46] .....	28
Figure 14.	Protected Single Channel in SysML .....	28
Figure 15.	Homogeneous Duplex redundancy Pattern in SysML .....	28
Figure 16.	Homogeneous Triple Modular Pattern in SysML .....	29
Figure 17.	M-out-of-N Pattern (Moon) in SysML .....	29
Figure 18.	Monitor-Actuator Pattern in SysML .....	29
Figure 19.	Safety Executive Pattern in SysML .....	30
Figure 20.	Safety architectures in IEC 61508.....	31
Figure 21.	System overview E-Gas Monitoring Concept [33].....	32
Figure 22.	Linear Temporal Logic (LTL) boundaries within Modal Logic (ML).....	36
Figure 23.	Process of formalization of structured requirements using ForReq tool .....	38
Figure 24.	Automatic translation general diagram - From NL to LTL.....	39
Figure 25.	Requirements Analysis Example .....	41
Figure 26.	Example of quality evolution wrt time for a requirements specification.....	52
Figure 27.	Saving snapshot with the quality of the project .....	53
Figure 28.	Information of the snapshot .....	53
Figure 29.	Model Checking Schema.....	55
Figure 30.	Sabotage Framework for Simulation-Based Fault Injection .....	58
Figure 31.	Failure Type System.....	59
Figure 32.	Sabotage Metamodel. ....	60
Figure 33.	Integration workflow: from contract-based design to the generation of saboteurs and monitors.....	61
Figure 34.	High-level assurance argument-pattern for architectural pattern contract-based assurance.....	63
Figure 35.	An argument example of the Acceptance Voting Pattern application.....	64
Figure 36.	The Acceptance Voting Pattern assumptions argument-fragment .....	64
Figure 37.	Contract-driven requirement satisfaction assurance argument pattern .....	65
Figure 38.	Contract satisfaction assurance argument pattern.....	65
Figure 39.	ARTA SystemComponentSpecification and ArchitectureDrivenAssurance components.....	68
Figure 40.	ArchitectureDrivenAssurance components provided interfaces .....	69
Figure 41.	ARTA ArchitectureDrivenAssurance components and external actors/tools .....	70
Figure 42.	Logical Components Collaboration - part1 .....	71
Figure 43.	Logical Components Collaboration - part2 .....	72
Figure 44.	Logical Components Collaboration – part3 .....	73
Figure 45.	BlockType.....	75
Figure 46.	Composite BlockType .....	76
Figure 47.	Contract .....	78

Figure 48. Contract refinement.....	78
Figure 49. System .....	80
Figure 50. Failure Behaviour .....	81
Figure 51. Artefact and assurance-related entities connections .....	82
Figure 52. Links to the executed process .....	84
Figure 53. SystemComponentSpecification internal design .....	100
Figure 54. ArchitectureDrivenAssurance component internal structure - part1, with interface required from SystemComponentSpecification components .....	101
Figure 55. ArchitectureDrivenAssurance component internal structure – part2, with realized interface and interfaces required from SystemComponentSpecification components and from AMASS WP5 and WP6 technical work packages .....	102
Figure 56. ArchitectureDrivenAssurance component internal structure – part3, with realized interfaces .....	103
Figure 57. ArchitectureDrivenAssurance component internal structure – part4, with interfaces required from external tools.....	104
Figure 58. Failure Modes and Criticality .....	105
Figure 59. Failure Behaviours.....	106
Figure 60. Input and Output Events .....	107
Figure 61. Internal Events .....	108
Figure 62. Fault-tolerance events.....	109
Figure 63. Massif meta-model.....	118

## List of Tables

Table 1.	Design pattern template for the Acceptance Voting Pattern .....	26
Table 2.	Result of fault-tree analysis of generic design patterns .....	30
Table 3.	Mapping to RSHP models .....	46
Table 4.	Correctness metrics for models .....	48
Table 5.	WP3 requirements coverage .....	86

## Executive Summary

This deliverable, output of Task 3.2 Conceptual Approach for Architecture-driven Assurance, focuses on the design of the architecture-driven assurance approach by elaborating the way forward identified in D3.1 [2] and by covering the requirements identified in D2.1 [1].

The conceptual approaches, logical architecture, and meta-model supporting architecture-driven assurance are presented in this deliverable.

Concerning the conceptual approaches, elaborations about the following functionalities focusing the support of system assurance definition are provided:

- modelling of the system architecture,
- definition and instantiation of architectural patterns,
- contract-based design approach,
- activities supporting assurance case.

The logical architecture in charge of realizing the architecture-driven assurance on top of the AMASS platform is illustrated by refining the initial logical model presented in D2.2 [3] and then D2.3 [8]; in particular logical components and interfaces that will be in charge of realizing the presented approaches have been identified.

The metamodel for system component specification originally presented in D2.2 has been also reviewed and extended to support what has been elaborated at the conceptual level.

A way forward for the implementation is also proposed, by tracing the sections elaborating the conceptual approaches to the requirements currently assigned to WP3 and by providing some considerations about the current feedback received from the evaluation of the Prototype Core and Prototype P1 of the AMASS platform.

These results, presented in this deliverable, will guide the implementation of the architecture-driven assurance features of the AMASS prototype (Task 3.3 Implementation for Architecture-driven Assurance).

Finally, Task 3.4 Methodological Guidance for Architecture-driven Assurance will build upon the results identified here to provide methodological guidance to the AMASS end-users for the application of the architecture-driven assurance approach.

This deliverable represents an update of the AMASS D3.2 [7] deliverable released at M15; the sections modified with respect to D3.2 have been marked with (\*), then the details about the differences and modifications are provided in Appendix F: Document changes respect to D3.2 (\*).

## 1. Introduction (\*)

This deliverable is the output of Task 3.2. It reports the design of the architecture-driven assurance prototype, including its conceptual aspects and tool infrastructure. We group the functionalities provided by the prototype into four blocks.

**System Architecture Modelling for Assurance.** This block contains the functionalities that are focused on the modelling of the system architecture to support the system assurance, which are:

- Supporting the modelling of additional aspects (not already included in the system component specification), related to the system architecture, that are needed for system assurance.
- Tracing the elements of the system architecture model to the assurance case.
- Generating evidence for the assurance case from the system architecture model or from the analysis thereof.
- Importing the system architecture model from other tools/languages.

**Architectural Patterns for Assurance.** This block contains the functionalities that are focused on architectural patterns to support system assurance, which are:

- Management of a library of architectural patterns.
- Automated application of specific architectural patterns.
- Generation of assurance arguments from architectural patterns application.

**Contract-Based Design for Assurance.** This block introduces the functionalities that support the contract-based design of the system architecture, which provides additional arguments and evidence for system assurance. These functionalities, also include:

- Contracts specification, i.e., specification of components' assumptions and guarantees.
- Contract-based reuse of components, i.e., a component reuse that is supported by checks on the contracts.
- Generation of assurance arguments from the contract specification and validation.

**Activities Supporting Assurance Case.** This block contains the functionalities that are focused on enriching the assurance case with advanced analysis to support the evidence of the assurance case. These functionalities include:

- Requirements formalization into temporal logics.
- Analysis of requirements' semantics based on their formalization into temporal logics.
- Analysis of requirements based on quality metrics.
- Contract-based verification and analysis, i.e. exploiting contracts to verify the architectural decomposition, to perform compositional analysis, and to analyse the safety and reliability of the system architecture.
- Formal verification (model checking) of requirements on the system design.
- Design space exploration to compare different architectural configurations.
- Model-based specification of fault-injection and analysis of faulty scenarios with simulation or model checking (model-based safety analysis).

The deliverable is structured in the following way:

- Section 2 provides the conceptual vision supporting the aforementioned features.
- Section 3 provides a logical architecture supporting the conceptual vision.
- Section 4 provides information related to the WP3 requirements coverage.
- Section 5 provides the conclusions.



## 2. Conceptual level

This chapter builds on the way forward discussed in AMASS D3.1 [2] Section 5 while covering the WP3 requirements identified in D2.1 [1]. For each of the main topics of interest for AMASS related to architecture-driven assurance goal, several approaches and features planned to be supported by the AMASS tool platform are presented.

### 2.1 System Architecture Modelling for Assurance

In this section, the information concerning system architecture, which is important for the assurance case, is elaborated.

#### 2.1.1 Extended modelling of system architecture with safety aspects

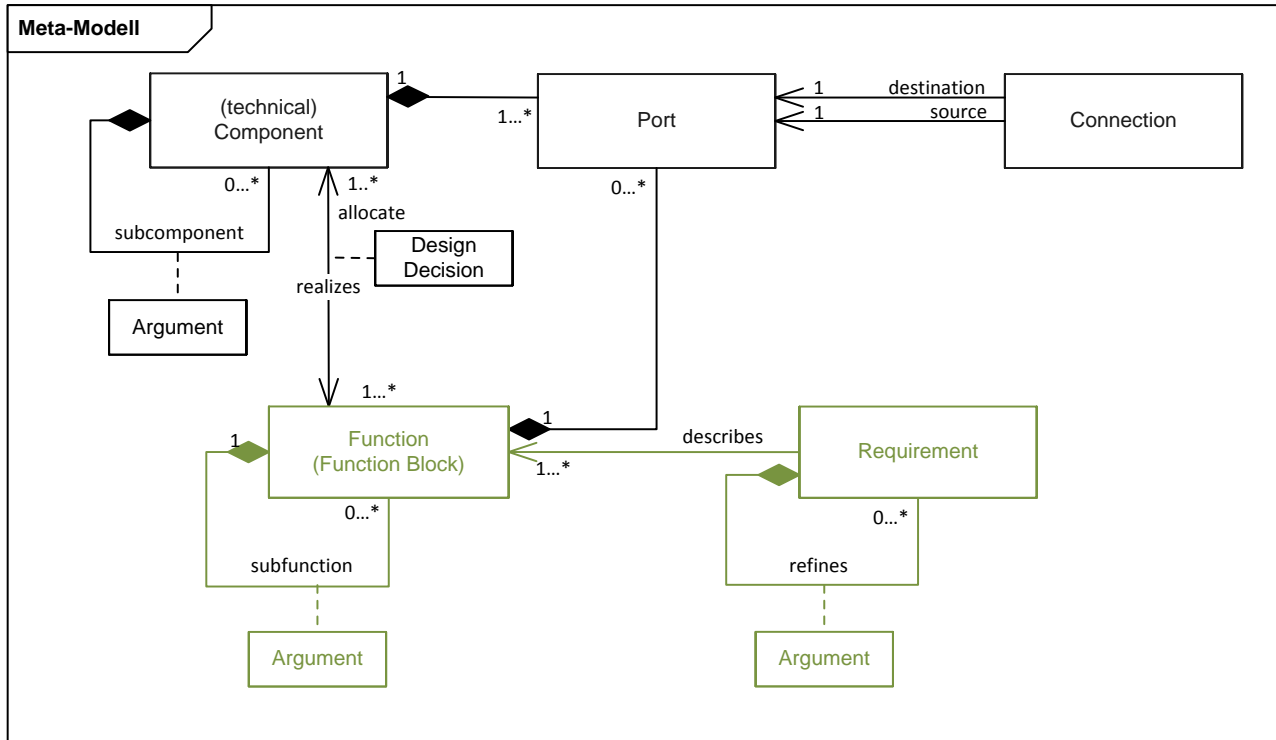
In AMASS D3.1 [2] it is stated that: “The system architecture is one of the first artefacts produced by the development process and includes many design choices that should be reflected in the assurance case. Therefore, we have to understand which elements of the system architecture are important for the assurance case.” What modelling elements are available for expressing the architecture of a technical system and what relationships are allowed between them is defined by a meta-model.

Within the AMASS consortium, different partners have different, but in many aspects similar meta-models, which need to be compared to get a common understanding, even if a full unification is not possible due to existing tools.

In this section, we reflect upon the system modelling itself but also the assurance and safety analysis upon the system and the relations between the system and its safety analysis. In addition to the connections between system modelling and its safety aspects, which are merely the different kinds of safety analysis and the terms used therein (e.g. fault, failure, hazard), safety mechanisms that are introduced into the system architecture to prevent or mitigate these failures or their consequences are also considered.

##### 2.1.1.1 Product Meta-model

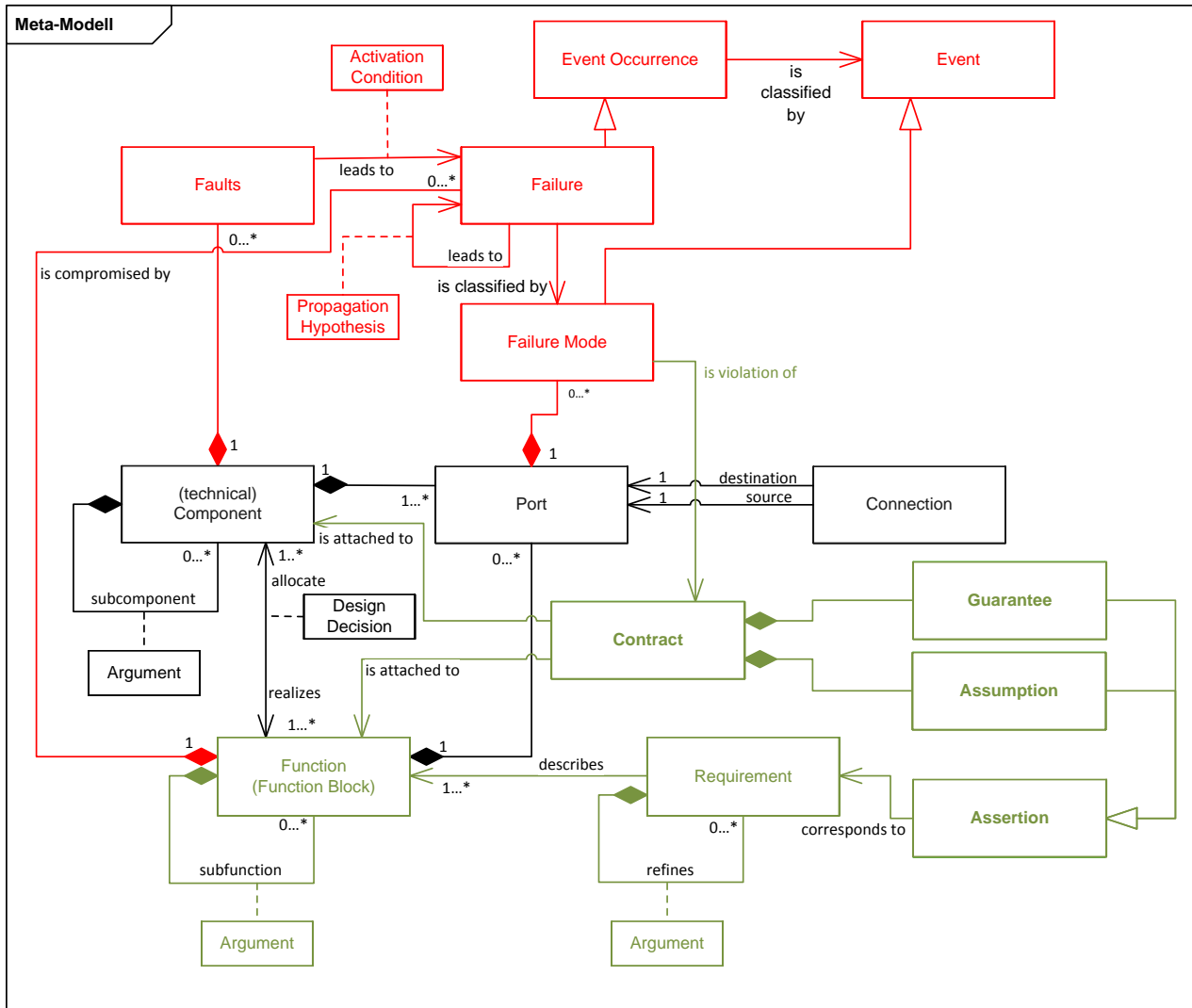
In this section, we introduce a meta-model for system architecture (product) modelling and then integrate it into an assurance framework. This integrated meta-model bridges the gap between an assurance meta-model (e.g. the assurance meta-model described in D2.2 [3]) and a system architecture modelling meta-model, therefore enabling a detailed definition of the system and the analysis of its dependability.



**Figure 1.** Meta-model of System Architecture Modelling

Figure 1 shows the meta-model for system architecture modelling. The artefacts are grouped into two groups, where the green-coloured group corresponds to the functional abstraction level, and the black-coloured group corresponds to the technical abstraction level. On the abstraction level of the functional architecture, we model the *functional blocks* of the system, the nominal behaviour of which is described in detail by the *requirements* that should be satisfied. As a typical recommendation (e.g. from ISO 26262-9), requirements are hierarchically organized where a requirement may be *refined* by a set of lower level requirements. Accordingly, a function may be composed of several *sub* functional blocks in a hierarchical way, with each functional block fulfilling the corresponding requirements.

When defining the technical architecture, the main modelling artefacts are components, which realize the functions (in other words: functions are allocated onto components). *Components* are also organized in a hierarchical way, and one component may contain several *sub* components. Each component may have some *Ports*, which define its interface, and Ports are connected via *Connections*. A Connection allows communications between components through the associated source and destination ports.

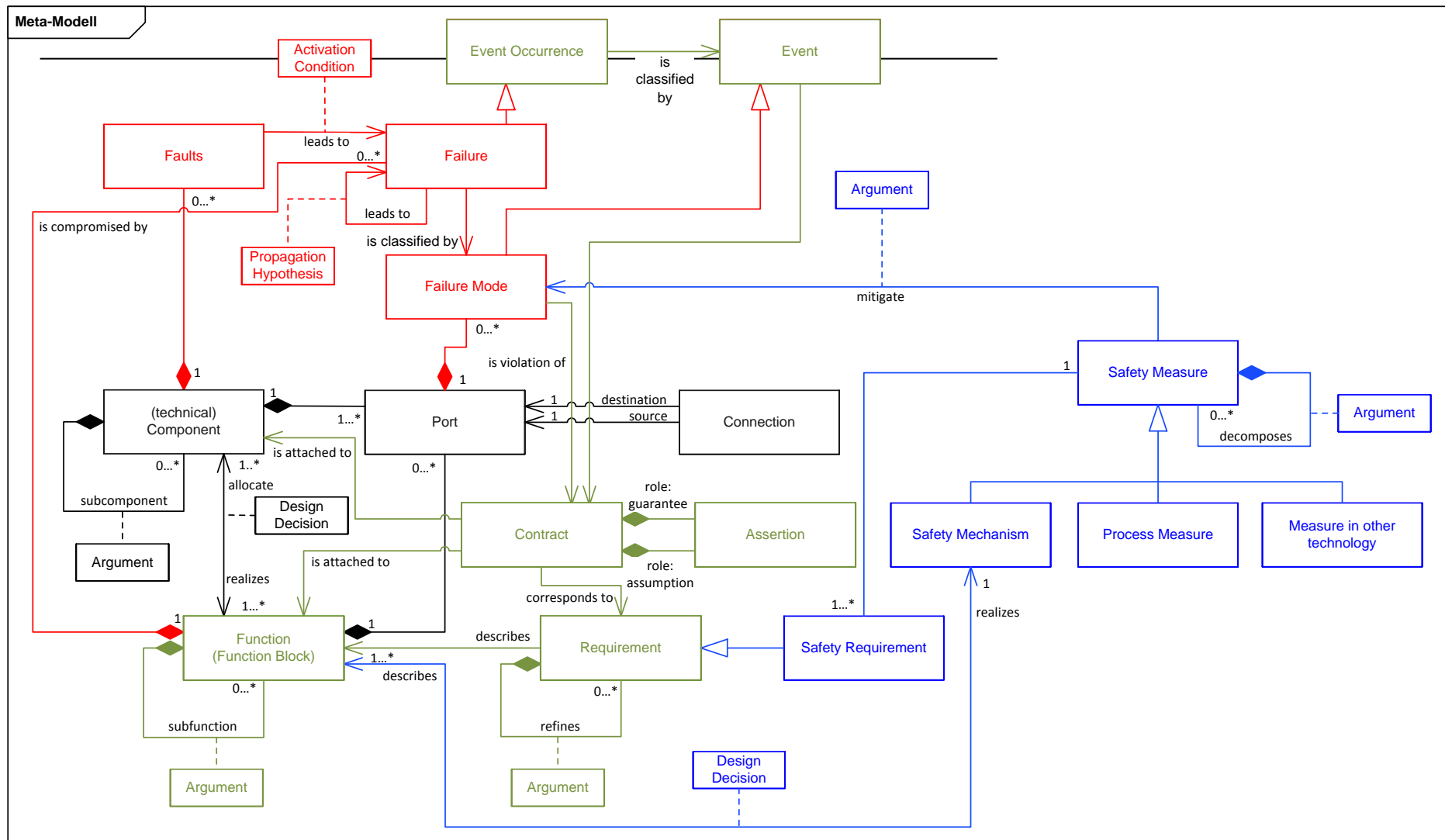


**Figure 2.** System Architecture Modelling integrated with Safety Analysis

As shown in Figure 2, a set of *Faults* may be identified regarding each component as the result of safety analysis over the technical components, which may lead to *Failures* during the operation of the component. For example, a missing Connection between the controller component and the actuator component may lead to the failure that the actuator never executes the command issued by the controller. Therefore, a Failure is an *Event*, which *occurs* in real time during the operation of the component. Failures can be further categorized into different *Failure Modes*, which are different types of Failures that are observed at the Ports of the Component (e.g. “input value of Port A is out of range”, or “No output command on output Port B is issued despite command request is received at input Port A”).

Readers should be aware that throughout different communities and standards the terminology of fault and failure (and sometimes other terms like error or malfunction come additionally into play) may differ, so this meta-model should be regarded as a generic explanation of our intended proceeding and needs to be fine-tuned and mapped to the different existing standards and tools.

Contracts and assertions are also represented in Figure 2, as green-coloured artefacts. In the context of contract based design, *Contracts* are formalized requirements that a system must fulfil with the given conditions. Contracts can be applied to both functional and technical levels. The conditions that are given by the environment of the system are *assumptions* and the expected behaviours are the *guarantees*. Therefore, both assumptions and guarantees can be seen as system properties (i.e. *Assertions* over systems) from different perspectives. In this perspective, Failure Modes can be interpreted as those system properties that violate the Contracts.

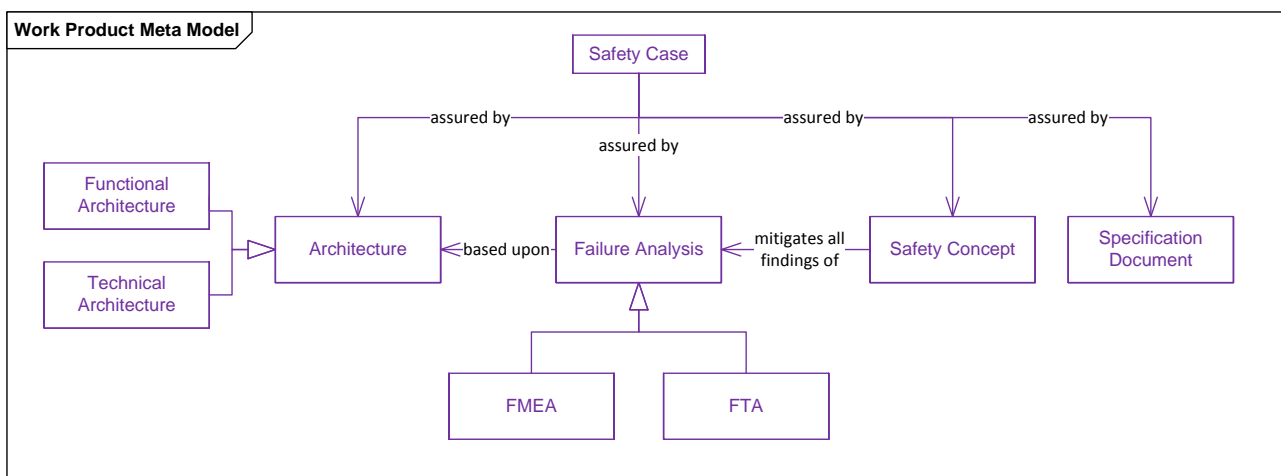


**Figure 3.** System Architecture Modelling integrated with Safety Analysis and Safety Aspect

Figure 3 further integrates the Safety Mechanisms (the blue-coloured group) into the meta-model. Following the safety analysis, a safety concept (may be named differently in different industry domains) is written to define safety measures that prevent or mitigate potential failures or their hazardous consequences. They establish countermeasures against failures at runtime and thereby assure that finally the overall system satisfies the *Safety Requirements*.

Safety measures can be divided into two different classes: process measures (e.g. development process maturity, depth of testing, operator training) and technical measures, which can be further subdivided into functional safety mechanisms (e.g. runtime failure diagnostics implemented in software, with the reaction of a transition to some safe state) and measures in other technologies (e.g. a mechanical protection against touching dangerous parts). For the technical architecture (considering electronic hardware and software), only the safety mechanisms are of interest.

### 2.1.1.2 Work Products of Safety Aspects



**Figure 4.** Work Products of Safety Aspects

The *safety case* is a compilation of the work products (usually in the form of documents) during the safety lifecycle. As a result of the safety analysis, the safety case records the identified hazards and risks of the system under development. It also describes how the safety measures are developed and deployed in order to ensure that the risks are controlled and failures can be detected or prevented. As shown in Figure 4, the safety case consists of four parts:

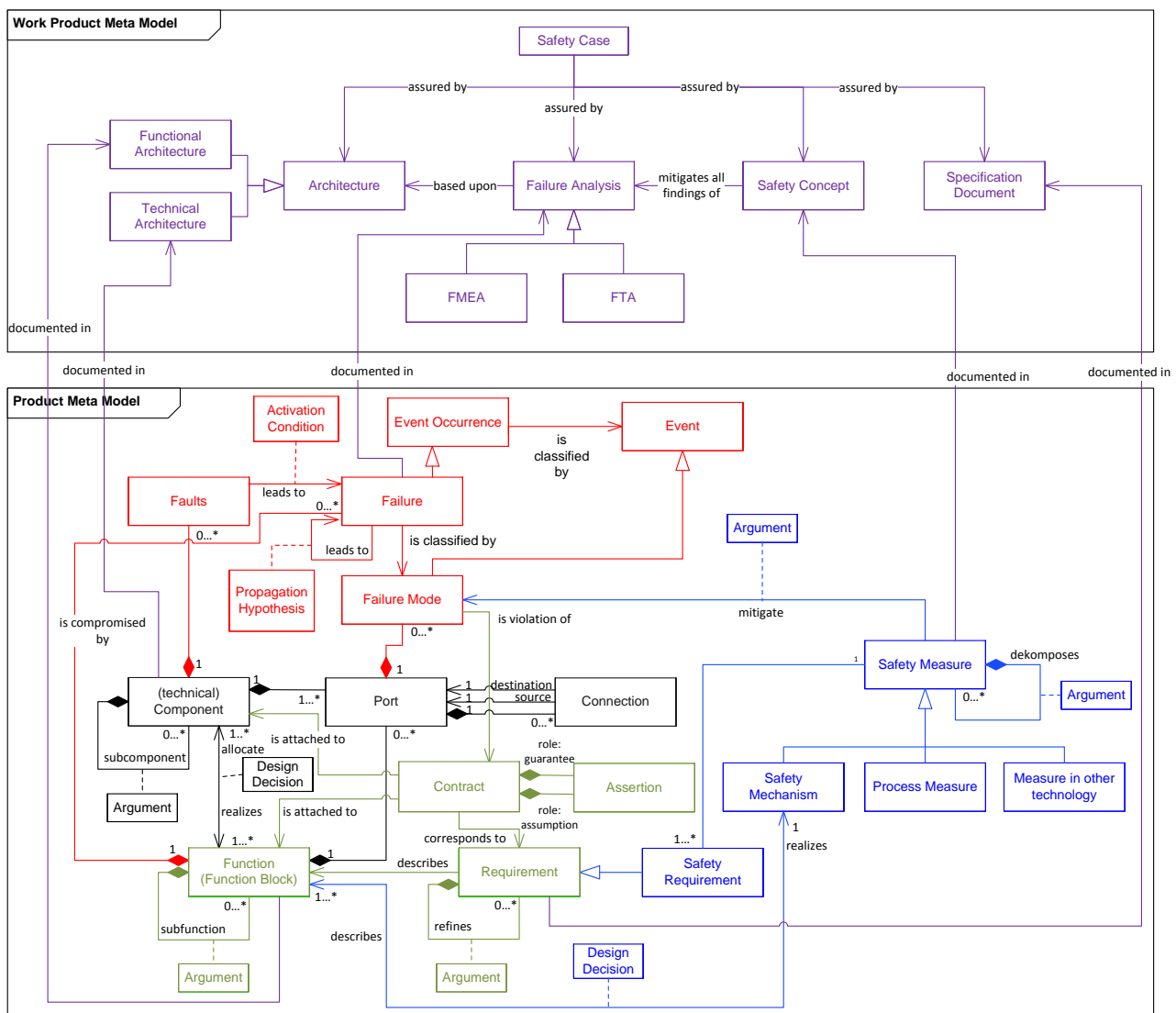
- The *architecture* describes the system modelling, which contains both the *functional* and *technical* architecture.
- *Failure Analysis* describes the safety analysis procedures performed based on the system architecture in order to identify the risks and hazards and the corresponding results (for example FMEA and FTA).
- The *Safety Concept* describes the *safety measures* that are required in order to *mitigate* the failures found in the phase of failure analysis.
- The *Specification Document* describes the requirements of the system under development. In the iteration after performing the safety analysis and writing the safety concept, this also includes the *safety requirements*, which have been derived in the safety concept and which describe in detail how the safety mechanisms shall behave.

The relationship between the work products of a safety case and the artefacts generated during the process of system development and safety analysis is shown in Figure 5.

Note that, just as all parts of the meta model, the safety case part of the meta model (Figure 4 and upper part of Figure 5) is generic and to be understood as an example. Clearly, there are more types of safety analyses than just the two shown in the graphics (FMEA and FTA), and also the *safety case* consists of many

more ingredients than the ones that are shown (ISO 26262 knows as much as 122 work products, not counting the outcomes of the “normal” development process that may also be part of the safety case – but tailoring reduces and condenses the work products actually to be delivered). Which ingredients a Safety Case has, depends on the industry domain, the kind of project and the role of a company within the supply chain (e.g. car/airplane/plant OEM vs. Tier1 supplier vs. component supplier). Tailoring a safety process and, accordingly, the Safety Case is a large topic on its own and addressed in AMASS at other places (e.g. by using the tool OpenCert). The essential message of this meta model is that a link is necessary between the process activities and their output artefacts on the one hand and the product-defining model elements in the SysML world on the other hand: An architecture holds the system components, a requirement specification holds the system requirements, a failure analysis holds the system failures, the Safety Concept holds safety mechanisms, the test specification holds test cases and so on. This has to be extended and adapted to all model elements actually used in some user-specific process setting.

The link made by the meta-model relations finally makes the argument of the *safety case* (or safety assurance case) complete: on process level, the Safety Case argues that the process activities have been carried out carefully (the HARA, the Safety Concept, etc.), and this, in turn, justifies that all hazards have been found, and if the Safety Concept contains measures against all failures contributing to the hazards and they have actually been implemented and verified in the product delivered, then the product can be claimed to be safe.



**Figure 5.** Overview of the meta-models

### 2.1.2 Tracing CACM with results from external safety analysis tools (\*)

As stated in AMASS D2.2 [3], CACM is the evolution of OPENCOS CCL (Common Certification Language) [56] and SafeCer metamodels [11]. CACM is the union of the process-related meta-models (planned process with EPFComposer [58] and executed process with CCL, the assurance meta-model, the evidence meta-model and the component meta-model.

CACM should allow to trace different information, like requirements with system components, results from safety analysis, verification reports, test cases, validation reports, and parts of the safety case; regarding the process, CACM should allow the links between the generated work products and the executed process, the links between the executed process and the planned process, and the links between the generated work products and the planned process, when the executed process does not deviate from the plan.

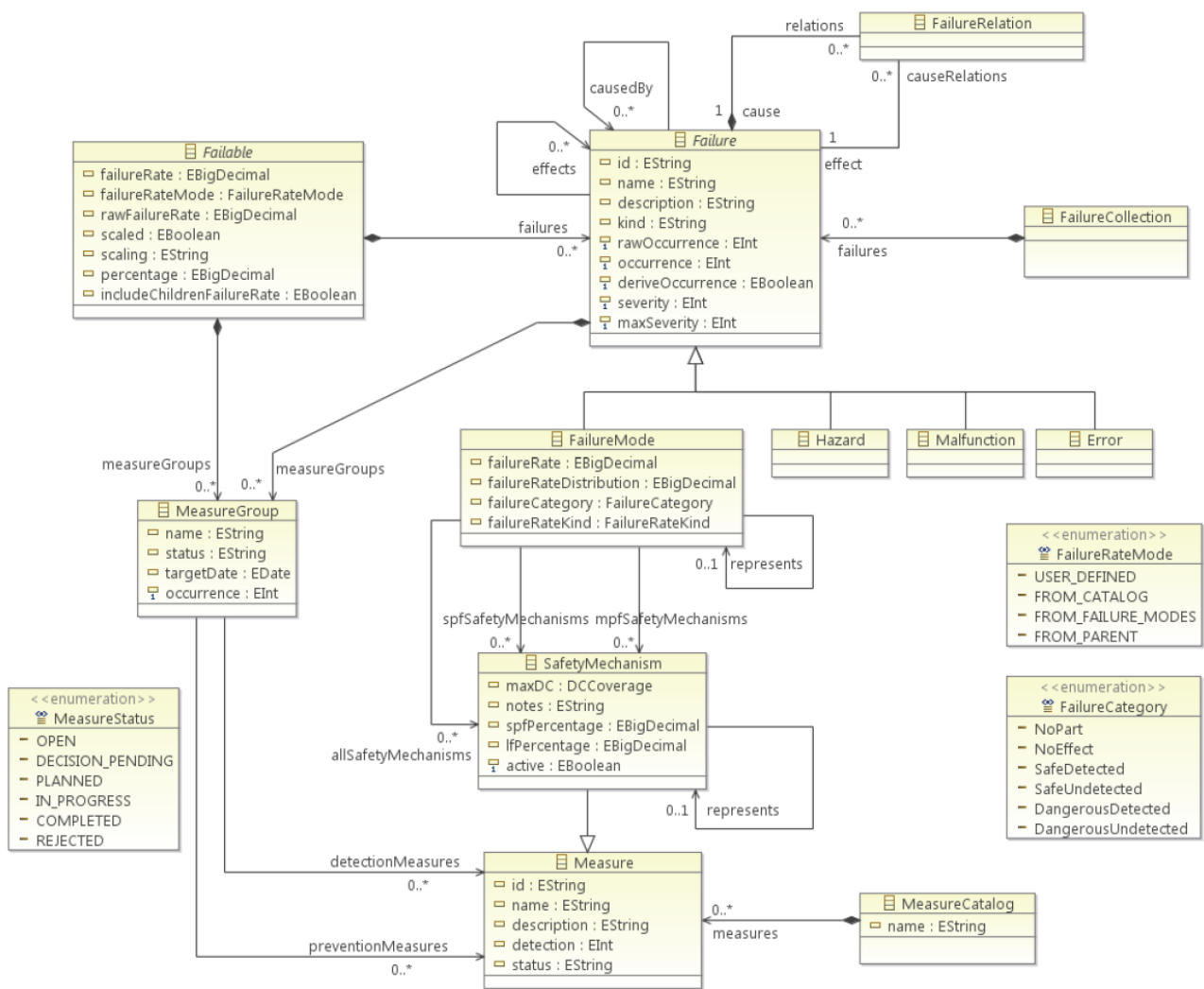
Doing so is desirable from the assurance perspective, as it explicitly defines dependencies between contents of different work products. It is also necessary in the context of a distributed development as defined in ISO 26262. Thereby CACM could support a consistent tracing of activities in the development interface agreement (DIA) as formalization of the responsibilities of customer and supplier.

Consider the example of a system that is partitioned into components, some of which a supplier is developing. The failure modes of the components are tied directly to its functions/interfaces, meaning the type of partitioning greatly influences the failure mode model. That scenario demands traces between parts of different work products and possibly across company borders to preserve the logical structure of components, functions and failure modes. Document based exchange is time consuming and error prone. The associated costs are prohibitive to an iterative process with frequent exchange, review and testing, making document-based exchanges an undesirable option.

For some work products, the AMASS CACM and tool infrastructure already allows to trace links to its sections, such as in most requirements management databases. A model-based approach makes sense for the system model but it is not feasible for many other artefacts. For example, results from safety analysis vary between different domains such as automotive and avionics as well as with respect to security and safety concern. It is not desirable to fit them all into a common metamodel (i.e. into the CACM); there is no added benefit from copying the safety analysis results into the AMASS prototype if instead all related safety analysis can be traced with each other and with CACM model elements. So, for instance, analysis results performed by using external tools to the AMASS platform can be kept according to the metamodel provided by the external tool and properly linked to the CACM (for instance to the executed or planned process).

Tracing data within the AMASS prototype and to external data is part of WP5 which aims to greatly enhance the tool interoperability of OPENCOS. While OPENCOS was open source and therefore open to extension, its CDO-based approach for tool integration fell short in terms of integrating third-party tools in a seamless manner. The goal of AMASS is to employ state of the art live collaborative editing techniques across tool boundaries and provide methods to create traces to artefacts that are external to the platform. Such a link-based approach is the best way to put the single source of truth principle into practice while being flexible and driving down costs.

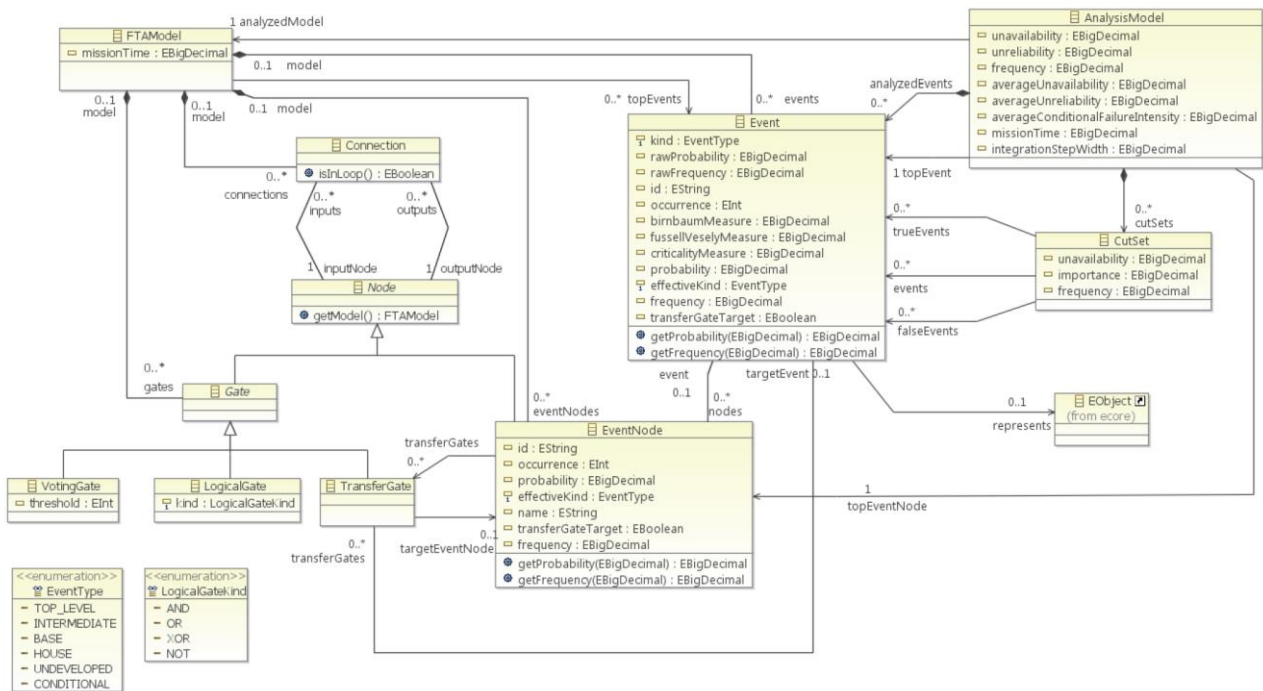
In this section, we discuss what type of artefacts and work product content can already be provided by safety analysis tools such as *Medini Analyze*, which specializes on ISO 26262. It stores its data in well-structured models that allow traces into every part of all models (Figure 6). Information from models created within *Medini Analyze* can enrich the CACM with regard to linking sections within work products for assurance purposes.



**Figure 6.** Safety Core model from Medini Analyze

The type names from the safety core metamodel mostly reflect the terminology from ISO 26262 and are therefore easily understood by safety engineers working with the AMASS prototype. The main class is *Failable*, which is the abstract base class for all elements that can have failures (contained via the reference failures). A component model such as in the SysML modelling language or the one used in the context of AMASS can inherit from this class to receive all safety relevant properties. For example, *Failable* provides a *failureRate* as quantified rate of the amount of failures over time.

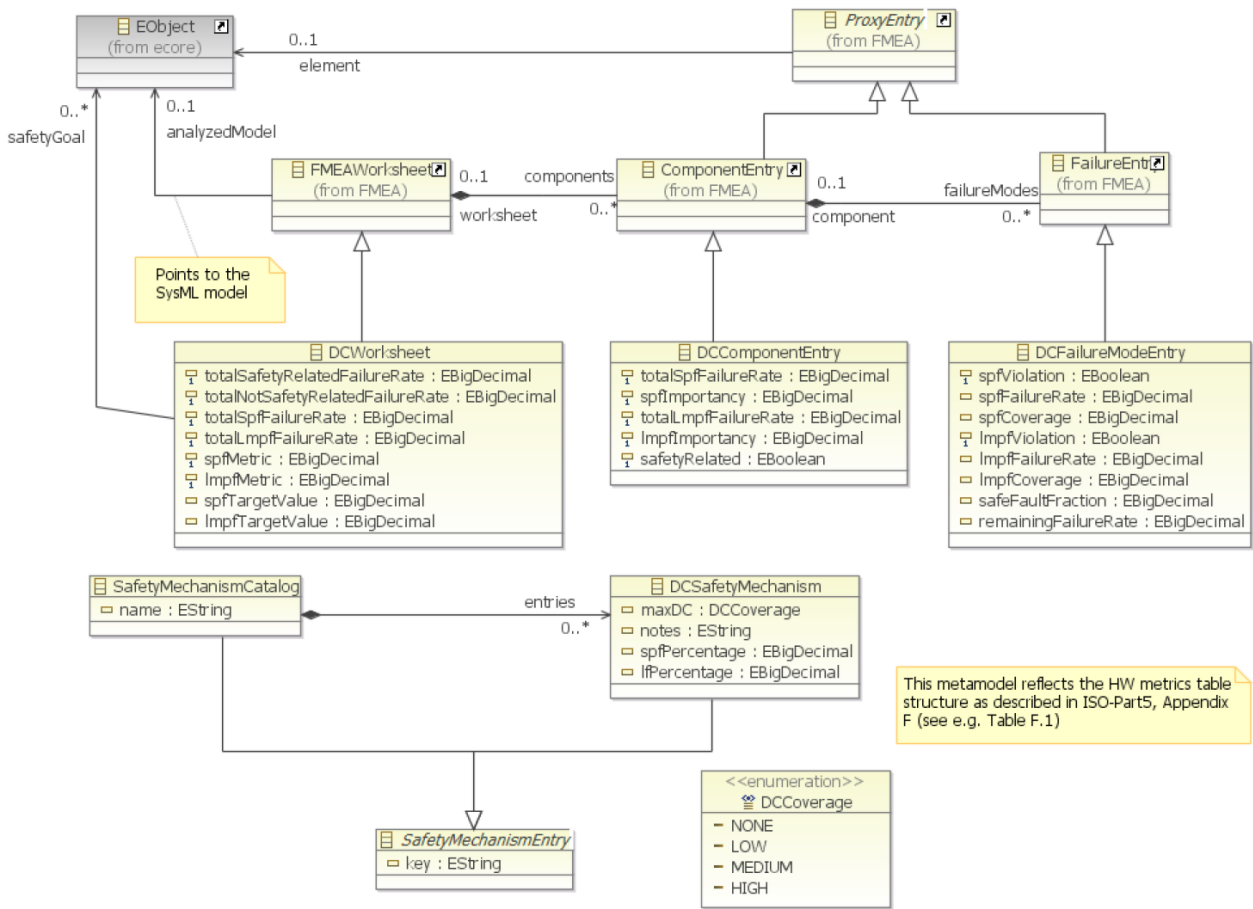




**Figure 7.** Fault Tree Analysis package from Medini Analyze

Figure 7 presents the fault tree model, which consists of a tree structure with various node types, mainly events (metaclass *EventNode*) and gates (metaclasses *LogicalGate*, *VotingGate*, *TransferGate*). The connection between nodes is realized by the abstract metaclass *Connection* that links two *Node* instances.

Each *EventNode* of the fault tree has a reference event to a single *event*, which holds all its properties. Hence, instances of metaclass *EventNode* describe where an event occurs in a fault tree, while metaclass *Event* defines the event itself in detail. In case of multiple occurring events, different *EventNode* instances can reference the same underlying *Event* object. How often an event is referenced from the fault tree is indicated by the *occurrence* attribute.



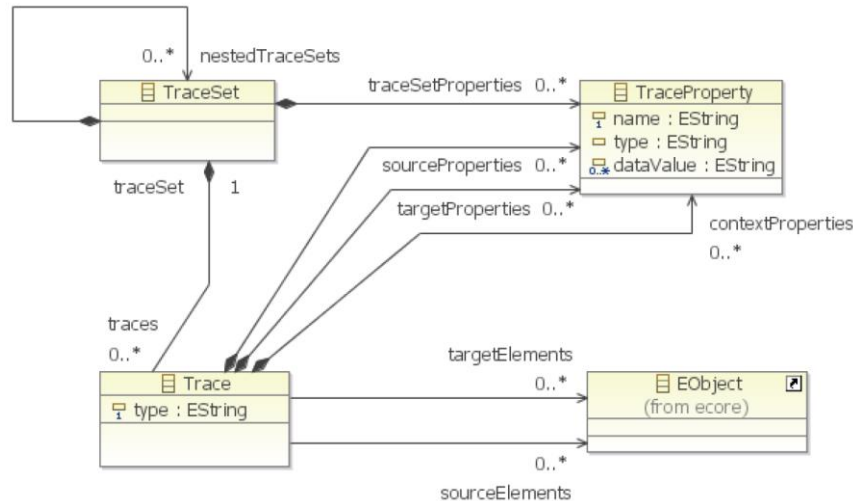
**Figure 8.** Diagnostic Coverage Worksheet metamodel from Medini Analyze

The three main classes in the diagnostic coverage metamodel presented in Figure 8 are *DCWorksheet*, *DCComponentEntry*, and *DCFailureModeEntry*. These classes consistently refine the structural classes of the Failure Mode and Effect Analysis (FMEA) worksheet to add the attributes required for the Failure Mode and Effects Diagnostics Analysis (FMEDA) Single Point Fault Metric (SPF) and Latent Fault. In detail:

- *DCWorksheet* inherits from *FMEAWorksheet* and adds all attributes relevant for the hardware architectural metrics. The safety goals under consideration are linked via *safetyGoal* reference. Target values from the set of goals are maintained in *spfTargetValue* and *ImpfTargetValue*. These attributes are not derived and can be changed as known from the tool UI. As defined in ISO 26262-5, the essential attributes for the computation of the SPF/LF metrics are available as *totalSafetyRelatedFailureRate*, *totalNotSafetyRelatedFailureRate*, *totalSpfFailureRate*, and *totalLmpfFailureRate* as well as the overall computed results *spfMetric* and *ImpfMetric*.
- A *DCWorksheet* contains always *DCComponentEntry* via the *components* reference. *DCComponentEntry* specializes *ComponentEntry* from FMEA to add the attribute *safetyRelated* and the derived attributes *totalSpfFailureRate*, *totalLmpfFailureRate*, *spfImportancy*, and *ImpfImportancy*. The latter four attributes are computed based on the contained *failureModes* and their properties related to the SPF/LF metrics.
- *DCFailureModeEntry* stores the main attributes required for the metric computations, i.e. *spfViolation* and *spfCoverage* (for SPF), and *ImpfViolation* and *ImpfCoverage* (for LF). In addition, the percentage of safe faults is accessible via *safeFaultFraction*. Beside these five attributes there are three derived attributes for the various failure rate fractions, namely *remainingFailureRate* (after subtraction of safe fault percentage), *spfFailureRate* (after incorporation of the *spfCoverage*), and *ImpfFailureRate* (after further incorporation of the *ImpfCoverage*).

Lastly, we present the tracing model from Medini Analyze (Figure 9). As many traces are generated in 3rd party tools such as requirements management databases or safety analysis tools, there will be many trace links generated inside these tools. Since it would be tedious to duplicate those traces manually, it is preferable to import them into the AMASS tool platform.

For these needs, in the context of WP5 (see AMASS deliverable D5.5 [9], the Capra project [59] for generic traceability is used and adapted for the AMASS/WP3 needs. Capra comes with a dedicated metamodel for traceability which is quite close to the one presented in Figure 9 (it can also be customized). So, Capra can be used to support traceability links between CACM, in particular the component model, and other assurance-related information, like results from AMASS external analysis tools.



**Figure 9.** Tracing metamodel from Medini Analyze

## 2.1.3 Arguments, Architectures and Tools

### 2.1.3.1 Argument Fragment Interrelationships

Requirement WP3\_APL\_005 indicates: "The system should be able to generate argument fragments based on the usage of specific architectural patterns in the component model." Our objective concerns the ability to both represent complex argument relationships and achieve a component-oriented assurance architecture. We start with a simple example, to demonstrate the argument components and relationships needed, and then we generalize to metamodel concepts that would need to be included.

As an example, consider a derived safety requirement that a system fails silent. This is a derived requirement that comes from safety analysis, to ensure that when a processing component fails, it does not produce any further output. The system designer might use an architectural pattern to meet this requirement. For example, the design might use an independent protection mechanism whereby a safety system can detect that a component has failed, and disconnect or override its output drive so that it cannot affect the rest of the system.

In a safety argument, one would typically start by enumerating system hazards and showing that the list of hazards is complete, then deriving safety requirements to mitigate those hazards, followed by arguments to show that the system meets these safety requirements. In part, this is driven by the need to allocate requirements among software and hardware components, so this approach seems apt for architecture-driven assurance.

The argument has the following overall structure, starting from derived safety requirements:

- A claim that all derived safety requirements are met, contextualized by a specific architecture and a specific set of derived safety requirements.

- A subordinate claim for each derived safety requirement and applicable component, showing that this requirement is met for this component.
- For a component meeting a fail-silent requirement with an independent protection mechanism, a specific argument fragment can then be used:
  - A claim C1 that the architectural pattern meets the fail-silent requirement.
  - A claim C2 that the system correctly instantiates the architectural pattern.

Under claim C1, we can appeal to evidence from model-checking, for example, demonstrating that the fail-silent protection mechanism works correctly over mode changes, power cycles, system resets and so on.

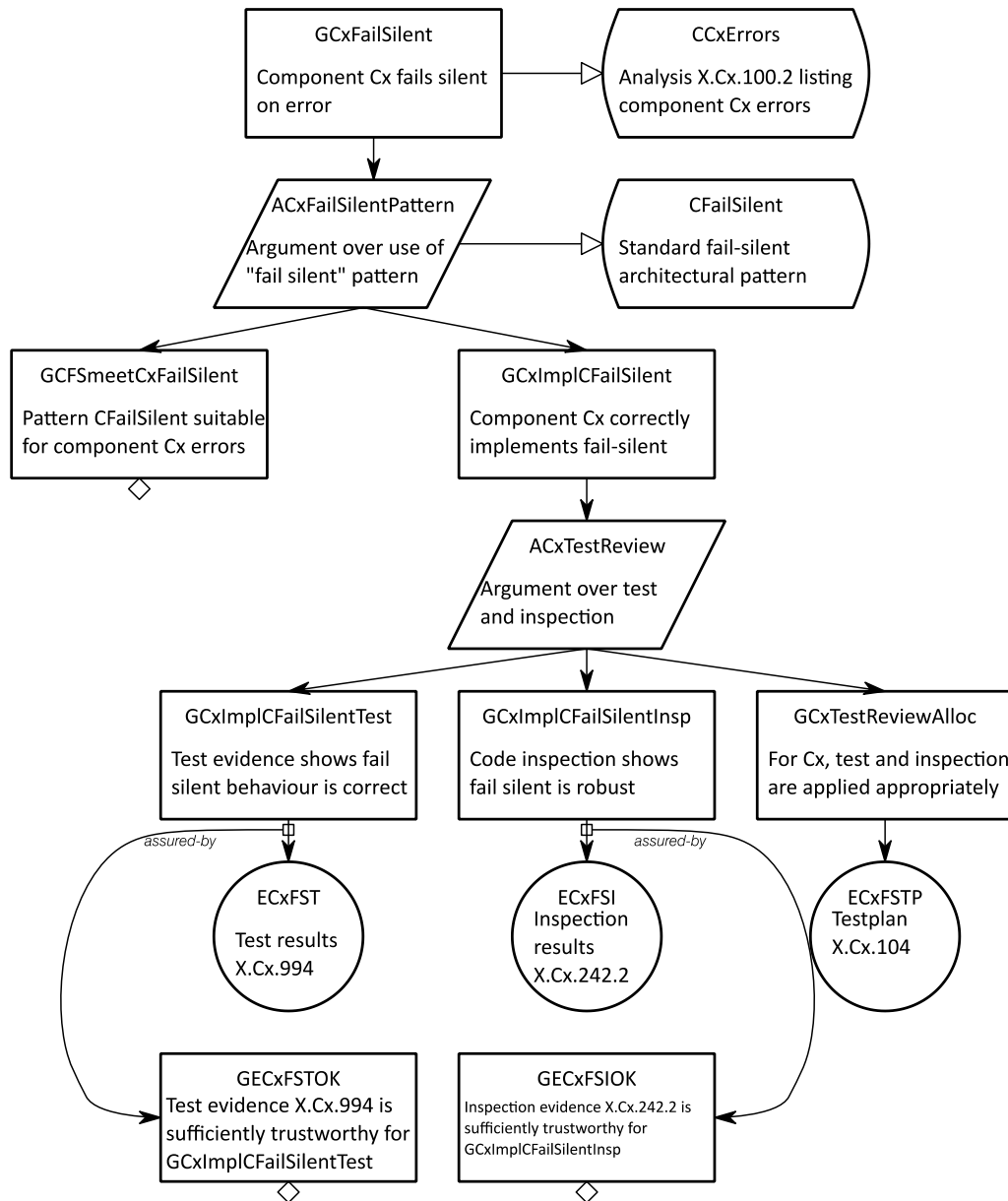
Under claim C2, we can appeal to design review for some instantiation rules. For this type of pattern, we could also appeal to specific tests of the implementation in scenarios achieving, for example, 1-switch coverage of transitions in the model used in claim C1.

In claim C1, we use a model-checking tool to obtain evidence about the behaviour of a model. In claim C2, we use a test execution tool to obtain evidence about the behaviour of the software. In both cases, it is important to show that the evidence is trustworthy. This is an argument about the ability of that evidence to substantiate a higher-level claim, which sits alongside the main assurance argument.

To claim that evidence is trustworthy, we appeal to the workflow used to generate the evidence. The model-checking workflow involves generation of an accurate abstract model, correct configuration of the model-checking tool to perform appropriate analysis and qualification evidence showing that the tool faithfully performs the analysis required. The workflow for testing the protection mechanism involves generation of a sufficiently representative verification environment, generation of appropriate traceable test cases, correct configuration of the test tool to perform appropriate tests and qualification evidence showing that the tool faithfully performs the analysis required.

To benefit from architecture-driven assurance, we would like to link these fragments together: the overall safety argument, arguing over derived requirements, the specific treatment of the fail-silent protection mechanism, the model-checking evidence assurance case and the test execution evidence assurance case. Not all of these links are “support” links; the last two do not themselves argue towards supporting a particular claim, but instead argue about the ability of some other evidence to support that particular claim. It must be possible in the argument and architecture metamodels to represent these links. An illustration is given in Figure 10 to put these ideas in context.

It is worth to highlight and clarify here that the envisaged AMASS approach regarding usage of architectural patterns and associated argument fragments is presented and discusses in more detail in Section 2.2. The elaborations presented here apply in general, not only in case of architectural patterns application; patterns are used here as example to elaborate about the needed argument relationships.



**Figure 10.** GSN illustration of assurance links

We propose that a fragment may therefore need to describe its top-level relationship not only as a *support*, but also as an *assure*:

- Some element *supports* some external element with a contractual description of its role. For example, the fragment for “external protection pattern argument” could have a strategy “argument by design” that iterates over the elements of the external protection design and supports any claim of the form “external protection design meets the fail-silent requirement”.
- Some element *assures* some external “support” association with a contractual description of its role. For example, the fragment for “trustworthy model-checking” could have a strategy “argument by model-checking workflow” that assures any structure with a claim “{statement} in all configurations” supported by “{model-checking evidence}”.

The situation is further complicated when considering evidence that includes testing of an embedded target. In this case, the off-the-shelf analysis tool includes custom components for that specific embedded target. Such tools can be arranged as follows:

- An off-the-shelf part, containing facilities for source code analysis, build system interception, data collection and data processing;
- A custom part, known as an *integration*, containing specific configuration settings, scripts and build system and test system modifications to coordinate the off-the-shelf tools.

By using the RapiCover tool<sup>1</sup>, a coverage analysis for critical software from Rapita Systems, a tool qualification kit is offered, which is itself made of two parts:

- The off-the-shelf kit, giving evidence that the off-the-shelf facilities operate correctly in a specific range of configurations called the qualification scope.
- The custom kit, known as an *integration qualification*, containing review results, a specific test program and expected results from that program. The user must follow the instructions in the integration qualification report to obtain actual results from executing that test program on his own target, and then compare those with the expected results. Additionally, the off-the-shelf kit and the integration qualification report both contain conditions of use that limit the user's use of the tool or require that the user performs some additional manual process steps.

These two elements, the associated qualification test configuration, the test evidence, the conditions of use and the user's own process steps must plug in under tool qualification using appropriate links, to provide the required level of assurance that the evidence really supports the claim. If there are specific development assurance levels involved, then this relationship should also be checked, perhaps by providing assurance levels and qualification levels as attributes of the various elements in the argument.

#### 2.1.3.2 Evidence Reuse

We expect that, among the user's safety materials there will be some consideration of tool qualification, in DO-178C for example, there are specific entries in the Plan for Software Aspect of Certification (PSAC) relating to such considerations (objective 11.1g). This gives rise to a potentially reusable body of evidence with associated argument, comprising the coverage run configuration, the coverage results, the justification review status for coverage holes, the generic qualification kit documentation, the integration-specific qualification documentation, the tool configuration used during the integration test, the integration test result and the comparison between the integration test result and the expected result.

Hence, for the tool user, we must consider:

- how does the AMASS user relate the generic kit documentation pack to the safety argument?
- how does the AMASS user control and perhaps automate execution of the integration qualification test?
- how does the AMASS user provide evidence links between the tool qualification part of the safety argument, the development process and the conditions of use in the generic qualification kit and the integration qualification report?
- how does the AMASS user use the comparison of the expected and actual results in the integration qualification kit to the tool qualification part of the safety argument?

At Rapita Systems, we have applied some of the above considerations to the design of the input and output interfaces for the RapiCover coverage tool. For each of the above scenarios, the interaction with AMASS imposes some constraint on that interface that must be addressed in the design.

---

<sup>1</sup> <https://www.rapitasystems.com/products/rapicover>



### 2.1.3.3 Testing context

The user must be able to characterize the test run in terms of what components are being tested and what verification environment is being used. For example, the user may end up with four pieces of structural coverage evidence:

- structural coverage of the operating system and drivers from unit testing on the target with test applications;
- structural coverage of the application on a host simulation environment from a system test suite;
- structural coverage of selected parts of the application running dedicated unit tests designed to hit error-handling code that cannot normally be triggered;
- justifications and analyses for each coverage hole that was not exercised during the testing.

The user will need to refer to these items to argue that structural coverage objectives have been satisfied (DO-178C, table A-4). The argument should also substantiate several related claims about the evidence:

1. that the configuration of the software and hardware used in the tests matches the configuration being submitted for certification;
2. that the unit testing of the operating system and drivers complies with reusable software component criteria (e.g. AC20-148);
3. that the justifications and analyses are complete per their process;
4. that the qualification test was completed for the test environment(s) used;
5. that the integration was reviewed per defined integration review processes.

Depending on the user's argumentation strategy, there may be associated claims here about the traceability between the test evidence and the requirements, or that type of claim may be presented in a different argument.

For RapiCover, an additional concern is how exactly the user will provide the overall configuration information to the tool so that it can be embedded within the generated structural coverage evidence. We already have a system of tagging build identifiers throughout the process, so the investigation here would start with trying to extend that facility. This does raise two design issues, which we have started to address:

- To supply external configuration information at the start of the build to attach to the results, we hook into ongoing work to provide a central configuration file for a Rapita Verification Suite<sup>2</sup> (RVS) integration. We have designed this system to be extensible at run-time by wrapping the static integration configuration with additional runtime information. We will create appropriate configuration options for additional configuration information or references.
- We are also experimenting with ways of tracking corresponding data throughout the integration, especially in cases of incremental rebuilds of the software under test. We will need to take this into account in the design, so that we can track advanced configuration information without making the on-target tracking excessively complex.

### 2.1.3.4 Model-based tagging

The user's tools or processes may insert traceability tags into the source code to identify where they come from in higher-level artefacts such as models or requirements. The coverage tool should include facilities to extract these tags to associate the tested code and the structural coverage results with those artefacts. These tags could take the form of a convention for identifiers in the code, use of attributes, use of pragmas, formatted comments in the code, or some external information associating identifiers with parts of source files.

For RapiCover, the aim at this stage is to provide flexible scripting to allow the user to handle a specific scheme, rather than trying to match RapiCover directly to a particular model-based tool environment. Such

---

<sup>2</sup> RVS (Rapita Verification Suite) is a tool suite to verify the timing performance and test effectiveness of critical real-time embedded systems (<https://www.rapitasystems.com/products/rvs>)

scripting should be able to access the source code structures and surrounding comments as well as reading structured file data such as JSON or XML. The extracted identifiers can then be attached to the corresponding coverage data and exported or queried by other tools when extracting the coverage information.

#### **2.1.3.5 Qualification kit**

We expect that the AMASS CACM ManagedArtifact::CitableElement (see Chapter 3.2.2.5) follows the OPENCOSSEvidence metamodel's use of the Resource class, providing the ability to link to a specific file as a document containing supporting evidence for a claim made in an argument. (Note: it may also be useful to provide some facility to automatically check that the document is the declared document, by querying its content for metadata per the declared format and checking against the content of the Artefact instance in the model, but this might not be within the scope of the project).

If the AMASS toolset can control the deployment and execution of software with associated tests, it could also provide for execution of the integration qualification test. We typically ship a custom procedure that the user should follow to obtain the result and check that it conforms to the expected result of the integration qualification test. We envisage creating an off-the-shelf procedure to fit within AMASS evidence management that comes with the corresponding scripting and argument structures to show that the integration is in a valid configuration. However, there will still be some need for user interaction. For example, the user typically needs to rerun this check after moving lab equipment involved in the on-target testing; this is not a situation that we expect the AMASS infrastructure will be able to detect.

#### **2.1.4 System Modelling Importer**

The system architecture can play a fundamental role in the assurance of the system. Therefore, the AMASS platform must be able to define the system architecture with built-in functionalities and to import the system architecture from other modelling tools. The System Component Specification is a core component of the AMASS platform, which allows to specify the system architecture in SysML. The Architecture-Driven Assurance component of the AMASS platform will have a System Modelling Importer subcomponent that will take care of importing existing models from other languages/tools. For example, it will import models from OCRA<sup>3</sup> and will automatically create the CHESSE diagrams for the model. The importer will make sure that the import will preserve the semantics of the original model. In particular, it will configure the time model and the type of composition (synchronous vs. asynchronous) based on the semantics of the original model. It will restrict the functionalities when the import is not possible due to non-supported features (e.g., non-supported data types).

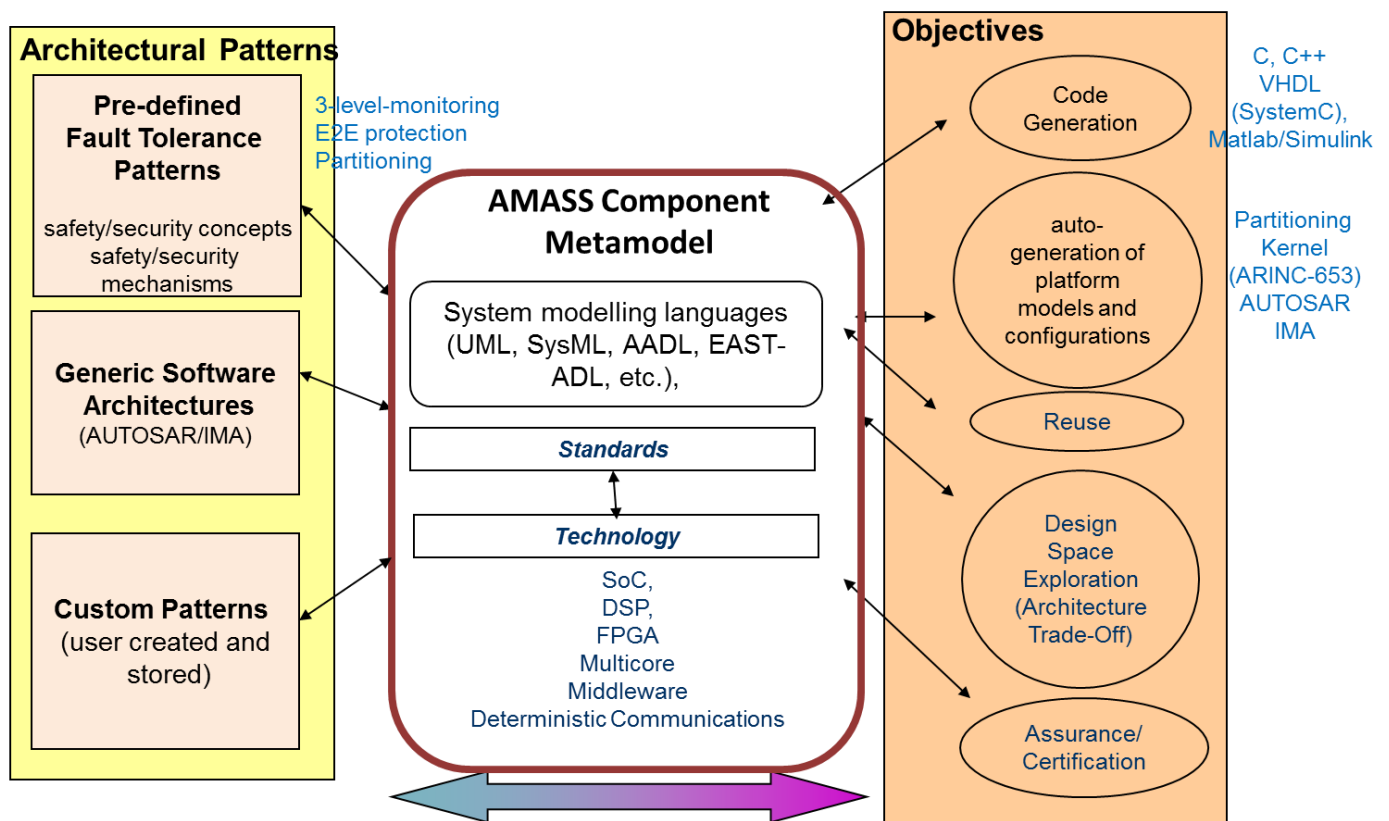
### **2.2 Architectural Patterns for Assurance (\*)**

Design patterns were proposed in [36] by the architect Christopher Alexander in order to establish a common solution to recurring design problems. This approach helps designer and system architect when choosing suitable solutions for commonly recurring design problems. Furthermore, design patterns lead to remarkable benefits and they can be applied for different purposes (c.f. Figure 11). Through applying this solution, component reuse is more easily achieved. Moreover, it facilitates design space exploration process to trade-off between different properties, creates automatic system configurations from system models and generates automatic code avoiding the introduction of systematic faults. In brief, this concept might be very useful to support the design of safety-critical systems.

---

<sup>3</sup> <http://ocra.fbk.eu/>





**Figure 11.** Relationship between architectural patterns, AMASS System Component and architecture driven assurance objectives

Designing the architecture of a safety-critical system implies analysing different safety tactics in order to decide the most suitable safety concept based on dependability, certification and cost requirements.

As previously mentioned, a design pattern presents the solution to a recurring design problem in a consistent and coherent manner. Taking into account that several functions and sub-systems are common to different vehicle or aircraft models, the interest of applying component and safety artefacts reuse is considerably increasing. The use of design patterns emerges as a viable solution to address the aforementioned issue. By doing so, the safety of a system such as AUTOSAR or IMA can be achieved in a modular manner. Those patterns might be developed by means of different architecture modelling languages such as OMG SysML at the system level.

## 2.2.1 Library of Architectural Patterns

### 2.2.1.1 Design Patterns: general structure

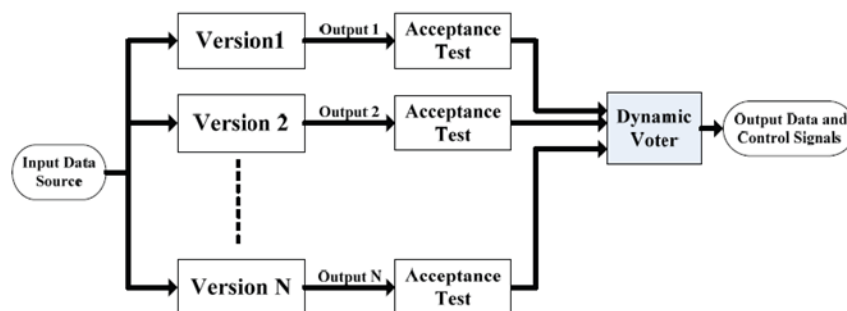
In this section, the proposed design pattern template is introduced which includes the following parts:

- **Pattern Name:** define a name which describes the pattern in a univocal way.
- **Other well-known names:** this item refers to other names with which the design pattern can be known in different domains of application or standards.
- **Intent/Context:** define in which context the pattern is used. For example, define if the pattern is recommended for a specific safety-critical domain.
- **Problem:** description of the problem addressed by the design pattern.
- **Solution/Pattern Structure:** the solution to the problem under consideration. Main elements of the patterns are described.

- **Consequences:** define the implication or consequences of using this pattern. This section explains both the positive and negative implications of using the design pattern.
- **Implementation:** set of points that should be taken under consideration when implementing the pattern. Language dependent.
- **PatternAssumptions:** the contract assumptions related to the design pattern.
- **PatternGuarantees** the contract guarantees associated to the design pattern.

A remarkable feature is how contracts can be associated to architectural patterns. Having a contract associated to a specific architectural pattern allows deriving some argumentation fragment automatically. Furthermore, the information regarding the implication of using this pattern is collected in a form of assumption/guarantee (i.e. PatternAssumption and PatternGuarantee). Even if the field of design pattern is large, AMASS focuses on applying its usage on safety-critical systems. Hence, the development of fault tolerance design patterns and its usage for different technologies (also known as technological patterns) are some of the addressed AMASS objectives.

To understand how the design patterns are constituted, the following lines introduce the so-called Acceptance Voting Pattern [35] (cf. Figure 12 and Table 1) example. This design pattern is a hybrid software fault tolerance method aiming at increasing the reliability of the standard N-version programming approach.



**Figure 12.** The Acceptance Voting Pattern

**Table 1.** Design pattern template for the Acceptance Voting Pattern

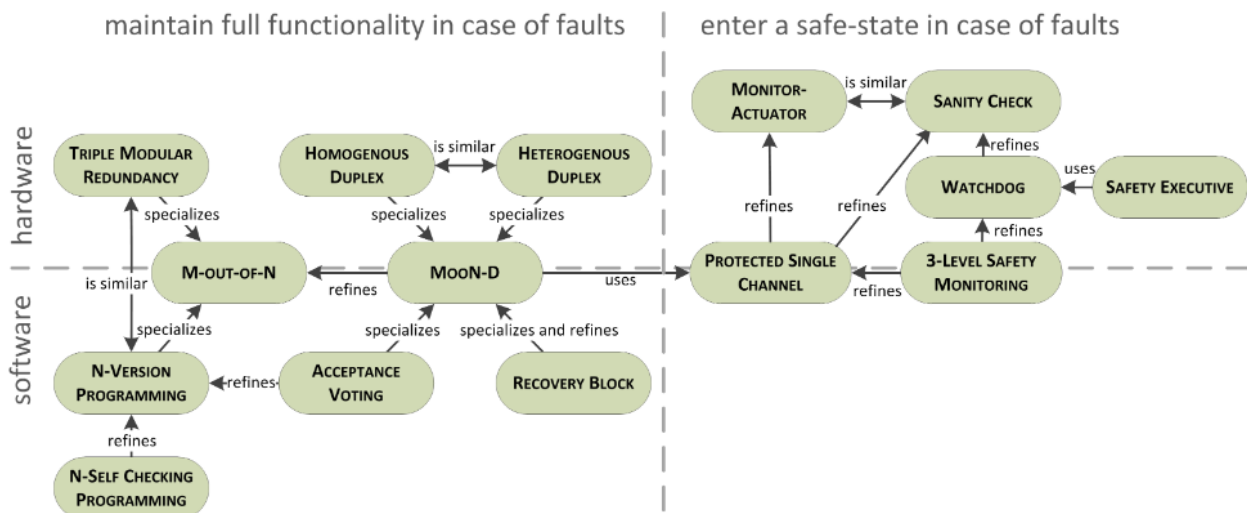
Pattern Name	Acceptance Voting Pattern
Other well-known names	---
Intent/Context	<p>This pattern is suitable to be applied when:</p> <ol style="list-style-type: none"> <li>1. Due to safety reasons, tolerance of software faults is required (i.e. acceptance test).</li> <li>2. High reliability of the system's output is required (several software versions)</li> <li>3. The correctness of the results delivered by the diverse software versions can be checked by an acceptance test.</li> <li>4. The development of diverse software versions is possible regardless additional development costs.</li> </ol>
Problem	Software faults shall be tolerated to achieve safety and reliability requirements.
Solution/Pattern Structure	The Acceptance Voting Pattern (AV) is a hybrid pattern representing an extension of the N-version programming approach by combining this approach with the acceptance test used in the recovery block approach. (cf. Figure 12).
Consequences	<ol style="list-style-type: none"> <li>1. A high dependency on the initial specification which may propagate faults to all versions and effort of developing N diverse software versions.</li> <li>2. The problem of dependent faults in all N software versions is less critical than</li> </ol>

	in the original N-version programming approach.
Implementation	<p>Hybrid patter combining the idea of N-version programming and fault detection using an acceptance test.</p> <ol style="list-style-type: none"> <li>1. The acceptance test should be carefully designed to assure the quality of the acceptance test and to able to detect most of the possible software faults.</li> <li>2. The success of the pattern depends on the level of the quality of the diversity between the N versions to avoid common failures between different versions.</li> <li>3. The voting technique must be implemented by: <ul style="list-style-type: none"> <li>- Majority voting</li> <li>- Plurality voting</li> <li>- Consensus voting</li> <li>- Maximum Likelihood Voting</li> <li>- Adaptive Voting</li> </ul> </li> </ol>
PatternAssumptions	<ol style="list-style-type: none"> <li>1. The majority voting technique is used in the voter software component.</li> <li>2. The failures in the different versions are statically independent.</li> <li>3. The different versions have the same probability of failure (<math>f</math>) and the same reliability (<math>R_i = R</math>).</li> <li>4. The diverse software versions in this pattern are executed in parallel, ideally on N independent hardware devices. Execution time of the acceptance test and the voter is negligible.</li> <li>5. The independent versions followed by the acceptance test and voting algorithm are executed sequentially on a single hardware.</li> </ol>
PatternGuarantees	<ol style="list-style-type: none"> <li>1. The probability that an output passes the test is equal to: <math>P\{T\} = RPTP + (1 - R) PFP</math></li> <li>2. The execution time of this pattern is "slightly" equal to single version software.</li> <li>3. The time of execution will increase by N times of a single version.</li> </ol>

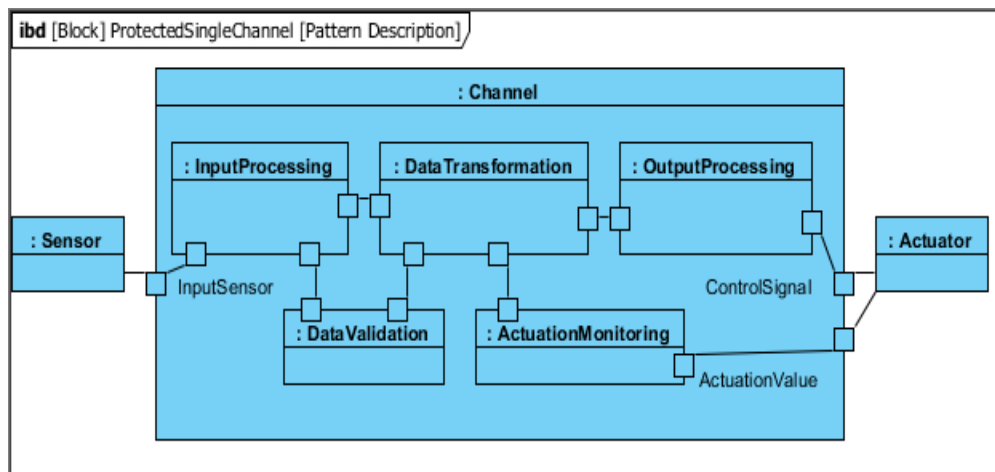
As previously mentioned, among different design patterns, the ones addressing fault tolerance require a special attention in the safety-critical domain. Here, architectural patterns are considered safety measures such as fault detection or redundancy. Such an architectural pattern will be associated with a mechanism to apply it to existing system architecture. For example, a redundancy pattern will be applied to a component by duplicating the component and adding a voter; a communication protection pattern will be applied to a pair of communicating components by adding encoding and decoding subcomponents for the sender and receiver.

### 2.2.1.2 Fault Tolerance Patterns

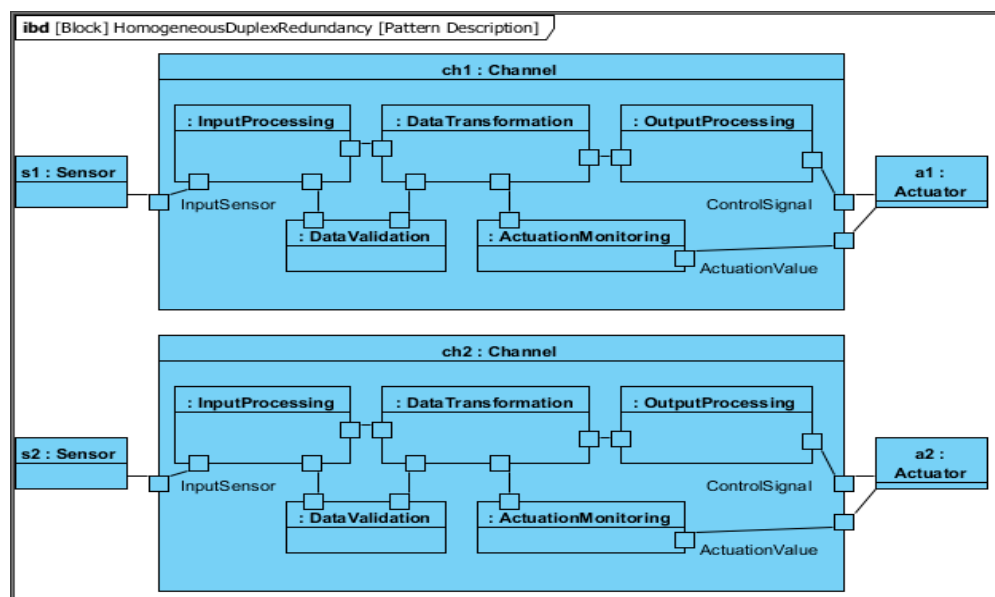
Some of the most remarkable design patterns are the ones adopted in the design of safety-critical systems. Figure 13 tackles the most common architectural patterns for fault tolerance which will be part of a library in AMASS. This catalogue includes a set of hardware and software design patterns which cover common design problems such as handling of random and systematic faults, safety monitoring, and sequence control [46]. For instance, this library will be composed of the following patterns: Protected Single Channel (see Figure 14), Homogeneous Duplex Redundancy (see Figure 15), Homogeneous Triple Modular (see Figure 16), M-out-of-N (see Figure 17), Monitor-Actuator (see Figure 18) and Safety Executive (Figure 19).



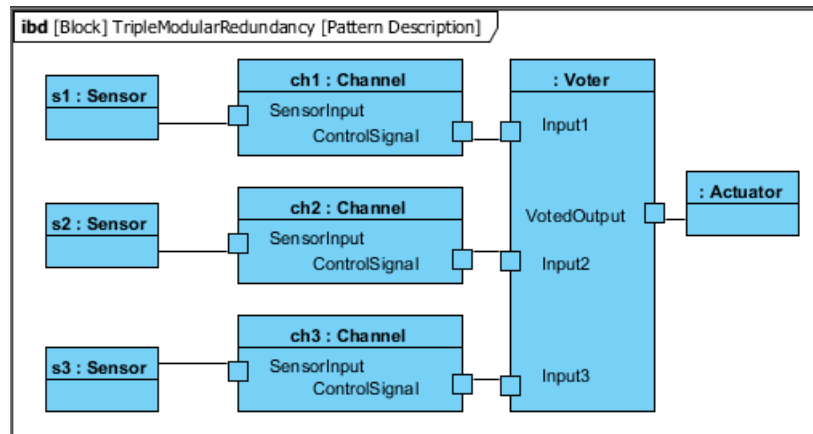
**Figure 13.** Safety architecture pattern system from [46]



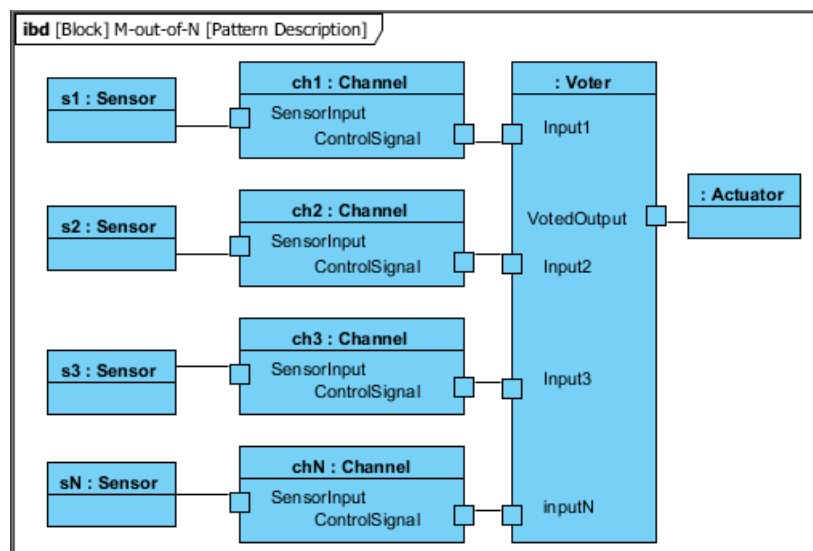
**Figure 14.** Protected Single Channel in SysML



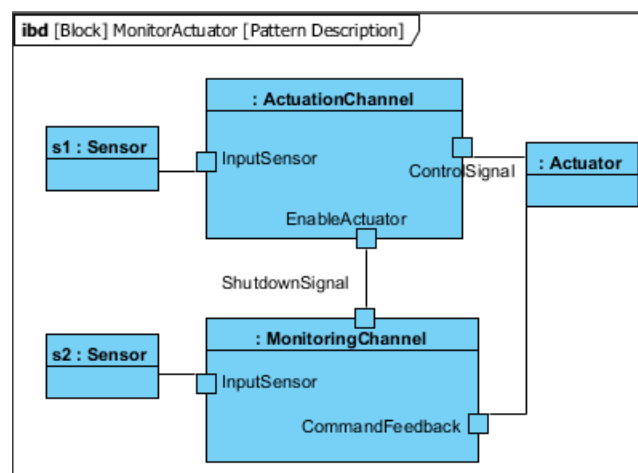
**Figure 15.** Homogeneous Duplex redundancy Pattern in SysML



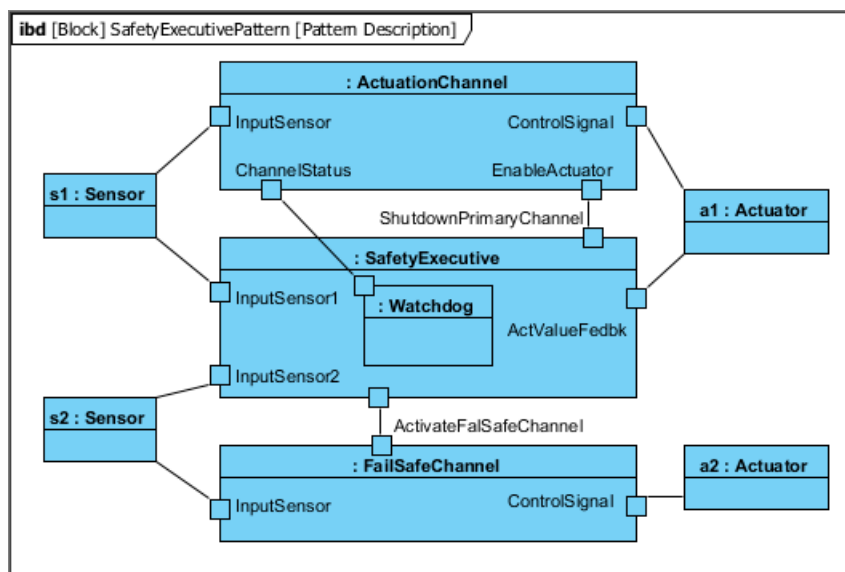
**Figure 16.** Homogeneous Triple Modular Pattern in SysML



**Figure 17.** M-out-of-N Pattern (MoonN) in SysML



**Figure 18.** Monitor-Actuator Pattern in SysML



**Figure 19.** Safety Executive Pattern in SysML

Matos et. Al [45] performed a fault tree analysis of each of the generic design patterns resulting in the following results (cf. Table 2):

**Table 2.** Result of fault-tree analysis of generic design patterns

Design Pattern	Availability	Integrity
Protected Single Channel	3.10E-4	3.00E-5
Homogeneous Duplex Redundancy without fail safe state	9.61E-8	6.00E-5
Homogeneous Duplex Redundancy with defined fail safe state	9.61E-8	9.00E-10
Heterogeneous Duplex Redundancy	Same values that Homogeneous duplex redundancy applies	
Triple Modular Redundancy Pattern (TMR)	2.98E-11	2.70E-9
Monitor-Actuator Pattern (ma)	3.40E-4	6.00E-9
Safety Executive	1.58E-7	3.00E-5

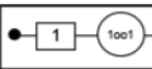
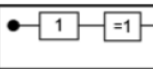
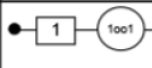
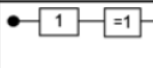
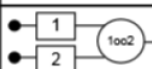

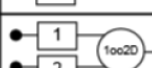
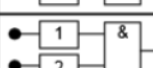
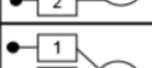
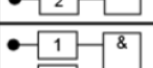


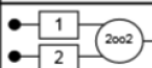

That information can be stored as part of PatternGuarantees which has been previously defined within the proposed design pattern template.

The designer is expected to combine the generic design patterns in order to successfully reach the required dependability/safety level.

### 2.2.1.3 Relation to standards

Fault-tolerant patterns are recommended by different safety standards. When choosing a pattern, it would be important to know which standards recommend such pattern. This information can be stored within the design pattern template and added to the assurance argument. It is therefore important to collect it in the library of patterns.

For example, the IEC 61508 functional standard also recommends different safety architectures. Figure 20 depicts some of the most remarkable safety architecture or architectural patterns to reach the required level of safety and availability. Furthermore, it illustrates how metrics such as average probability of dangerous failure on demand (PFDavg) are calculated depending on the selected architecture.

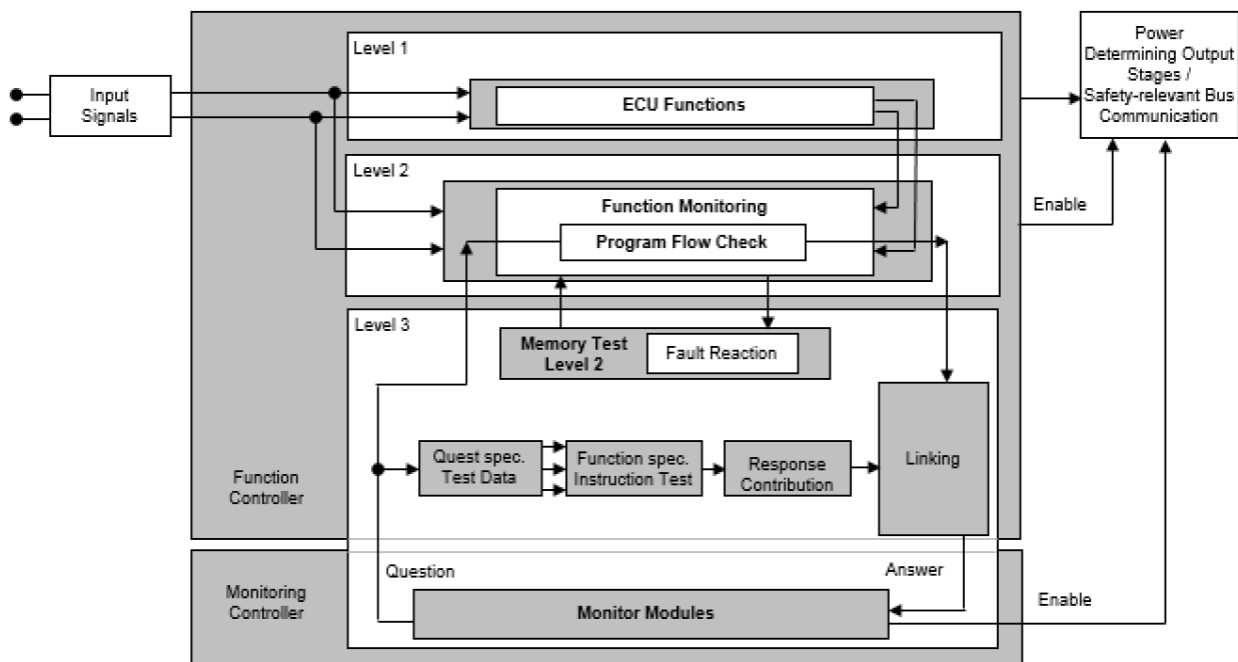
safety architecture	name	shortened definition (source: IEC 61508)	block diagram (source: IEC 61508)	trip input	logic relation for de-energized -to- trip configuration	PFD <sub>AVG</sub> (excl. PC and β)	PFD <sub>AVG</sub> (excl. PC, incl. β)	
1oo1	one out of one	demand or failing element commands output to a safe state		command to safe state	"0"		1/2x(λ <sub>DU</sub> xT)	1/2x(λ <sub>DU</sub> xT)
1oo1	one out of one, Inherent Fail Safe	demand or failing element commands output to a safe state		command to safe state	"0"		λ <sub>D</sub> xMTTR	λ <sub>D</sub> xMTTR
1oo2	one out of two	one demand or one failing element commands output to a safe state		command to safe state	"0"		1/3x(λ <sub>DU</sub> xT) <sup>2</sup>	1/3x((1-β)xλ <sub>DU</sub> xT) <sup>2</sup> + 1/2xβxλ <sub>DU</sub> xT
1oo2D	one out of two, with diagnostics	one demand or simultaneous failing elements command output to a safe state		command to safe state	"0"		1/3x(λ <sub>DU</sub> xT) <sup>2</sup>	1/3x((1-β)xλ <sub>DU</sub> xT) <sup>2</sup> + 1/2xβxλ <sub>DU</sub> xT
1oo3	one out of three	either demand or failing element commands a output to a safe state		command to safe state	"0"		1/4(λ <sub>DU</sub> xT) <sup>3</sup>	1/4x((1-β)xλ <sub>DU</sub> xT) <sup>3</sup> + 1/2xβxλ <sub>DU</sub> xT
2oo2	two out of two	two demands or simultaneous failing elements command output to a safe state		command to safe state	"0"		λ <sub>DU</sub> xT	(1-β)xλ <sub>DU</sub> xT + 1/2xβxλ <sub>DU</sub> xT
2oo3	two out of three	two demands or two failing elements command output to a safe state		command to safe state	"0"		(λ <sub>DU</sub> xT) <sup>2</sup>	((1-β)xλ <sub>DU</sub> xT) <sup>2</sup> + 1/2xβxλ <sub>DU</sub> xT

**Figure 20.** Safety architectures in IEC 61508

Regarding the automotive domain, ISO 26262 [34] is the relevant functional safety standard. The Appendix D: Design patterns for fault tolerance applied to technology according to ISO 26262 of this document collects the information of ISO 26262 Part 5 - "Product development at the hardware level". In Annex D "Evaluation of the diagnostic coverage", chapter D.2 "Overview of techniques for embedded diagnostic self-tests" a safety mechanism is described which is related to the fault tolerance patterns introduced in chapter 2.2.1.2.

The Standardized E-Gas Monitoring Concept or 3-Level Safety Monitoring Pattern (3-LSM) (cf. Figure 21) for Gasoline and Diesel Engine Control Unit [33] is in the automotive domain a well-established safety architecture for drive by wire systems. The E-Gas Monitoring Concept is based on a 3-Level Monitoring Pattern. The system overview is presented in Figure 21.





**Figure 21.** System overview E-Gas Monitoring Concept [33]

The first level is called function level, which contains the control functions. It also includes component monitoring and input-/output-variable diagnostics. If a fault is detected, the level 1 controls the system fault reaction.

The second level monitors the function level and is called function monitoring level. It detects the defective process of level 1, e.g. by diverse calculation or estimation of the output-values and applying range checks to level 1 values. In case of detected faults, level 2 is either able to apply referring reactions by itself or it initiates a reaction carried out by level 1.

The third level, the controller monitoring level, monitors the controller for its integrity. The monitoring module can be seen as an independent part, e.g. an application-specific integrated circuit (ASIC), a watchdog, or another controller are frequently requesting specific answers to questions, which guarantee a proper functioning of the controller.

Safety-concepts for common-purpose applications in the automotive domain are derived from the E-Gas monitoring concept for gasoline and diesel engine control unit.

At the same time, the aforementioned design patterns can be applied w.r.t. different technologies like analogue and digital I/O, processing units or non-volatile memory. When doing so, these patterns are known as technological patterns.

Standards can be used to complete the information contained in a form of PatternAssumption and PatternGuarantee. For instance, the specific safety mechanisms described in ISO 26262-5 Annex D (evaluation of the diagnostic coverage) can be considered as technological patterns to obtain a certain degree of diagnostic coverage (PatternGuarantee) where the described notes are a certain number of assumptions (PatternAssumption). The table in Appendix D: Design patterns for fault tolerance applied to technology according to ISO 26262, depicts how the relationship between general design patterns for fault tolerance (architectural patterns) are related to technology and a specific functional safety standard. For example, by applying HW redundancy by means of a lock step (1oo2D architectural pattern applied to processor technology), a high diagnostic coverage is guaranteed assuming that it depends on the quality of redundancy and that common mode failures can reduce diagnostic coverage. For more information, patterns for fault tolerance applied to technology according to ISO 26262 can be found in Appendix D. AMASS will develop a meta-model defining a new formalism for describing patterns which will consider



generic architectural or fault tolerance patterns together with its relation to technology and functional safety standards.

### 2.2.2 Parametrized architectures for architectural patterns

Architectural patterns can be implemented by supporting the definition and instantiation of a *parametrized architecture*. Here we propose to use the notion defined by FBK within the project CITADEL (see deliverable D3.1 of CITADEL<sup>4</sup>). A parametrized architecture is an architecture in which the set of components, ports, connections, and attributes depends on the values of some parameters. For example, a system in a parametrized architecture has static attribute  $n$ , an array  $p$  of  $n$  Boolean attributes, an array  $a$  of  $n$  subcomponents of type  $A$  and an array  $b$  of  $n$  subcomponents of type  $B$ ; each component of type  $A$  has an integer attribute  $m$  and an array  $q$  of  $m$  output ports and each component of type  $B$  has an integer attribute  $m$  and an array  $q$  of  $m$  input ports; the system has a connection from  $a[i].q$  to  $b[i].q$  if  $p[i]$  is true for every  $i$  between  $0$  and  $n-1$ .

A *configuration* of a parametrized architecture is given by assignment to all parameters. For example, a configuration for the above-mentioned parametrized architecture can be  $n=2$ ,  $p[0]=true$ ,  $p[1]=false$ ,  $a[0].m=1$ ,  $b[0].m=1$ ,  $a[1].m=1$ ,  $b[1].m=1$ . Given a configuration, one can instantiate the parametrized architecture with the given parameter values obtaining a standard architecture.

A *configuration constraint* is a constraint over the parameters that restrict the valid configurations to those parameter values that satisfy the constraint. For example, in the above parametrized architecture, we can consider the constraint *for all  $i$ ,  $0 \leq i < n$ ,  $a[i].m = b[i].m$* . Note that this constraint is satisfied by the configuration example described above.

Parametrized architecture can be used as model for an architectural pattern. It can be also used to formalize specific architectures prescribed by some standards. For example, in the railways, the ETCS define the structure that is at the basis of the interoperability between on-board and trackside systems; in the space, the PUS defines the telemetry/telecommands that are exchanged between on-board and ground systems. An architectural pattern in this case can be predefined and instantiated by setting the values of specific parameters according to the application needs.

#### 2.2.2.1 Employment in AMASS

CHESS will be extended to support the specification and instantiation of parametrized architectures. SysML part associations with multiplicity different from 1 will represent arrays of subcomponents and multiplicity \* will represent a symbolic number of components. Parameters will be declared as attributes of block. Constraint blocks will be used to define parametrized connections and to constrain further the value of parameters. OCRA and the translation from CHESS to OCRA will be extended as well to support parametrized architectures so that it will be possible to analyse the architecture in different configurations.

## 2.3 Contract-Based Assurance Composition

### 2.3.1 Contracts Specification

A contract specifies the assumption and guarantee of a component. Its specification takes as input the component interface and generates two formal properties, which are respectively the assumption and the guarantee. Therefore, the contract specification is intrinsically connected to the property specification (see Section 2.4.2).

The component interfaces define the elements (component ports) that can be used to form the assumptions and guarantees. The Architecture-Driven Assurance component of the AMASS platform will

---

<sup>4</sup> <http://www.citadel-project.org>

have a Contract-Based Design subcomponent to provide standard advanced editing features such as syntax highlighting, auto-completion for contracts.

The Contract-Based Design subcomponent will interact with external tools such as OCRA to produce evidence of the correctness of the system architecture based on the contract specification (including contracts validation, contracts refinement, compositional model checking, and contract-based fault-tree generation); see also Chapter 2.4.3.2.

### 2.3.2 Reuse of Components (\*)

The ultimate goal of contract-based design is to allow for compositional reasoning, stepwise refinement, and a principled *reuse of components* that are already pre-designed, or designed independently. This approach has been adopted also in the SafeCer ARTEMIS project<sup>5</sup> to exploit contracts to enable a compositional certification and reuse of pre-qualified components.

In this context, the behavioural model of a component is verified to satisfy the contract associated to that component. In order to be reused, the behavioural model must also be compatible with the environment of the component provided by the system design. We propose to exploit the contract specification to ensure that the component implementation is compatible with any environment satisfying the assumption of the contract.

### 2.3.3 Contract-Based Assurance Argument Generation (\*)

The bases of assurance cases are requirements. The goal of an assurance case argument is to explain how the supporting evidence satisfies the requirements. Component contracts are tightly coupled with the requirements allocated to those components and the evidence supporting such contracts. Reuse of safety-relevant components is incomplete without reuse of the accompanying evidence and arguments. We have defined during SafeCer and SYNOPSIS [53] projects a component meta-model to capture the relation of the contracts, evidence and the requirements such that assurance argument-fragments can be generated by automatically instantiating the pre-established argument patterns from the system models compliant with the component meta-model [52]. The meta-model uses strong and weak contracts to support reuse by allowing for more fine-grained association of assurance information with the contract specification. Contract checking using OCRA does not support distinction on strong and weak contracts. Hence such contracts need to be first translated into appropriate format for the contract checking. Since the contract refinement and validation checks on such translated contracts do not offer all the information needed about the weak contracts for the argument-generation, additional checks need to be performed on the weak contract specifications.

MDH plans to contribute to developing the user interface for capturing the strong and weak contracts and the associated information on the user level as described in the component meta-model. Then we plan to define the translation of the captured information to the OCRA format such that it is possible to get from OCRA all the necessary information to perform the argument-pattern instantiation from the contracts and the associated information. Besides the refinement and validation checks on the translated contracts, OCRA results should also include validation checks of the weak contract specification such that it is possible to identify which weak contracts are satisfied (i.e., their assumptions are fulfilled) in the context of the current system.

Further details about assurance patterns for contract-based design are provided in Chapter 2.5.

---

<sup>5</sup> <https://artemis-ia.eu/project/40-nsafecer.html>

## 2.4 Activities Supporting Assurance Case (\*)

### 2.4.1 Requirements Formalization with Ontologies

#### 2.4.1.1 Ontology as Specification Language

The system engineers need to have considerable knowledge and experience in the domain in order to define the system requirements and design the system architecture. They also need to have clear vision about the ultimate result of the development effort that will raise from the implementation of their system architecture and requirements. This section provides yet another example, how the system architecture and requirement authoring activities can be based on an ontology and what benefits that brings (e.g. the ontology can be used as a unified consistent language).

The ontology is perceived as a kind of specification language, which offers the user:

- A list of textual expressions (names of types/sorts/classes of things, names of individual things/values/parameters/constants, names of processes, names of relations, possibly supplemented with corresponding definitions of these concepts and bound to examples of contexts in which they occur) tightly related to (stemming from) the application domain. Such list of symbols is sometimes called the *signature*. The signature helps to suppress the ambiguity of the text, e.g. by limiting the usage of several synonyms for the same thing, which is okay when stylistic issues are important, but which is undesirable from an engineering point of view. The symbols of the signature together with the symbols of the logical operations like conjunction, negation, etc. of the chosen logic represent the constituent blocks of formal system specifications.
- The possible compositions of the lower-level expressions into higher-level expressions and even into the whole sentences. These potential compositions are inscribed in the diagrammatic structure of the underlying ontology captured in the UML. When the sentences are created, the user traverses appropriate continuous paths in the UML diagrams from one concept (class) to another concept via the existing connections (associations, generalizations, aggregations) and composes the names of classes and relations encountered along the way into a sentence.

#### 2.4.1.2 Employment in AMASS

The basic structure of the process to apply ontology in the system architecture creation and requirement authoring and formalization can be summarized in the following steps:

1. Create the (UML) ontology.
2. Write a (tentative, sketchy) informal system architecture and requirement.
3. For the most important notions of the informal system architecture and requirement, find the corresponding terms in the ontology.
4. Select the most appropriate paths in the ontology graph, which connect/include the important notions found in the previous step.
5. Compose meaningful sentences by concatenating the names of the elements (classes, objects, relations) traversed along the selected paths.
6. Repeat the steps 2 – 5 for all informal artefacts.

This approach is also applicable when the goal is to improve the quality of the current system architecture or requirements and rewrite them in a more clear and consistent form. In that case, the step 2 does not involve writing of new parts of system architecture or requirements, but taking the existing artefacts from their last iteration.

If we consider an extreme but desirable case that a new system's architecture is just a composition of reused components that have been developed separately around different ontologies, it is obvious that the ontology of the new and more complex system should be some composition of the simpler ontologies of its

subsystems. Therefore, for the development of Cyber-Physical Systems it is highly desirable to have appropriate means of gluing ontologies together to obtain ontologies that are more complex.

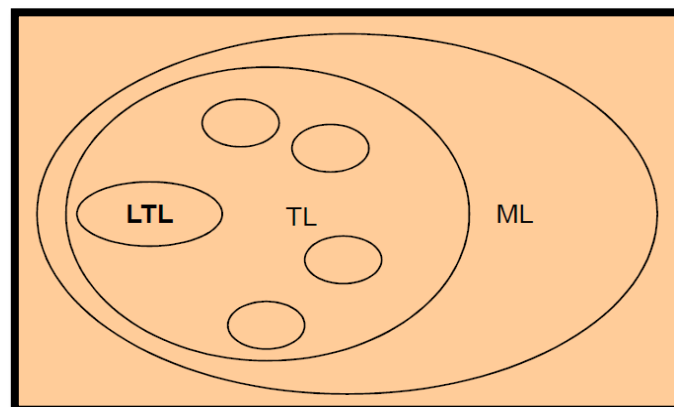
## 2.4.2 Requirements Formalization with Temporal Logics

Both requirement authoring supported with domain ontology and requirements formalization increases the quality of requirements and improves the capacity later to verify compliance to these requirements. In the case of requirements formalization, the benefits that this process brings are automatic formal verification, guaranteed verifiability, and the removal of ambiguity among requirements.

### 2.4.2.1 Temporal Logics as Specification Language

*Linear Temporal Logic* (LTL) first introduced by Pnueli [14] is more and more used in the Requirements Engineering to validate and verify requirements. LTL is a Modal Temporal logic, within the boundaries of Model Logic (ML) (see Figure 22), with operators or words that refer to time, which is represented linear and discrete, allowing, thus, formality, accuracy and unambiguity. With LTL it is possible to represent several modes, such as:

- Safety: to make sure that something bad will never happen.
- Liveness: to make sure that something good will happen.
- Fairness or correct equity in the distribution of resources.
- Reachability or proper access to certain state or resource.
- Deadlock freedom or no blockings.



**Figure 22.** Linear Temporal Logic (LTL) boundaries within Modal Logic (ML)

More formally, linear temporal logics (or LTLs for short) is an extension of classical logic including the following temporal operators:  $X \varphi$  (in the next moment in time  $\varphi$  is true),  $F \varphi$  (eventually  $\varphi$  is true),  $G \varphi$  (always in the future  $\varphi$  is true),  $\varphi U \psi$  ( $\psi$  is true at some moment in the future, and until  $\psi$  becomes true,  $\varphi$  is true). From these temporal modalities, various other operators can be derived such as a weak version of  $\varphi U \psi$ , denoted  $W$ .

The satisfiability problem for LTL is to decide, given an LTL formula  $\varphi$ , if there exists a model for which  $\varphi$  holds. Several techniques for determining satisfiability of LTL have been developed: tableau-based methods (e.g., [16], [17]), temporal resolution (e.g., [19][20]), and reduction to model checking (e.g., [18]).

In general, one of the tasks of the Requirements Analysis is to check the consistency of a given set of requirements. Such check can be usually automated by employing some consistency checking tool.

For example, if we assume that the requirements are expressed using LTL formulas, then the problem of checking consistency can be reduced to a model checking problem and one of several existing model checking tools can be employed.

An important extension of LTL is the version in which every future temporal operator has a dual past operator. So, “Y p” means in the previous time point “p” holds (and the current one is not the initial timepoint), “O p” means once in the past “p” held, “H p” means historically (always in the past) “p” held, “p S q” means “q” was true in the past and since then “p” remained true.

Metric Temporal Logic (MTL) is an extension of LTL where temporal operators can constrain the distance between two time points. So, for example “F [a,b] p” means that “p” will be true in the future in a time point which is between “a” and “b” time units from now. There are many variants of MTL. For example, Event-Clock Temporal Logic is a variant in which only the next occurrence of a state satisfying a property can be constrained. For example, “ $\blacktriangleright$  [a,b] p” means that the next time point in the future where “p” holds is between “a” and “b” time units from now.

An important aspect of the formal semantics of models and temporal logics is the model of time. For discrete-events systems, the time is seen as a discrete sequence of time points. In real-time and hybrid systems, the domain of time is given by the real numbers and may be strictly or weakly monotonic (also called super-dense): time always increases or it does not decrease but instantaneous changes/events are also considered.

OCRA provides an English expressions as alternative to express future and past temporal operators and event-clock operators: so “F p” can be written also as “in the future p”, “p S q” can be written as “p since q”, “ $\blacktriangleright$  [a,b] p” can be written as “time\_until(p)>=a and time\_until(p)<=b”.

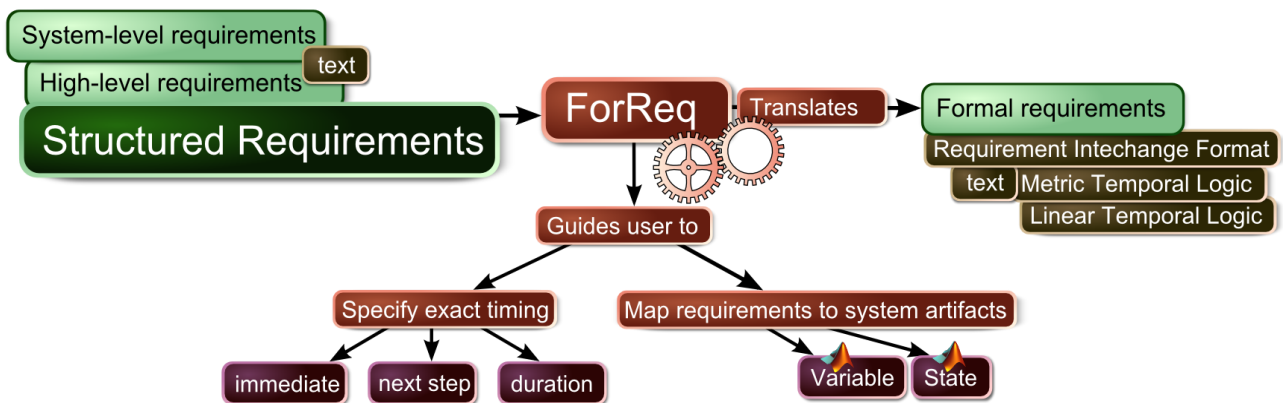
#### **2.4.2.2 Formalization with ForReq**

In some aerospace domains, e.g. Flight Controls, Flight Management Systems, Display and Graphics, the Honeywell requirements are written in a structured and restricted way to improve their quality. Yet these restrictions are not sufficient to guarantee machine readability and the subsequent automatic verification. The requirements language needs to be further restricted to be unambiguous and to have clear semantics, before a machine could read such.

Honeywell internal tool ForReq [22] allows requirement authoring based on a grammar for structured English requirements that serves two separate purposes. For the requirements already written that conform to this grammar, ForReq allows automatic translation into Linear Temporal Logic (LTL) and thus automatic verification. Yet, more importantly, the machine readability can be enforced for new requirements by the use of *auto-completion*. This new functionality suggests the requirements engineer the set of possible words to continue the requirements definition process. Thus, the requirements engineers save effort that would be needed for writing twice each of the requirements, i.e. the human readable version for stakeholders and the machine-readable version for verification.

In the case requirements do not use exact artefacts (variables or states) from the system (for example some system requirement are prohibited to contain such link), the ForReq tool now guides the user to create mapping from artefacts in requirements to the corresponding artefacts in the system. Moreover, requirements defining mapping between variable names and its textual descriptions used in requirements are supported to automate fully the process. These requirements are also verified and any inconsistency is reported to be fixed by the user.

However, the user has to specify the exact timing of each requirement, i.e. whether the effects shall happen immediately or in the next time step or after specified number of time steps or seconds. Honeywell ForReq tool supports this requirement formalization process as depicted in Figure 23 in order to enable automatic semantic requirement analysis as described in Section 2.4.3 and automated formal verification against system design as described in Section 2.4.5.



**Figure 23.** Process of formalization of structured requirements using ForReq tool

The requirements formalization is not a straightforward process and a considerable number of steps is required for incorporating formalization into real-world development of embedded systems. The goal of the ForReq development is to guarantee that the authored requirements are unambiguous, automatically verifiable (machine-readable) and conforming to the requirements reference (template, pattern, boilerplate, standard). Auto-completion, requirement standard grammar and requirement guidelines were implemented in ForReq to cover this need and to proceed further towards fully incorporating requirements formalization into the development process.

#### 2.4.2.3 Formalization with System Quality Analyzer (SQA)

This functionality is aimed at presenting and proving the applicability of a proposal of a model that, automatically, analyses the quality of a requirements specification regarding to its temporal consistency.

The System Engineering (SE), interdisciplinary field focused on developing successful systems, defines as necessity the assessment of requirements specifications that define a system or system of systems. For that purpose, within the SE responsibilities, the Requirement Engineering (RE) has the objective of ensuring the quality of the requirements, aside of defining a writing guide.

Measuring requirements quality consists of guarantying that they contain certain textual, syntactical and structural characteristics. This approach, better known as *Correctness*, is applied to the requirements individually, looking for defects in the text. On the other hand, there are two other approaches regarding analysing requirements quality, both applied globally in specification-level: first, *Completeness*, aimed to find omitted elements which will be involved in the specification, and *Consistency*, to ensure that there is no incoherence between requirements.

Using the scientific foundations of Natural Language Processing (NLP) together with the bases of the industry dedicated to Systems Engineering and Requirements Engineering, it has been defined a model that can solve the actual need: knowing and analysing the temporal elements found in the requirements in order to make a requirement temporal validation and verification as well as an early detection of temporal inconsistencies.

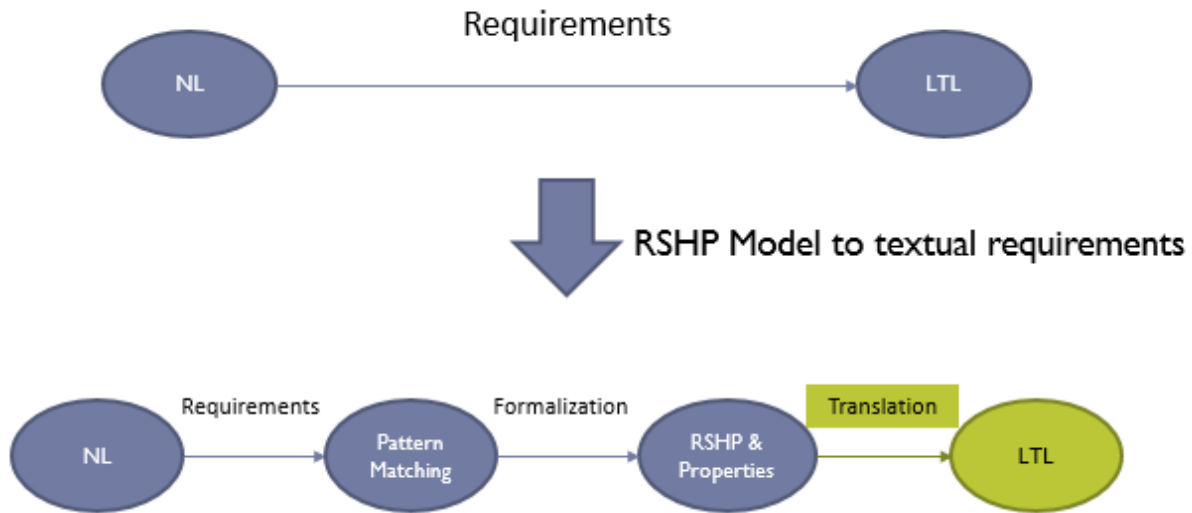
The assessment of the temporality used in the language has been done with *Linear Temporal Logic* (LTL). The proposal consists of a NLP software mechanism applied to textual requirements in order to make a quality assessment, in terms of temporal consistency.

The quality assessment is an automatic translation from requirements written in Natural Language to LTL, which is not a simple task. In this case, the mechanism used for that purpose is the RSHP Model [23] [24] applied to textual requirements. This model allowed an easier translation to LTL, and it has been the mayor point.

To summarize, this functionality looks for elements representing time in the requirements and then checks that they do not present temporal conflicts. This process starts by formalizing requirements using certain



writing patterns, with the objectives of extracting relevant information from them (concepts, relationships and properties), store and reuse them thanks to RSHP.



**Figure 24.** Automatic translation general diagram - From NL to LTL

According to the defined and presented proposal, it is performed an experiment consisting in the analysis of a requirement specification, detecting its temporal elements and then determining their temporal reachability.

The *System Quality Analyzer* tool, through its available API, allows users to implement and incorporate externally developed applications in order to get different or non-covered by the tool goals. Because of this extensibility, a showcase has been done to demonstrate the proposed model.

This showcase is based on the *Shared Resource Arbiter* problem, consisting on the evaluation of a resource that is shared by several components at the same time, and guarantying its correct distribution among them. Furthermore, it has been checked that a requirements specification containing simultaneously-accessed elements guarantees its reachability and accessibility to them, as well as avoiding deadlocks.

The final result for this showcase has been successful, obtaining an application implemented and integrated in SQA [27], so that a user can create and configure it as a temporal consistency metric. This way, it is feasible to analyse requirements specification, whose results will be helpful to the Quality Assurance department of the company to early-detect temporal inconsistency defects, and continue with the rest of layers in the system lifecycle.

#### 2.4.2.4 Employment in AMASS

In AMASS, we will use LTL and MTL as formal properties to formalize requirements. These properties have a formal semantics that is parameterized by the time model: if the model of time is discrete, the properties are interpreted over discrete-time execution traces; if the model of time is continuous, the properties are interpreted over continuous-time execution traces. A property is attached to a component, which defines the set of variables (events, data ports, etc.) that can be used in the property.

The AMASS platform will provide editing support to formalize the requirements into temporal logic and to import the properties from external tools such as ForReq used for the formalization.

The different editors can use different syntactic sugar or patterns or constrained natural language, but the properties will be internally mapped to LTL/MTL as defined in Appendix A: LTL/MTL.

## 2.4.3 Semantic Requirements Analysis

### 2.4.3.1 Tools for Semantic Requirement Analysis

Formalisation of software requirements, i.e. translation from human language into a mathematical formalism, can simplify the delivery of high quality requirements. Such requirements have properties that greatly reduce the cost of the following development process. Ideally, requirements should be accurate, atomic, attainable, cohesive, complete, consistent, etc. Some of these properties can be presently achieved using automatic tools for semantic requirements analysis. More details on which properties are provided by the AMASS tools can be found in the deliverable D3.1 [2], Section 3.7.1.

### 2.4.3.2 Contract-Based Analysis

Besides the validation checks mentioned above based on consistency, possibility and assertion problems, the contract-based specification enables further analysis:

- **Refinement checks:** for each requirement derivation, it is possible to check the completeness of the requirements in order to ensure the guarantees of the upper level and the assumptions of the lower level.
- **Safety analysis:** taking into account the possibility of failures, a fault-tree analysis is performed to identify single-point of failures and to analyse in general the system reliability. This is possible also without a behavioural model of the components as described in [13], but with a fault injection of the behavioural model the analysis can be more precise.

The verification of contracts refinement adapts existing formal methods developed for checking the satisfiability of properties. In fact, as described in [54], the problem of checking the refinement of contracts is reduced to a series of validity/satisfiability problems for the underlying logic that is reduced to model checking.

In order to prove the validity of the proof obligations deriving from contract refinement, OCRA interacts with nuXmv in case of LTL contracts and with HyCOMP in case of HRELTL contracts.

### 2.4.3.3 Exploring the Inconsistency of Requirements

One of the tasks of Requirements Analysis is to check a given set of requirements for consistency. Such check can be usually automated by employing some consistency checking tool. For example, if we assume that the requirements are expressed using LTL formulas, then the problem of checking consistency can be reduced to a model checking problem and one of several existing model checking tools can be employed.

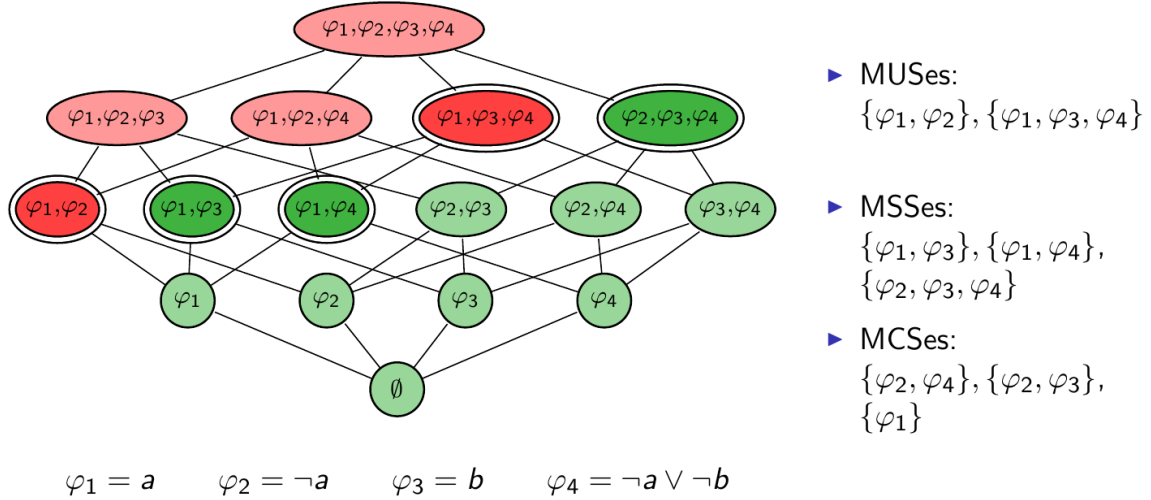
In the favourable case, the given set of requirements is consistent. In the other case, it is desirable to find out the reason for the inconsistency.

Nowadays, there is an active research on analysing the inconsistency in the area of constraints processing. However, a slightly different terminology is used. Instead of using “consistent” the set of requirements/formulas is said to be satisfiable or unsatisfiable, respectively. There are three concepts that are used to explain the unsatisfiability:

- **Maximal Satisfiable Subset (MSS):** for a given unsatisfiable set of requirements  $U$  and its subset  $N$ , we say that  $N$  is a maximal satisfiable subset of  $U$  if  $N$  is satisfiable and none of its (proper) supersets is satisfiable.
- **Minimal Unsatisfiable Subset (MUS):** for a given unsatisfiable set of requirements  $U$  and its subset  $N$ , we say that  $N$  is a minimal unsatisfiable subset of  $U$  if  $N$  is unsatisfiable and none of its (proper) subsets is unsatisfiable.
- **Minimal Correction Set (MCS):** for a given unsatisfiable set of requirements  $U$  and its subset  $N$ , we say that  $N$  is a correction subset of  $U$  if  $U - N$  is satisfiable (i.e.  $U$  becomes satisfiable by removing elements of  $N$  from it). Moreover,  $N$  is said to be a minimal correction subset of  $U$  if none  $N$ 's (proper) subsets is a correction subset of  $U$ .



Intuitively, MSSes are the maximum which can be simultaneously satisfied, MUSes are the core sources of the unsatisfiability, and MCSes are the minima that must be removed from an unsatisfiable set of requirements in order to make is satisfiable. Note that the maximum/minimum is meant with respect to the subset inclusion rather than cardinality of the sets.



**Figure 25.** Requirements Analysis Example

Figure 25 illustrates these concepts on a simple example; for clarity only basic propositional formulas with two variables are used. Each variable corresponds to some property of the system, for example the first formula states that the system shall have the property  $a$  and the last formula states that the system shall not have the property  $a$  or shall not have the property  $b$ .

The identification of MUSes, MCSes and MSSes of an unsatisfiable set of requirements can help one to properly understand the reasons of the unsatisfiability and it provides suggestions about how to fix the requirements.

There are existing tools that for a given set of unsatisfiable LTL formulas enumerate its MUSes, MSSes and MCSes. We propose to define an interface which calls these tools in order to support the refinement of unsatisfiable (inconsistent) sets of requirements.

#### 2.4.3.4 Requirements Sanity Checking

Specifying functional system requirements formally, e.g. in Linear Temporal Logic, allows for clear and unambiguous description of the system under development. In order to further improve the assurance that the architecture build using these requirements is correct, we suggest checking sanity of those requirements. In our setting, sanity consists of three components: consistency checking, redundancy checking, and checking the completeness of requirements. These notions and the methods for their automation are described in more detail in the deliverable D3.1 [2], Section 3.7.2.

#### 2.4.3.5 Checking Realisability of Requirements (\*)

Consistency of requirements demonstrates the existence of a model that satisfies all those requirements. In the case of LTL requirements, the model is a labelled transition system, where each state label represents the set of atomic propositions that hold in that state. These atomic propositions in turn represent the validity of relations between variables (in the case of software) or signals (in the case of reactive systems). Consequently, the fact that a set of requirements is consistent merely states that, if we were able to prescribe the validity of relations between signals in individual states of the system, we would be able to build such system. Yet in the case of input signals or variables, we often cannot arbitrarily predetermine their values.

Realisability of requirements acknowledges the distinction between input and output signals in that output can be controlled but input cannot. For reactive systems in particular, realisability is more relevant than consistency, because consistency does not guarantee that a system can be built that would satisfy the requirements. We can exemplify the difference between consistency and realisability on the development of a system with one input signal `in1` and one output signal `out1`. Let us further assume two simple requirements: 1) that whenever the input `in1` is observed, the output `out1` is produced within 2 seconds, and 2) output `out1` must never be produced. These two requirements are consistent because there is a sequence of signal evaluations that satisfies both of them, i.e. input `in1` is never observed and output `out1` is never produced. On the other hand, the requirements are not realisable because the input `in1` can be observed and then the system has to choose between producing `out1`, thus violating requirement 2; or not producing `out2`, thus violating requirement 1.

Checking realisability, however, is more difficult than checking consistency [48] and is effectively equivalent to synthesising the appropriate system. This introduces additional challenges in terms of limited scalability as well as the potential to utilise the synthesised system. The output of realisability checking (synthesis) is a transition system labelled with a combination of input and output signals. If realisable, each transition (labelled  $i,o$ ) prescribes how the system needs to react (produce output  $o$ ) to a particular input  $i$ . If unrealisable, each transition (labelled  $i,o$ ) prescribes how what input  $i$  does the environment have to produce to eventually force the system to violate the requirements. Thus, in order to check realisability the tool has to calculate the strategy for either the system or the environment, i.e. effectively to construct the correct-by-construction system or to prove that none exists.

There are two possible approaches of tackling the computational complexity of realisability checking. First, there is a number of approaches and tools that implement realisability checking. These tools use different algorithms and thus they have different efficiency when solving particular problems. Hence, we intend to compare relevant tools on a selected benchmark of requirements to determine which tool is most suitable for our domain. In particular, Acacia+ improves on the classical automata-based approach and Party-Elli employs bounded synthesis to decide realisability.

The second approach is based on the fact that the high complexity relates to the full LTL language. There are, however, interesting fragments of LTL for which the problem of realisability is easier. In this regard, we will investigate the smallest fragment of LTL sufficient for our requirements and then assess whether algorithms that are more efficient could be used. In particular, the GXW fragment can be checked for realisability by the Autocode tool.

The third approach is to translate the requirements realisability problem to a software verification problem. Specifically, we intend to generate a valid C code fragment for each requirement. When this C code is symbolically executed by a static analyser one of two possibilities can occur. Either the value of input variables was restricted during each execution or there is at least one execution in which the inputs are unrestricted. In the first case the requirements are not realisable, while in the second they are realisable.

The ability of the realisability checking tools to synthesise the proof of realisability in terms of a valid system provides a number of potential applications. In the case the requirements are not realisable, the system represents the behaviour of the environment (user input) that leads to the violation of the requirements. This system thus represents the counter-example that the developer can use to identify the cause of requirements violation. In the case the requirements are realisable, the system represents one of the possible valid implementations of the requirements. We intend to investigate this case further as well, to detect requirements which are trivially realisable and to assess the completeness of the requirements. The completeness of requirements will be measured in terms of the complexity of the input-output behaviour. Requirements will be called trivially realisable when they are satisfied by a system with simple input-output behaviour.

#### 2.4.3.6 Employment in AMASS (\*)

The Architecture-Driven Assurance of AMASS will have a Requirements Validation component which will interact with different tools to provide evidence that the requirements specification is correct. The provided functionalities will include consistency checking, possibility checking, assertion checking, contract-based refinement checking, contract-based safety analysis, and possibly also redundancy checking and realisability checking. This sanity checking package will be maintained separately from the AMASS interface thus ensuring that the latest and the most comprehensive analysis is executed. The verification results will include execution traces, MUSs, and possibly also MSSs and MCSs. The results of the analysis will be traced to be used in the assurance case.

### 2.4.4 Metrics

#### 2.4.4.1 Metrics for requirements

While assessing the quality of the requirements there are different approaches (points of view, or simply, views):

- *Correctness* summarizes the set of desirable individual characteristics of a correct requirement:
  - Understandability: requirements are clearly written and can be properly understood without difficulty.
  - Unambiguity: there is one and only interpretation for each requirement (unambiguity and understandability are interrelated; they could be even the same characteristic).
  - Traceability: there is an explicit relationship of each requirement with design, implementation and testing artefacts.
  - Abstraction: requirements tell what the system must do without telling how it must be done, i.e. excess of technical details about the implementation must be avoided in the specification of the requirements.
  - Precision: all used terms are concrete and well defined.
  - Atomicity: each requirement is clearly determined and identified, without mixing it with other requirements.
- *Completeness* means there are no omissions that compromise the integrity of the specification: all needed requirements, with all needed details. Completeness is easily defined, but not that easily ascertained, because we do not have a source specification with which to compare (it is obvious: we are creating the source specification). Besides, a specification is not usually a static set of requirements.
- *Consistency* means there are *no contradictions* among requirements. Like completeness, consistency is not easily achieved, since contradictions may be very subtle, especially in the presence of ambiguous requirements. A good organization of requirements is essential to achieve consistency. A classical method is the use of a *cross-reference consistency matrix*, where each pair of requirements is labelled as: **Conflicting (x)**, **Redundant (=)**, **Overlapping (+)**, **Independent**.

For each point of view, several metrics are provided to measure its quality.

As result of the participation of TRC and UC3 in the CRYSTAL project [50], TRC tools were improved with new correctness, completeness and consistency metrics. Now in AMASS this set of metrics has been extended and improved.

In this section, these changes will be revised.

#### Correctness Metrics

There are different metrics based on Requirement Management Systems, on simple textual content and structure, on System Knowledge Base and on Special Sentences to measure correctness point of view.

Moreover, users are able to implement and integrate their own metrics according to their own needs through the Custom-coded metrics.

While analysing text in natural language with the help of a knowledge base, there are different aspects to focus in:

- The terminology, the actual word or phrase in the text
- The term tag or syntactic category: such as nouns, verbs, prepositions, adverbs, etc.
- The semantic clusters: subjective grouping of the items in the terminology for different purposes.

For further information about this use of knowledge-bases to process text written in natural language use reference [24][51].

For this deliverable, there are new metrics based on the System Knowledge Base:

- **In-System Conceptual Model Nouns (SCM Nouns):** This metric evaluates all words in the requirement that are classified as SCM Nouns. In other words, it checks that each term either belongs to one or more SCM Views or to one or more semantic clusters.

The rationale of this metric is to indicate that the use of too many nouns belonging to the SCM can denote more than one need in a single requirement. Its quality function is:

- $[0, 1)$  → Bad quality
- $[1, 2)$  → Good quality
- $[2, 3)$  → Medium quality
- $[3, + \infty)$  → Bad quality

- **Out-of-System Conceptual Model Nouns (Out-of-SCM Nouns):** This metric evaluates all words in the requirement that are classified as out-of-SCM Nouns. In other words, it checks that each term does not belong to any SCM View and it does not belong to any semantic cluster.

The rationale of this metric is to indicate that the use of any noun not belonging to the SCM must be avoided. Its quality function is:

- $[0, 1)$  → Good quality
- $[1, + \infty)$  → Bad quality

- **In-Semantic Clusters Nouns (SCC Nouns):** This metric evaluates all words in the requirement that are classified as SCC Nouns. In other words, it checks that each term belongs to one or more semantic clusters.

The rationale of this metric is to indicate that the use of too many nouns belonging to the SCC can denote more than one need in a unique requirement. Its quality function is:

- $[0, 1)$  → Bad quality
- $[1, 2)$  → Good quality
- $[2, 3)$  → Medium quality
- $[3, + \infty)$  → Bad quality

- **Out-of-Semantic Clusters Nouns (Out-of-SCC Nouns):** This metric evaluates all words in the requirement that are classified as out-of-SCC Nouns. In other words, it checks that each term does not belong to any semantic cluster.

The rationale of this metric is to indicate that the use of any noun not belonging to the SCC must be avoided. Its quality function is:

- $[0, 1)$  → Good quality
- $[1, + \infty)$  → Bad quality

- **In-Hierarchical Views Nouns (SCV Nouns):** This metric evaluates all words in the requirement that are classified as SCV Nouns. In other words, it checks that each term belongs to one or more SCM views.

The rationale of this metric is to indicate that the use of too many nouns belonging to the SCV can denote more than one need in a unique requirement. Its quality function is:

- $[0, 1)$  → Bad quality
- $[1, 2)$  → Good quality
- $[2, 3)$  → Medium quality
- $[3, +\infty)$  → Bad quality

- **Out-of-Hierarchical Views Nouns (Out-of-SCV Nouns):** This metric evaluates all words in the requirement that are classified as out-of-SCV Nouns. In other words, it checks that each term does not belong to any SCM view.

The rationale of this metric is to indicate that the use any noun not belonging to the SCV must be avoided. Its quality function is:

- $[0, 1)$  → Good quality
- $[1, +\infty)$  → Bad quality

The very same technique of distinguishing which nouns belong to these elements in the ontology has been applied to verbs. The new metrics are:

- **In-System Conceptual Model Verbs (SCM Verbs)**
- **Out-of-System Conceptual Model Verbs (Out-of-SCM Verbs)**
- **In-Semantic Clusters Verbs (SCC Verbs)**
- **Out-of-Semantic Clusters Verbs (Out-of-SCC Verbs)**
- **In-Hierarchical Views Verbs (SCV Verbs)**
- **Out-of-Hierarchical Views Verbs (Out-of-SCV Verbs)**

Their quality functions are the same that the ones used for nouns.

Users can manage how many nouns or verbs the metrics should contain and to assign different quality values for each one.

#### 2.4.4.2 Applying machine learning to improve the quality of requirements

The objective is to emulate the experts' judgment of the quality of new requirements that are entered in the system. In order to achieve this goal, the experts must contribute with an initial set of requirements that they have previously classified according to their quality, and that they have chosen as appropriate for establishing the demanded standard quality (this implies that the initial set must include requirements classified in all quality levels; in other words, including only good requirements is not enough) [26].

For each requirement in the given set, metrics that quantify the various dimensions of quality presented in the works [25], where they are explained in detail, are extracted.

Then Machine Learning techniques (namely Rule Inference) to emulate the implicit expert's quality function are used, i.e. the value ranges for the metrics, as well as the way the metrics are combined to yield the interpretation of requirements quality by the domain expert. The result will be a computable formula made of simple arithmetic and logical operations.

This method has the advantage of being easily customizable to different situations, different domains, different styles to write requirements, and different demands in quality. All we need is a tool that computes quality metrics on textual requirements, and the initial set of requirements previously classified by the expert, in order to feed the learning algorithms. The main contribution of the work, then, is a methodology to build a classifier that learns from the information provided by the expert, and adapts itself to best emulate the expert's judgment. Besides, we can provide automatic suggestions to improve the requirements, by computing the quality rule that could be satisfied with the least effort.

### 2.4.4.3 Metrics for models (\*)

The approach to use the already existing metrics in *System Quality Analyzer (SQA)*, [27] (previously known Requirements Quality Analyzer (RQA)) designed for Requirement Specifications, for models have required work in several dimensions:

- Improving SQA [27] to manage models as well as requirements.
- Mapping the content of a model into a RSHP model [23] [24].
- Improving SQA [27] to avoid assess correctness requirement-specific metrics into model elements.
- Adapting the completeness and consistency assess requirement-specific metrics to assess models from their transformation in RSHP models [23] [24].
  - In Table 3, for each completeness or consistency metrics, the method to map elements from models to RSHP models [23] [24] and the adaptations performed on the metric evaluation are described.

**Table 3.** Mapping to RSHP models

View	Metric	Apply	Comment
Completeness	Terminology coverage	YES	Mapping: <ul style="list-style-type: none"> <li>Each element of the models has been identified with a term in RSHP model.</li> </ul> Analysis: <ul style="list-style-type: none"> <li>Thus, the terminology analysis is focused on the set of terms identified within the original model.</li> </ul>
	Relationships	YES	Mapping: <ul style="list-style-type: none"> <li>Each element of the models has been identified as a term in RSHP model.</li> <li>Each relationship in the model has been transformed in a relationship between those mapped terms in the RSHP model.</li> <li>The relationship type of each relationship in the RSHP model is got from the typology of the relationship in the original model.</li> </ul> Analysis: <ul style="list-style-type: none"> <li>The relationship analysis is focused on the relationships from the RSHP model, which behave equal than the ones got by transforming text in natural language to a RSHP model.</li> </ul>
	Relationship types coverage	YES	Mapping: <ul style="list-style-type: none"> <li>The same mapping as in the “Relationships from SCM View Coverage” metric has been used.</li> </ul> Analysis: <ul style="list-style-type: none"> <li>The analysis is focused on the existence of the selected relationship types with the types of the relationships mapped in the RSHP model.</li> </ul>
	Model-content coverage	YES	Compares the content of a model from the ontology with another model got from the connection source.           Mapping: <ul style="list-style-type: none"> <li>The same mapping as in the “Relationships from SCM View Coverage” metric has been used.</li> </ul> Analysis: <ul style="list-style-type: none"> <li>The analysis is focused on the existence of the same number of instances of relationships between the same terms as,</li> </ul>

			from the ontology-model and from the specification model.
	Properties coverage	YES	<p>Checks that a set of properties selected from the ontology have value in the model got from the connection source.</p> <p>Mapping:</p> <ul style="list-style-type: none"> <li>Each property from the model has been identified as a property in the RSHP model.</li> <li>Properties of the entities such as visualization id, comments, descriptions, etc. as properties in the RSHP model associated with each term mapped.</li> </ul> <p>Analysis:</p> <ul style="list-style-type: none"> <li>The analysis is focused on the existence of the selected properties in the model and their value is not empty.</li> </ul>
	Patterns coverage	NO	<p>This metric is not applicable in this stage of development because, in the ontology definition there is no matching between what is a pattern in the model world and in the RSHP model. This topic will be subject of further discussion and development in future works.</p>
	Link coverage	NO	<p>This metric is not applicable in this stage of development because it has to be investigated where to find the links among the models from the source.</p> <p>Each tool model management tool manages the model and does not provide mechanisms to link models. An external source of links shall be found and used.</p>
Consistency	Property values	YES	<p>Mapping:</p> <ul style="list-style-type: none"> <li>The same mapping as in the “Properties coverage” completeness metric has been used.</li> </ul> <p>Analysis:</p> <ul style="list-style-type: none"> <li>The analysis is focused on the existence of the selected properties in the model and checking that their values do not imply contradictions.</li> </ul>
	Properties allocation	YES	<p>Technically, it can be applied but we don’t expect companies to use it because the source information it needs to perform its assessment won’t be included in the models.</p> <p>Mapping:</p> <ul style="list-style-type: none"> <li>Based on the term identification in “terminology” completeness metrics.</li> <li>Based on the mapping from “Model-content coverage”.</li> </ul> <p>Analysis:</p> <ul style="list-style-type: none"> <li>It’s focused on finding the suitable value for a set property for each element of the ontology selected in the configuration.</li> <li>Then with all the values available checks that the operation holds the property in each higher level of abstraction of the SCM view of the ontology.</li> </ul>
	Overlapping requirements	YES	<p>It shall be renamed for models as overlapping models.</p> <p>Mapping:</p> <ul style="list-style-type: none"> <li>Based on the term identification in “terminology” completeness metrics.</li> <li>Based on the mapping from “Model-content coverage”.</li> </ul> <p>Analysis:</p> <ul style="list-style-type: none"> <li>Checks the similarity of the models by checking the similarity</li> </ul>



			of the RSHP models got from them.
	Measurement units	NO	It is not applicable because a relevant mapping from the models to the terms shall be found to be implemented.
	Measurement units for specific property	YES	<p>Technically, as in the “Arithmetic operation compliance with SCM”, it can be applied but we do not expect companies to use it because the source information it needs to perform its assessment will not be included in the models.</p> <p>Mapping:</p> <ul style="list-style-type: none"> <li>Based on the term identification in “terminology” completeness metrics.</li> <li>Based on the mapping from “Model-content coverage”.</li> </ul> <p>Analysis:</p> <ul style="list-style-type: none"> <li>Checks that a property allocated to an element of the selected SCM view is always expressed in the same measurement unit.</li> </ul>

- Creating set of correctness metrics to assess models.
  - In Table 4, the new correctness metrics to assess quality models are described.

**Table 4.** Correctness metrics for models

Type	Metric	Comment
Class model	Weighted methods per class (WMC)	<p>Description:</p> <ul style="list-style-type: none"> <li>It is the summation of the complexity of all the methods in the class. A simpler case for WMC is when the complexity of each method is evaluated to unity. In that case, WMC is considered as the number of methods in the class.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>
	Depth of inheritance tree (DIT)	<p>Description:</p> <ul style="list-style-type: none"> <li>This metric represents the length of the inheritance tree from a class to its root class.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Incremental complexity function.</li> </ul>
	Number of Children (NOC)	<p>Description:</p> <ul style="list-style-type: none"> <li>This metric represents the number of immediate subclasses subordinated to a class in the class hierarchy. NOC relates to the notion of scope of properties. It is a measure of how many subclasses are going to inherit the methods of the parent class.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale</li> </ul>
	Coupling between object classes (CBO)	<p>Description:</p> <ul style="list-style-type: none"> <li>This metric measures the level of coupling among classes. CBO relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables of another. Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>



Response for a Class (UFC)	<p>Description:</p> <ul style="list-style-type: none"> <li>The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. The cardinality of this set is a measure of the attributes of objects in the class.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Incremental complexity function.</li> </ul>
Method hiding factor (MHF)	<p>Description:</p> <ul style="list-style-type: none"> <li>This metric is a measure of the encapsulation in the class. It is the ratio of the sum of hidden methods (private and protected) to the total number of methods defined in each class (public, private, and protected).</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale</li> </ul>
Attribute Hiding Factor (AHF)	<p>Description:</p> <ul style="list-style-type: none"> <li>This metric represents the average of the invisibility of attributes in the class diagram. It is the ratio of the sum of hidden attributes (private and protected) for all the classes to the sum of all defined attributes (public, private, and protected).</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Decremental complexity function.</li> </ul>
Public methods (PM)	<p>Description:</p> <ul style="list-style-type: none"> <li>This metric calculates the public methods in a class.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Incremental complexity function.</li> </ul>
Number of methods (NM)	<p>Description:</p> <ul style="list-style-type: none"> <li>This metric counts all methods (public, protected, and private) in a class.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Incremental complexity function.</li> </ul>
Design Size in Classes (DSC)	<p>Description:</p> <ul style="list-style-type: none"> <li>This metric is a count of the total number of classes in the design.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Incremental complexity function.</li> </ul>
Data Access Metric (DAM)	<p>Description:</p> <ul style="list-style-type: none"> <li>This metric is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Decremental complexity function.</li> </ul>
Direct Class Coupling (DCC)	<p>Description:</p> <ul style="list-style-type: none"> <li>This metric is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>
Class Interface Size (CIS)	<p>Description:</p> <ul style="list-style-type: none"> <li>This metric is a count of the number of public methods in a class.</li> </ul>

		<p>Evaluation:</p> <ul style="list-style-type: none"> <li>Incremental complexity function.</li> </ul>
	Number of Methods (NOM)	<p>Description:</p> <ul style="list-style-type: none"> <li>This metric is a count of all the methods defined in a class.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>
	Class Category Relational Cohesion (CCRC)	<p>Description:</p> <ul style="list-style-type: none"> <li>The CCRC metric measures how cohesive the classes are in a class diagram design. Relational cohesion is the number of relationships among classes divided by the total number of classes in the diagram.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>
Package model	Abstractness (ABST)	<p>Description:</p> <ul style="list-style-type: none"> <li>The abstractness metric measures the package abstraction rate. A package abstraction level depends on its stability level. Calculations are performed on classes defined directly in the package and those defined in sub-packages. In UML models, this metric is calculated on all the model classes.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Decremental complexity function.</li> </ul>
	Instability (I)	<p>Description:</p> <ul style="list-style-type: none"> <li>The instability metric measures the level of instability in a package. A package is unstable if its level of dependency is higher than that of those depending. The instability of a package is the ratio of its afferent coupling to the sum of its efferent and afferent coupling.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Incremental complexity function.</li> </ul>
	Distance from Main Sequence (DMS)	<p>Description:</p> <ul style="list-style-type: none"> <li>The DMS metric measures the balance between the abstraction and instability of a package.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Decremental complexity function.</li> </ul>
Sequence Diagram	NonAnonymObjRatio (SDm1)	<p>Description:</p> <ul style="list-style-type: none"> <li>Measures the ratio of objects with name to the total number of objects in a sequence diagram.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>
	NonDummyObjRatio (SDm2)	<p>Description:</p> <ul style="list-style-type: none"> <li>Measures the ratio of non-dummy objects (objects that correspond to classes) to the total number of objects in a sequence diagram.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>
	MsgWithLabelRatio (SDm3)	<p>Description:</p> <ul style="list-style-type: none"> <li>Measures the ratio of messages with label (any text attached to the messages) to the total number of messages in a sequence diagram.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>

	NonDummyMsgRatio (SDm4)	<p>Description:</p> <ul style="list-style-type: none"> <li>Measures the ratio of non-dummy messages (messages that correspond to class methods) to the total number of messages in a sequence diagram.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>
	ReturnMsgWithLabelRatio (SDm5)	<p>Description:</p> <ul style="list-style-type: none"> <li>Measures the ratio of return messages with label (any text attached to the return messages) to the total number of return messages in a sequence diagram.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>
	MsgWithGuardRatio (SDm6)	<p>Description:</p> <ul style="list-style-type: none"> <li>Measures the ratio of guarded messages (messages with conditional checking) to the total number of messages in a sequence diagram.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>
	MsgWithParamRatio (SDm7)	<p>Description:</p> <ul style="list-style-type: none"> <li>Measures the ratio of messages with parameters to the total number of messages in a sequence diagram.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>
	Sequence Diagram (SeqLoD) score (LoDsd)	<p>Description:</p> <ul style="list-style-type: none"> <li>The sequence diagram metrics cover two aspects of detailedness, namely object detailedness and message detailedness, the first two metrics belong to the former and the rest belongs to the latter.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>
Uses case	Number of actor action steps of the use case (NOAS)	<p>Description:</p> <ul style="list-style-type: none"> <li>Number of actor action steps of the use case.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>
	Number of system action steps of the use case (NOSS)	<p>Description:</p> <ul style="list-style-type: none"> <li>Number of system action steps of the use case.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>
	Number of use case action steps of the use case (NOUS)	<p>Description:</p> <ul style="list-style-type: none"> <li>Number of use case action steps of the use case (inclusions or extensions).</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Incremental complexity function</li> </ul>
	Number of steps of the use case (NOS)	<p>Description:</p> <ul style="list-style-type: none"> <li>Number of steps of the use case.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>
	Number of exceptions of the use case (NOAS_RATE)	<p>Description:</p> <ul style="list-style-type: none"> <li>Rate of actor action steps of the use case.</li> </ul> <p>Evaluation:</p> <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>

	Rate of system action steps of the use case (NOSS_RATE)	Description: <ul style="list-style-type: none"> <li>Rate of system action steps of the use case.</li> </ul> Evaluation: <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>
	Rate of use case action steps of the use case (NOUS_RATE)	Description: <ul style="list-style-type: none"> <li>Rate of use case action steps of the use case.</li> </ul> Evaluation: <ul style="list-style-type: none"> <li>Function defined by intervals scale.</li> </ul>

#### 2.4.4.4 Metric checklists (\*)

For this deliverable, there are metrics based on completeness of checklists. A checklist is a test with a series of questions that the user must answer. Depend on the answers, it is possible to weigh the result to provide a quality measure using quality ranges defined.

System Quality Analyzer (SQA) [27], previously known Requirements Quality Analyzer (RQA), will provide a framework to create the checklists, provide the platform to answer the questions and send the results. The tool will import the results and give a quality measure based on quality ranges.

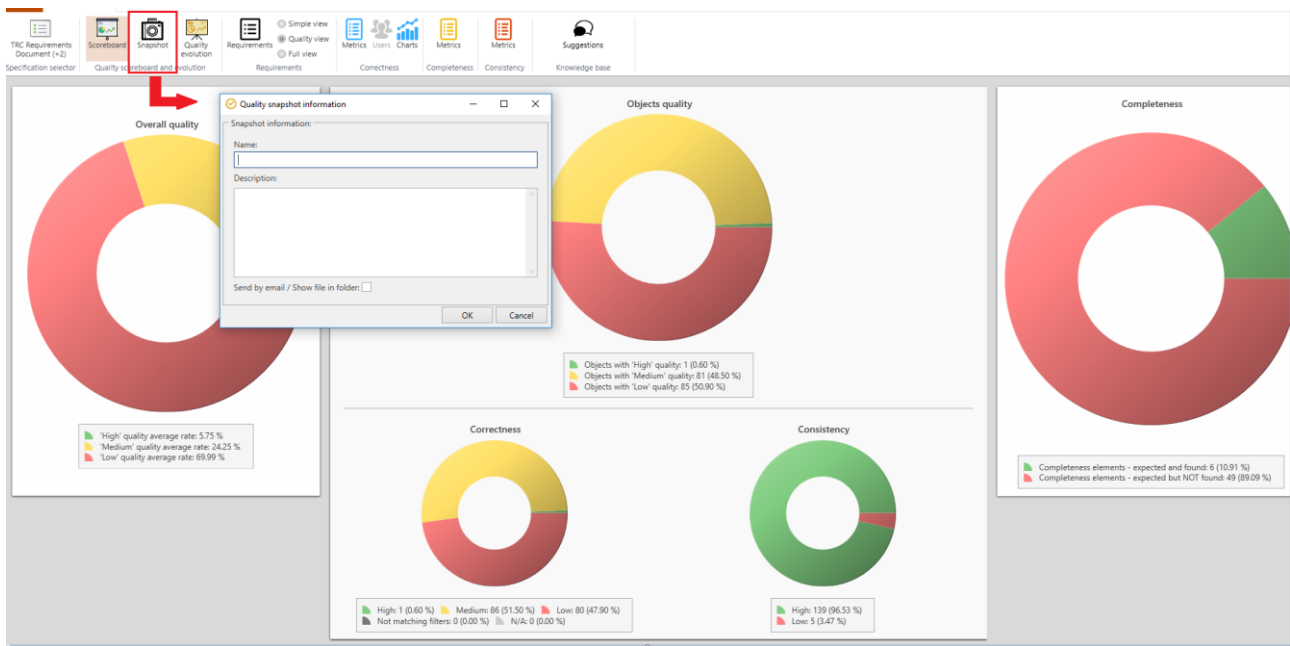
#### 2.4.4.5 Quality evolution

One of the fundamental tasks in the quality management is to know its evolution over the time. For this reason, a quality evolution manager has been developed. The objective is to facilitate access to information related to the quality of the specification that has been carried out over time (Figure 26).



**Figure 26.** Example of quality evolution wrt time for a requirements specification

With this manager, the user can save a snapshot of the current quality of the specification in order to give the possibility to review it later (Figure 27). It is composed of every different type of quality reports available in the application.

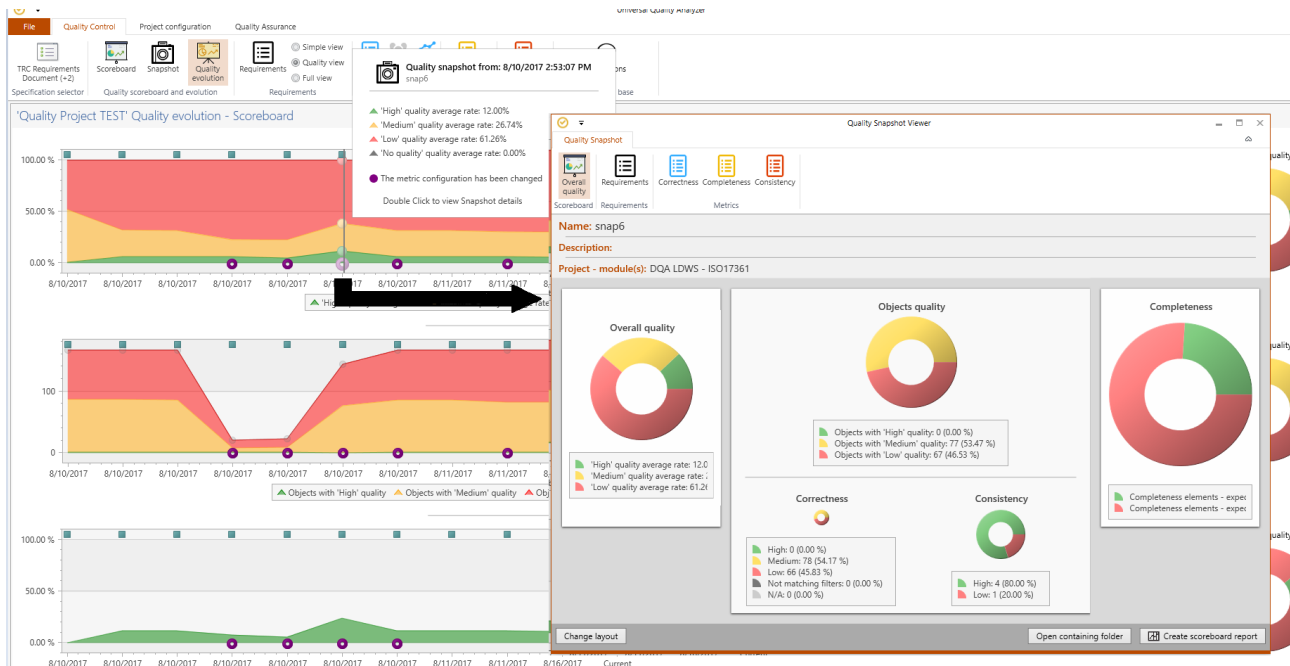


**Figure 27.** Saving snapshot with the quality of the project

Each snapshot is composed by:

- Each requirement assessment belonging to correctness, completeness and consistency point of views.
- Every involved metric configuration and their quality functions.

For each snapshot is possible to show the quality information of the project saved at some point, Figure 28. The snapshot contains all the information of the quality of the metrics and requirements that were included when the snapshot was saved.



**Figure 28.** Information of the snapshot

#### 2.4.4.6 Employment in AMASS

The Architecture-Driven Assurance of AMASS will have a Quality Assessment interface so that the AMASS tool platform interacts with Quality Management Tools. These tools will provide AMASS with

measurements of quality metrics for different work products. Such information could be used, for example, as:

- Evidence Artefact evaluations
- V&V evidence artefacts
- Information associated to the elements of System Models

Finally, the input to the Quality Management Tools could correspond to data from the AMASS Tool platform (e.g. a CHESS model) or any other external tool (e.g. a Rhapsody model).

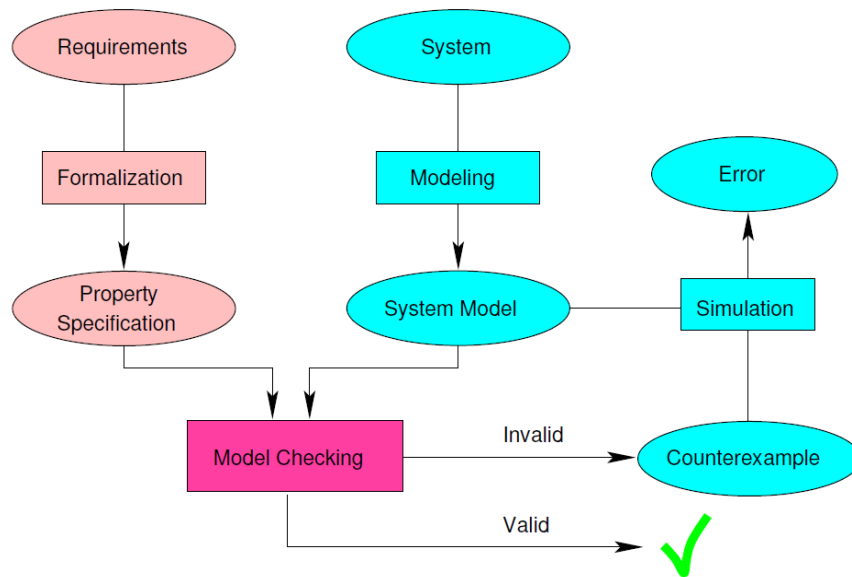
## **2.4.5 Verifying Requirements against System Design**

Once the requirements are formalized and free of any defects, they can be automatically verified against the corresponding system design by using model checking technique. The model checking verification consumes a model of the system under verification and a specification the model should meet. For those two inputs it performs an algorithmic decision about the validity of the system with respect to the specification. If the system meets the specification, the model checking procedure simply returns a message informing the user about the fact. In the other case a counterexample is provided, i.e. behaviour of the system witnessing the violation of the specification. The basic schema of the model checking process is shown in Figure 29.

### **2.4.5.1 Automated Formal Verification**

There are several model checking tools, e.g. DIVINE, NuSMV or nuXmv that can be used to verify the requirements against corresponding system design. The choice of a suitable model checker depends on several factors. Mainly, we are limited by the language that is used to model the system and also by the language in which we specify the requirements, because different tools support different languages. Linear Temporal Logic (LTL) is one of the most widely supported languages for requirements specification. As for the model of the system, Simulink, C and C++ are representatives of commonly used languages to specify the behaviour of the system or components.

Besides the modelling languages, the right choice of a model checking tool depends on many other factors and different tools are suitable in different situations. A verification expert should decide which model checking tool is the most convenient in a particular situation. Therefore, in AMASS we propose to define a universal interface that calls a model checking tool (chosen by the user), in order to support formal verification of requirements against corresponding system design.



**Figure 29.** Model Checking Schema

#### 2.4.5.2 Toolchain for Automated Formal Verification

Development of embedded systems in the avionics domain is strictly guided by certification standards, such as RTCA DO-178. These standards predefine a comprehensive set of processes and assurance artefacts that need to be provided before the product is ready to flight. The standard, however, does not provide guidance as to how these artefacts should be obtained, what tools should be used during each development phase or how best to check that the artefacts are correct. There are tools for individual stages, for writing and tracing requirements, for modelling the system architecture, for simulating performance properties, etc. but they often come with their own specification languages, mostly incompatible with the tools supporting the next development stage.

The AMASS approach to systems assurance is compliant with the development process allowed by avionics standards. An architecture-driven (or model-based) development is preferred since a model of the final system is more amenable to the verification of requirements than the source code. Therefore, the AMASS tools will be readily applicable to the development of avionics systems and will remedy the lack of supporting tools available to avionics engineers. Most importantly, one of the AMASS products will be a platform unifying the tools to a single toolchain, thus eliminating the need to perform each development stage in separation using mutually incompatible tools.

At the centre of this toolchain will be the AMASS platform linking together unmodified tools provided by various partners. The common communication language will be provided by the OSLC that will wrap around each input-output artefact for each tool. This approach has two advantages: first, the partners do not need to modify their tools, only the OSLC for artefacts is required; second, new tools can be easily incorporated since both the platform and the OSLC layer will be open. Having the platform as a central point will allow to maintain the state-of-the-art standards, assuming the new version of incorporated tools will remain compliant with the OSLC layer.

The prerequisite for these tools to be used by the platform will be publicly available, for example by a remote call via HTTP. Since not every tool vendor provides this functionality, we further intend to establish an automation server that will allow access to even these tools. The automation server will use the OSLC automation standard to communicate the input and output of selected tools which will be executed locally on the automation server.



### 2.4.5.3 Employment in AMASS

The AMASS Architecture-Driven Assurance will include a Requirements Verification component that will interact with external tools to prove properties on the system design and/or implementation. The verification report and possibly the related proofs will be used as evidence in the assurance case.

The platform will maintain the assurance case specification and provide evidence and compliance management so that the system developers could rely on a single toolchain throughout the whole process, from stakeholder requirements elicitation to product certification. In particular, the platform prototype delivered within AMASS will allow to capture and formalize requirements in Papyrus, modelling the system architecture in CHESS (using UML/SysML), and then verifying the requirement contracts in OCRA. Another possible use case will be to formalize requirements – thus obtaining their LTL version – and then checking their sanity (consistency, non-redundancy, and completeness) by looney or Acacia+, after which they can be checked against the system architecture by a model checker (NuSMV, DIVINE, etc.); as specified by a particular assurance case. Furthermore, the evidence – formal requirements, proof of compliance, or a violating counter-example – gather during each application of the platform will be stored and maintained, later to be used for certification of the product (or reused in another development).

## 2.4.6 Design Space Exploration (\*)

In order to argue about the assurance of a system it is sometimes necessary to support the design choices with evidence to say that one choice was better than another. The same support can be used at design time for design space exploration. The basic concepts of design space exploration are parameters (and thus a parameterized architecture) and configurations, which can be defined as assignments to the parameters.

We focus here on parameters that define the structure of the architecture as described in section 2.2.2. They define for example the number of components, ports, connections or if a subcomponent is enabled/disabled. Therefore, a parameterized architecture represents in a concise way a potentially infinite number of architectures, one for each configuration. In a parallel project, called CITADEL<sup>6</sup>, these notions have been formally defined on top of AADL and its variant SLIM supported by the COMPASS<sup>7</sup> toolset.

Once we have defined a set of configurations (with a parameterized architecture or with a set of architectural models), we can compare them with respect to a number of properties based on formal verification and analysis, as done in [32]. In particular, we can formalize a set of soft requirements into qualitative formal properties (e.g. in LTL) and compare the different configurations based on which soft properties are satisfied and which are not. Moreover, we can compare the results of safety analysis (for example in terms of the number of cut sets related to the same top-level event) in the different configurations.

### 2.4.6.1 Employment in AMASS

OCRA will be extended to analyse a set of configurations. The interaction between CHESS and OCRA will be extended to compare the different configurations based on verification, validation, and safety properties.

Moreover, tools for specification and processing of design space exploration (different architectural configurations) based on dependability parameters (e.g., criticality level, etc.) will be investigated.

---

<sup>6</sup> <http://www.citadel-project.org/>

<sup>7</sup> <http://www.compass-toolset.org/>

### 2.4.7 Simulation-Based Fault Injection Framework (\*)

As Ziade surveyed in [28] and Benso tackled in [29] fault injection has been deeply investigated by both industry and academia as a dependability validation technique. However, the use of simulation-based fault injection during early design phases has not been the object of attention so far.

Among the different fault injection techniques, simulation-based fault injection contains remarkable benefits. For instance, it allows high observability and controllability of the experiments without corrupting the original design. Through those models, which can be implemented at different levels (e.g. system, hardware), the dependability can be already evaluated during early design phases. In other words, the system is simulated on the basis of simplified assumptions to:

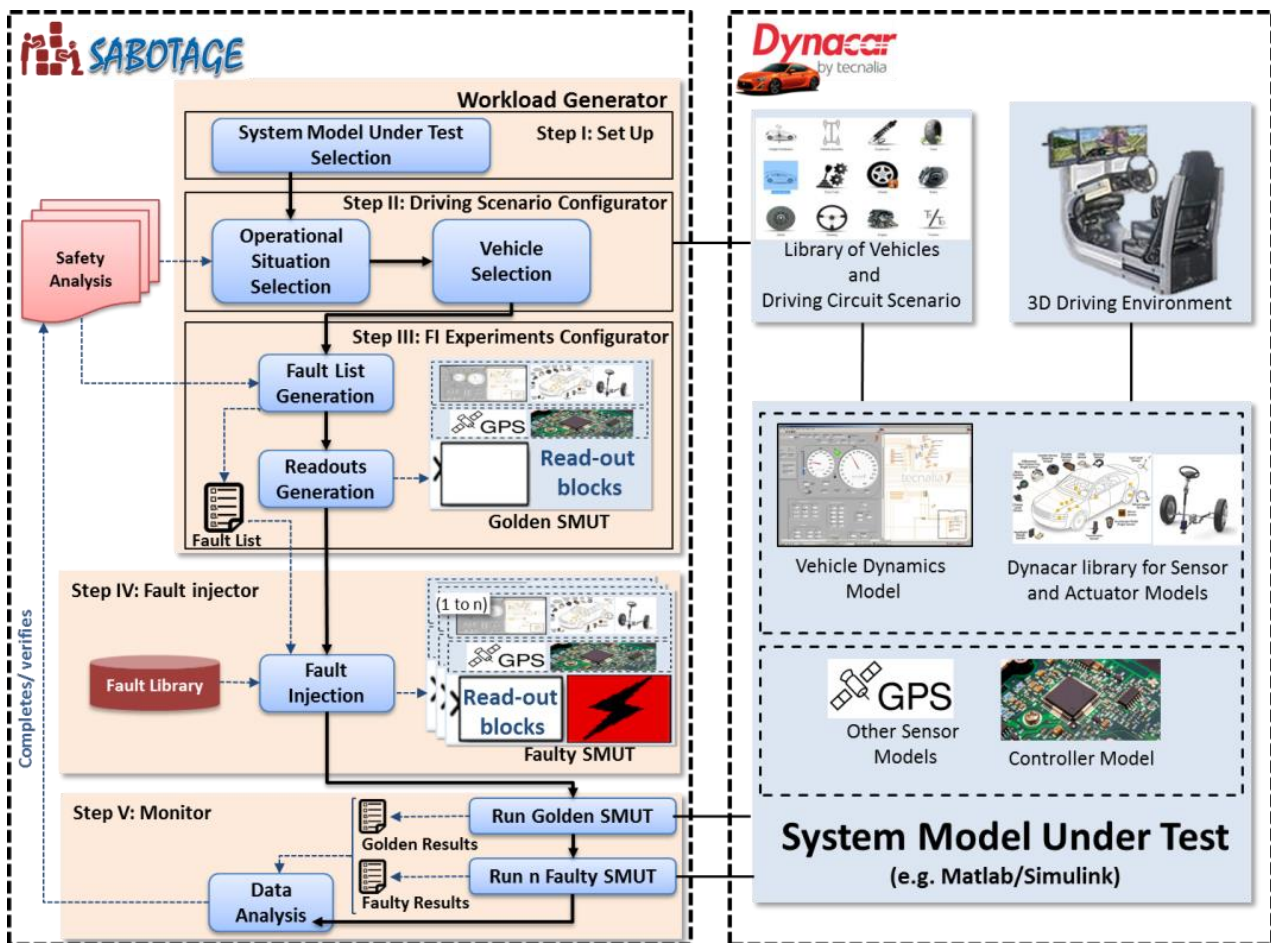
- 1) Forecast or predict its behaviour in the presence of faults.
- 2) Estimate the coverage and latencies of fault tolerant mechanisms.
- 3) Explore the effects of different workloads (different activation profiles).

Model-based design combined with a simulation-based fault injection technique and a virtual vehicle poses as a promising solution for an early safety assessment of automotive systems. Initially, a fault forecasting process takes place. The design with no safety consideration is stimulated with a set of fault injection simulations. The main goal of this first set of fault injection simulations is to evaluate different safety strategies during early development phases estimating the relationship of an individual failure to the degree of misbehaviour on vehicle level. Once the most appropriate safety mechanisms are included, a second set of fault injection experiments is performed in order to early validate the safety concept. All this avoids late redesigns, leading to significant cost and time savings.

In this section, Sabotage prototype tool framework that is provided by Tecnia will be introduced. Sabotage is a simulation-based fault injection tool framework based on the well-known FARM model [30]. This model stands for (a) **Fault (F)**: identifying the set of faults to be injected, (b) **Activation (A)**: activating the set of faults exercised during the experiments, (c) **Readouts (R)**: setting the readouts which constitute the observers of the experiments, (d) **Measures (M)**: compute the obtained measures to evaluate dependability properties. Figure 30 illustrates the main building blocks of the Sabotage tool framework to perform an early safety assessment of automotive systems. Especially, this approach allows setting up, configuring, running and analysing fault injection experiments.

In order to evaluate the effect considering the Dynamic of the system, an environment simulator is integrated in the fault injection tool. For the automotive domain, Dynacar [31] offers a library of different vehicles and driving circuit scenarios together with a 3D driving environment. The dynamics of the selected vehicle is modelled as part of an S-function in the Matlab/Simulink environment. An S-function is a computer language description of a Simulink block written in Matlab, C, C++, or FORTRAN. Moreover, Dynacar already provides the designer with a set of sensor and actuator models contained in a library. The whole system model under test (SMUT) is completed by including simulation models representing the controller. All these models are developed, for instance, using Matlab/Simulink.

The Dynamics of the system will be not applicable in the AMASS project.



**Figure 30.** Sabotage Framework for Simulation-Based Fault Injection

The Sabotage framework is the responsible to automatically inject faults into the SMUT and compute the results. First, the Workload Generator creates the functional inputs to be applied to the SMUT. More specifically, it is the responsible of the following subtasks:

- selecting the system model under test;
- choosing the operational scenario from an and environment scenario library;
- configuring the fault injection experiments. This includes creating the fault list and deciding the read-out or observation points (signal monitors).

Once the Workload is conceived, the Fault Injector block utilises the fault list and a fault model library to create the saboteurs (Matlab/Simulink S- functions). For now, the Fault Injector script is implemented as a Matlab code and the library of fault models as C code templates. However, AMASS will investigate on further improving the proposed approach in order to provide non-language dependent solutions. The saboteurs or fault injection blocks are included as part of the faulty SMUT. After saboteurs are injected, the faulty SMUT is completed and ready to be simulated. At this point, the designer can generate as many faulty SMUTs as needed. The simulation process starts from a run of a fault free version of the SMUT (Golden). The simulation environment is invoked through the Monitor (the Oracle implemented in Java). The Monitor is the responsible of running the fault injection experiments under the pre-configured vehicle scenario. After running the Golden simulation, the same applies to the Faulty SMUT. To finish with, it compares and analyses the collected data by comparing golden and faulty results.

❖ **Workload Generator:** This block selects the SMUT, chooses the most appropriate environment scenario, which represents the operational situation, and configures fault injection experiments. The basis for specifying the operational situations are driven by safety analysis. These operational situations

include the specification of environment conditions and so on. The following list defined for an automotive domain, provides a more detailed list of which kind of variables need a configuration:

- Location: highway, urban
- Road conditions: uphill, on a curve
- Environment conditions: good conditions, heavy rain
- Traffic situations: fluent
- Vehicle speed (Km/h)
- Manoeuvres: parking, overtaking, lane keeping
- People at risk: driver, passenger, pedestrians.

Then, the designer selects the environment scenario that best symbolises those operational situations to be simulated. Together with that selection, the system (e.g. vehicle or robot) to be simulated is chosen. For example, to emulate that vehicle and the driving circuit scenarios, Dynacar manages a scenario catalogue that includes up to 150 configurable parameters. After performing this step, the *fault injection experiments configurator* block in Sabotage gives the designer the possibility of creating the fault list and selecting where to monitor fault injection experiments by including signal monitors or readout blocks. The main strategy is to identify a representative and optimal failure type subset, e.g. omission, commission, timing or value, to reproduce target system malfunctions.

- ❖ **Fault Injector:** The fault list is used to produce a Faulty SMUT only in terms of reproducible and prearranged fault models by including saboteur blocks (S-functions). Fault models are characterised by a type (e.g. frozen, stuckat0, delay, invert, oscillation or random), target location, injection triggering (e.g. scenario position or time driven), and duration. In order to create a Faulty SMUT, the Fault Injector injects an additional saboteur model block per fault entry from the Fault List. Moreover, the injected block is fulfilled with information coming from a fault model template library. Saboteurs are extra components added as part of the model-based design for the sole purpose of FI experiments. Algorithm 1 depicts a generic fault model such as stuck-at last value can lead to an omission failure mode.

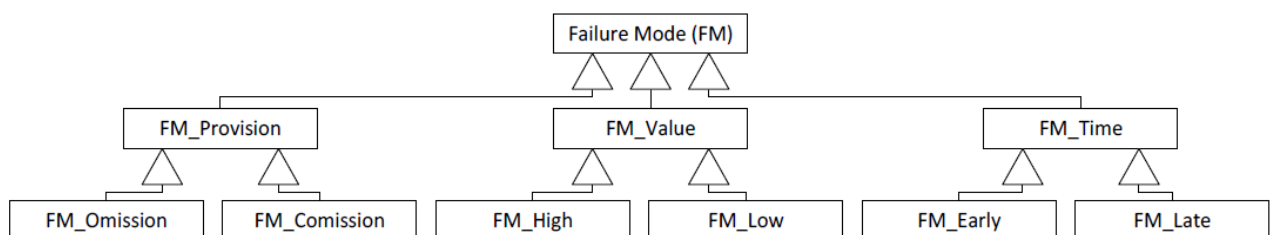
```

Require:  input, pos, simutime, faultdur;
1      If pos== triggerpos then
2          Freeze=input;
3          enable=1;
4      While enable==1 && simutime<=faultdur do return freeze;
5      return input;

```

**Algorithm 1.** Stuck-at last value

It is very important to stabilize a semantic of failure modes. Some European projects such a MOGENTES [41] evolved different types of failures which affect to the system. D. Domis [42] and B. Kaiser [43] have identified a semantic of failure modes based on the concept of HAZOP guidewords adapted for software, this method is called SHARD but it has been adapted to the system domain. Figure 31 is an example of a Failure Type System.



**Figure 31.** Failure Type System.

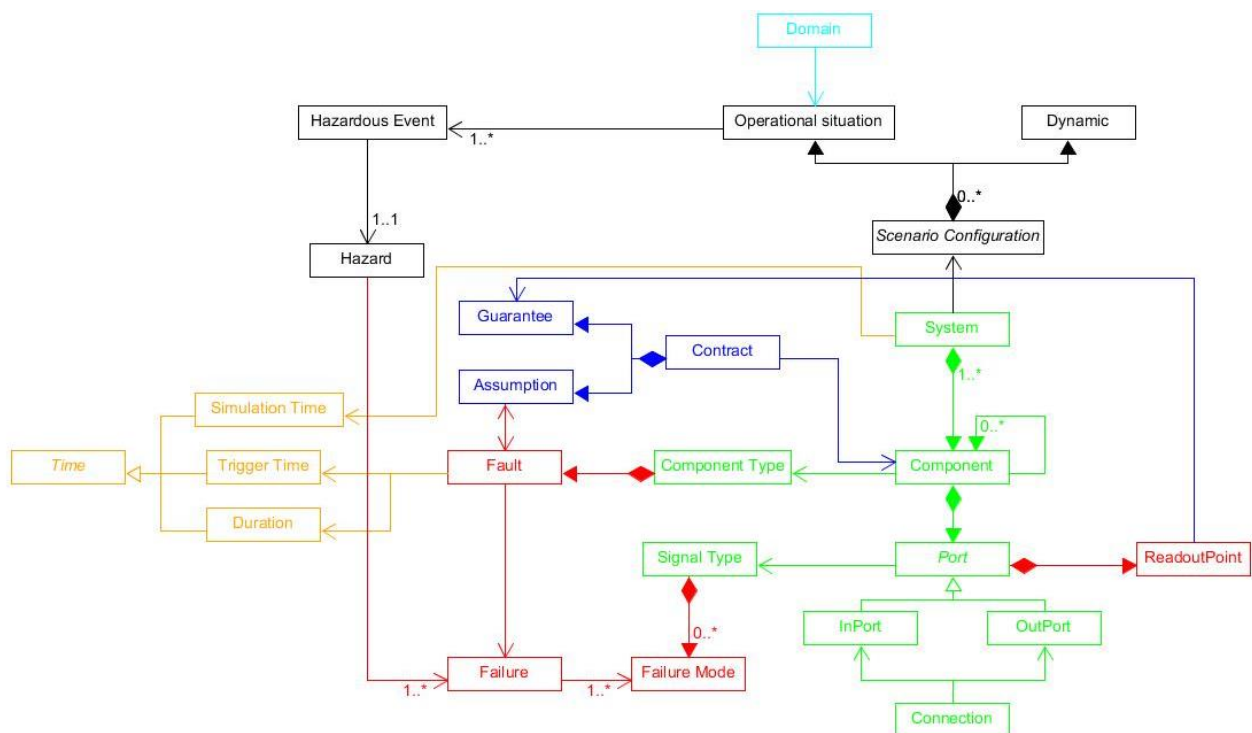
❖ **Monitor:** After performing the configuration of the fault injection scenarios and creating the required amount of Faulty SMUT, the Monitor invokes the simulator. It tracks the execution flow of the Golden and Faulty simulations. The Monitor compares Golden and Faulty SMUT results by the *data analysis* activity. The pass/fail criterion of the tests, which was established by the designer as part of Step III (cf. Figure 30, is used to compute and finalise the results. This criterion includes different properties like the maximum acceptable distance from optimal path considering the vehicle behaviour is acceptable in terms of vehicle dynamics. In other words, acceptable maximum system reaction times are obtained.

In brief, this approach aims at finding acceptable safety properties for model-based design of automotive systems. For instance, failure effects and system maximum tolerable response times are obtained. Through these remarkable outcomes, safety concepts and mechanisms can be more accurately dimensioned.

#### 2.4.7.1 Sabotage architecture

The Sabotage architecture is defined by the connection of Massif and Sabotage metamodels. Massif [44] is a feature to support the transformation from a system defined on Simulink behavioural model to store its information at Eclipse Modelling Framework and vice versa.

Figure 32 illustrates the Sabotage and Massif concept metamodel joint. Massif (see Appendix E: Massif Metamodel) provides the information of system and, on the other hand, Sabotage metamodel holds all the necessary information about the Scenario Configurator (optional in AMASS) and the Fault Injection Experiments configurator. On the first point, it is not necessary in order to carry out the fault injection experiments. However, the main benefit of this is that effect on system (e.g. vehicle or robot) dynamics level could be evaluated. This includes establishing the operational situation and the vehicle. On the other hand, the fault injection experiment configurator (fault list and readouts generation) collects the information such as failure mode or the injection timings. Compared to other approaches, determining the timing is essential due to the simulation nature. As specified in Figure 30 a semantics formalisation for failure modes is established. This list consists of: Omission, Commission, True when False, False when True, Too Low, Too High, etc.



**Figure 32. Sabotage Metamodel.**

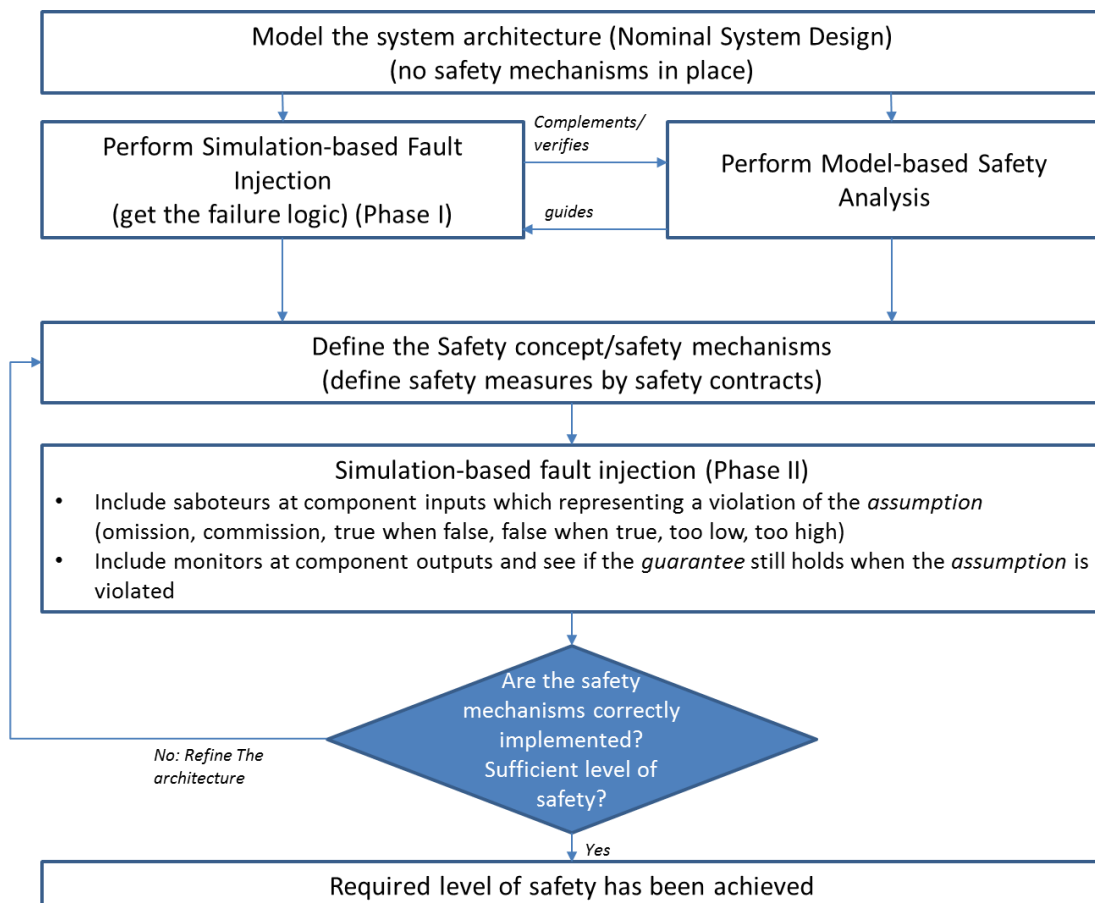


Based on certain information such as failure modes and component type, the Simulink saboteurs can be generated. Since a direct transformation between Papyrus/CHESS and Simulink is out of scope, an intermediary solution will be analysed.

#### 2.4.7.2 Employment in AMASS

In AMASS we plan to span the spectrum from relaxing the fault simulation constraints to instrumenting the automated assessment work. Sabotage framework will be developed and it will be examined the possibility of the integration with other AMASS approaches listed below:

- **Integrate Sabotage framework with contract-based approach (to automate contract-based safety engineering process):** combining the fault injection coming from the Sabotage framework, together with the contract-based approach and the insertion of monitors, is one of the goals in AMASS. This approach supports the automation of the safety concept creation process and an early validation of a safety concept. Besides, it allows dimensioning the safety concept and achieving its early safety validation.



**Figure 33.** Integration workflow: from contract-based design to the generation of saboteurs and monitors.

- **Connect to other system modelling environments such as Papyrus/CHESS/SysML:** Different information contained in the fault list such as the fault models could be derived from the faulty behaviour of the system component model. Furthermore, different approaches will be investigated and evaluated in order to establish a possible connection between Papyrus/CHESS and Matlab/Simulink. This would allow generating the saboteurs (Matlab/Simulink S-functions) from the information specified in the system model (Papyrus/CHESS/SysML).
- **Linking to model-based safety analysis tools:** Due to the fact that modelling the fault as representative as possible is a crucial factor, the fault models are mainly based on techniques such as Failure Modes

and Effects Analysis or Fault Tree Analysis. The proposed solution will use that information as starting point and complete it. For example, it will help constructing the failure logic of model-based fault trees or providing feedback to safety analysis techniques for non-known failure effects and verification of the safety analysis. Hence, this allows proving all the concerns are adequately addressed in the body of the safety analysis.

- **Compare Fault Injection simulation results with the results of performing fault injection in a real system (e.g. vehicle or robot):** A relevant factor is how accurate the simulation results are compared to the faulty behaviour of the actual system. With the aim of increasing the level of trust on the simulation approach, a set of fault injection experiments can be performed in a real system.

Once the safety mechanisms are included in the SysML/UML design, the nominal model is extended one more time, with the same fault effects added in the last extended system, but this time, including the contract-based monitoring technique which checks if the contracts are violated or not. If not, the architecture would need to be redefined and checked it again.

## 2.4.8 Model-Based Safety Analysis

Fault injection extends a nominal behavioural model with faulty behaviours. This extended model can be used to conduct Model-Based Safety Analysis on the system interacting with external tools such as xSAP. More precisely, xSAP can generate fault trees based on this extended model. Such fault tree is a set of Minimal Cut Sets (MCS), which are the minimal configurations of faults leading to a Top Level Event (TLE). The TLEs of the fault trees can be the violations of the LTL properties. A probability can be attached to each failure mode by the user, which will allow xSAP to compute the probability for the TLE to happen.

### 2.4.8.1 Employment in AMASS (\*)

In AMASS, xSAP will be used to automatically derive fault-trees from the system extended model specification, i.e. the system nominal behavioural model extended with faulty behaviour. The obtained fault tree will be used to derive safety requirements and will be stored as an artefact to support the assurance case.

An initial integration about CHES and xSAP was performed in the SafeCer project, to allow the automatic invocation of xSAP from CHES by using the information about the system nominal and faulty behaviour provided in the CHES model. Thus, in AMASS we will improve in term of functionalities what was done in SafeCer.

## 2.5 Assurance Patterns for Contract-Based Design (\*)

### 2.5.1 Assurance of Architectural Patterns

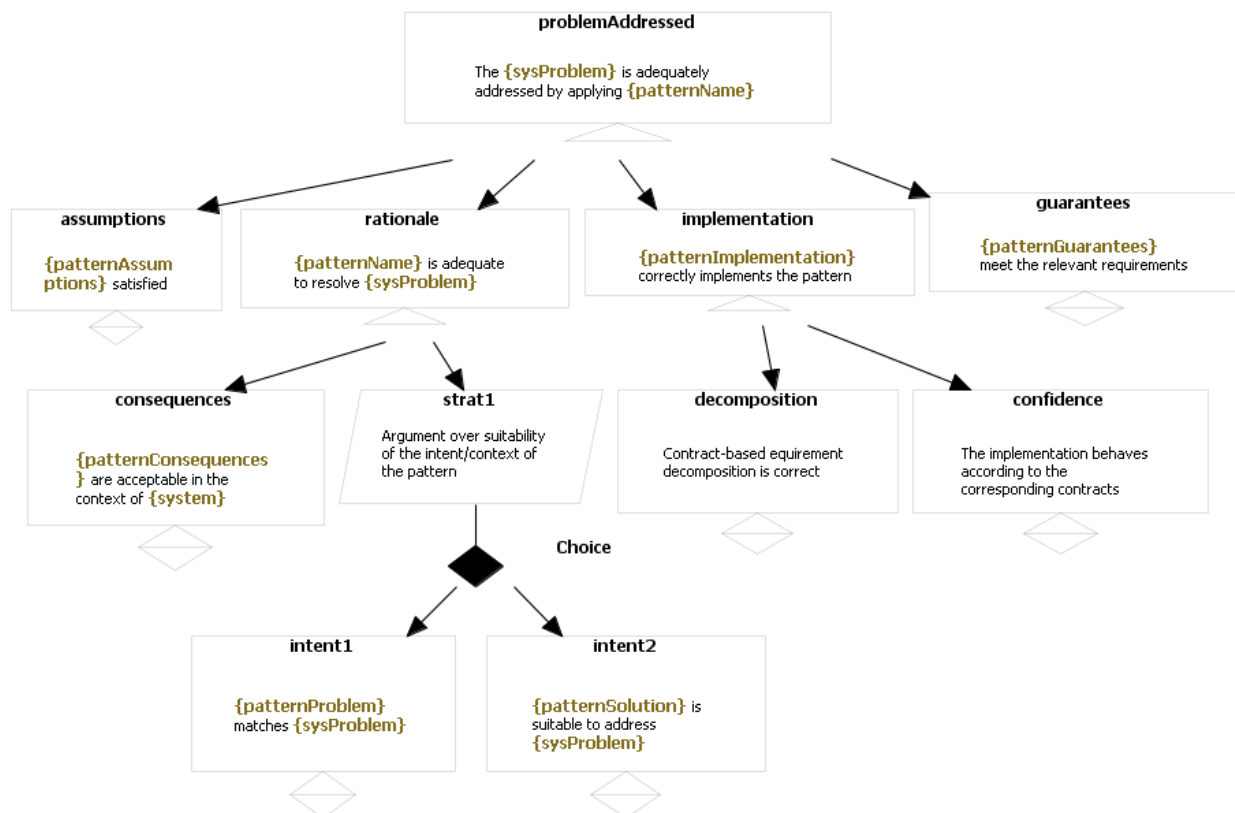
As mentioned earlier, capturing information about patterns in assumption-guarantee contracts style through the design pattern template can serve as the basis for assuring that the application of the architectural pattern adequately addresses the problem that is trying to solve. Using contracts for architectural patterns [40] offers a way of capturing under which conditions instantiating a pattern offers the desired specification. The guarantee of such a contract (referred to as PatternGuarantee) represents the instantiated pattern behaviour, while the assumptions (referred to as PatternAssumptions) represent the requirements that need to be met by the environment for the pattern to be considered correctly instantiated, i.e., for the guarantees to be offered.

Each architectural pattern offers a solution to a particular problem. The list of the known problems the patterns is addressing is captured in the design pattern template. After choosing to use a particular architectural pattern for a specific problem, we need to assure that the pattern is suitable to address this problem as well as that the pattern has been correctly applied, according to the information from the design pattern template. For assuring the application of an architectural pattern, we first need to assure



that the specific pattern is suitable to address the problem at hand. We can assure that, either by looking at the problem statement in the design pattern template and whether our problem matches any of the known problems the pattern is used for, or if our problem is not in that list, then we need to assure why this pattern is suitable to address that particular problem. Furthermore, the consequences of using the pattern should be acceptable in the context of the system. Once the pattern is deemed suitable, we then assure that the PatternAssumptions are met, and that the PatternGuarantees satisfy the relevant requirements. For example, introduction of the acceptance voting pattern influences timing behaviour of the system, hence we should assure that the PatternGuarantees do not impair the relevant timing requirements.

Finally, we need to assure that the implementation of the pattern is performed correctly, i.e., that it conforms to the contracts and the conditions specified in the design pattern template. An argument pattern depicting the assurance of the application of an architectural pattern is presented in Figure 34. The structure of the argument pattern complies with the ISO/IEC 15026 standard, which specifies the minimum requirements for the structure and contents of an assurance case. According to ISO/IEC 15026, the essential conclusions of an assurance case are the uncertainties regarding truth or falsehood of the claims we assure. Led by the same thought, the purpose of the argument pattern for assurance of the application of an architectural pattern is to highlight all the uncertainties involved in the application of the architectural pattern. An example of the instantiated argument-pattern for the fault-tolerant Acceptance Voting Pattern is shown in Figure 35 and Figure 36.



**Figure 34.** High-level assurance argument-pattern for architectural pattern contract-based assurance

To automate the instantiation of the argument pattern for a specific architectural pattern, it should be possible in CHESS<sup>8</sup> to describe the architectural pattern (or its instantiation) with contracts and associate them with requirements addressing the targeted system problems. It should be possible to clarify the contract and its relation to the specific architectural pattern, as well as to associate evidence to support confidence in the architectural pattern and the related contract. The instantiation of the argument pattern

<sup>8</sup> <https://www.polarsys.org/chess/>

should be initiated either in CHES or in OpenCert<sup>9</sup>, and the result should be stored in the assurance case module related to the specific architectural element.

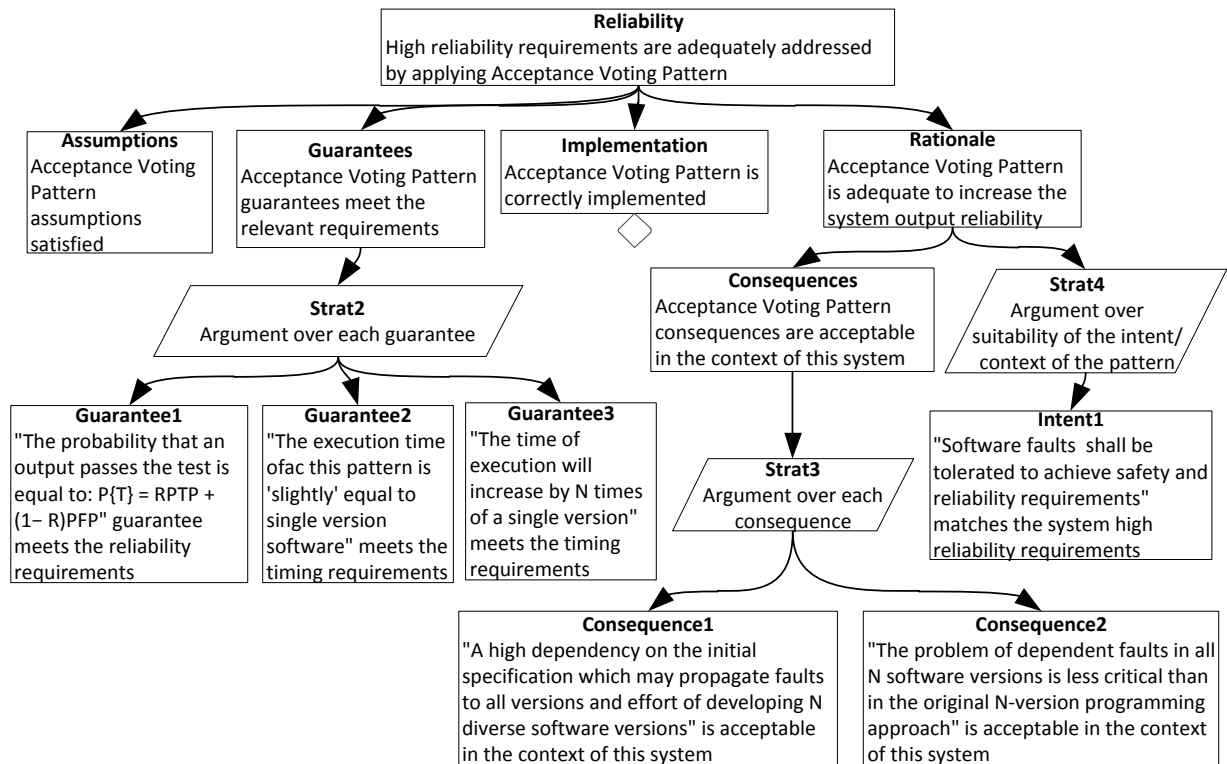


Figure 35. An argument example of the Acceptance Voting Pattern application

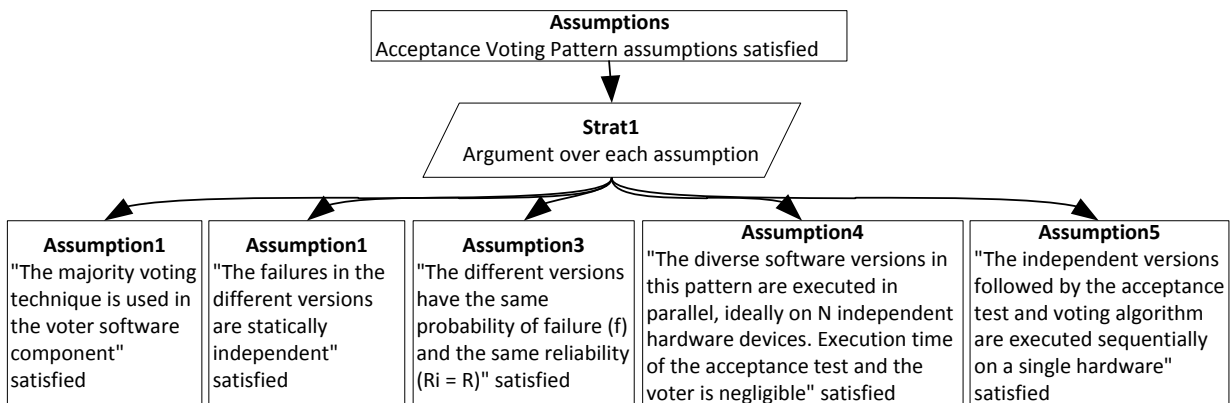


Figure 36. The Acceptance Voting Pattern assumptions argument-fragment

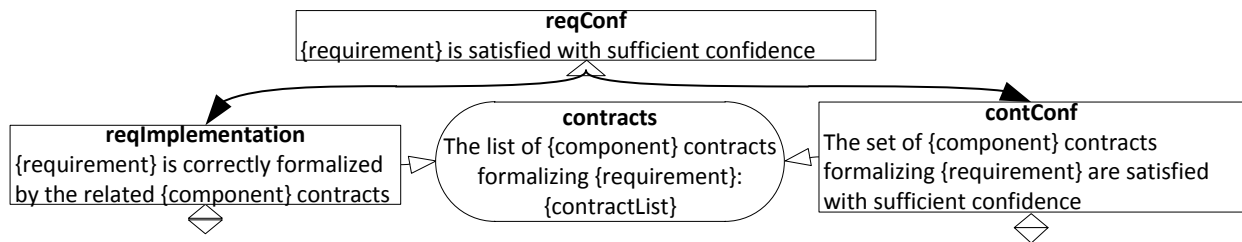
As stated in D3.1 [2], argumentation patterns can be related not only to fault tolerance but also to specific technologies such as safe access to shared resources related to multicore. The same approach described as part of automatic generation of product-based arguments can be applicable here.

However, mainly of the work has been done in relation with safety case patterns so far. Thus, assurance case patterns addressing technology specific solutions and considering some other concerns such as security will be possibly investigated in AMASS by considering the needs coming from the AMASS use cases implementation.

<sup>9</sup> <https://www.polarsys.org/proposals/opencert>

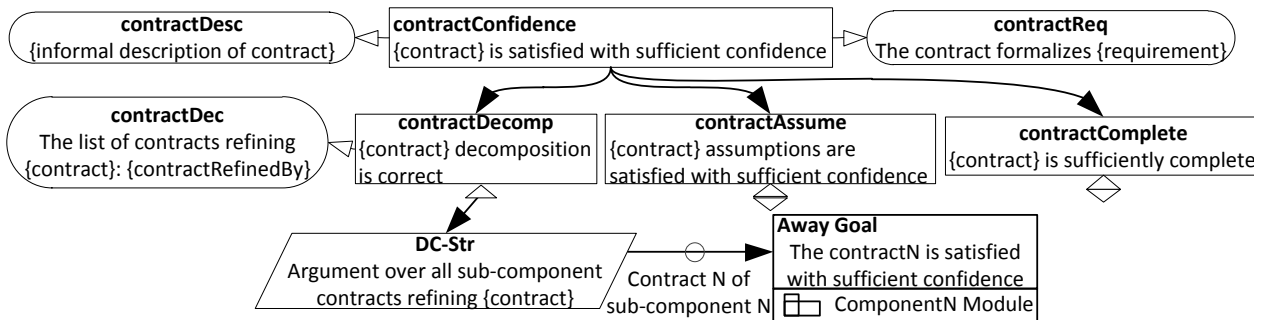
## 2.5.2 Assuring requirements based on OCRA results

Safety assurance is driven by safety requirements that are allocated to different components of the system. The corresponding contracts of those components are envisaged to formalize the allocated requirements. In that scenario each requirement is formalized or realized by a set of contracts. Establishing the validity of such requirements then boils down to checking the consistency and refinement of the contracts. But something more is needed to assure that the requirement is met by the system. To assure that a system satisfies a given safety requirement based on the related contract, we need to provide evidence that the contract correctly realizes the requirement (often said that its guarantees formalize the requirement) and evidence that the contract is satisfied with sufficient confidence in the given system context. We refer to this argument strategy as the contract-based requirements satisfaction pattern (Figure 37).



**Figure 37.** Contract-driven requirement satisfaction assurance argument pattern

While compositional verification of a system using contracts establishes validity of a particular requirement on the system model in terms of contracts, confidence that the system implementation actually behaves according to the contracts should also be assured. Hence, to drive the system assurance using contracts we have associated assurance assets with each contract. Those assets can be different kinds of evidence that increase confidence that the component (i.e., the implementation of the contracts) behaves according to



**Figure 38.** Contract satisfaction assurance argument pattern

the contract, i.e., that the component deployed in any environment that satisfies the contract assumptions exhibits the behaviours specified in the corresponding contract guarantees. To argue that a contract is satisfied with sufficient confidence we need to assure that the component actually behaves according to the contract, and that the environment in which the component is deployed satisfies the contract assumptions [57]. But when we deal with hierarchical systems where contracts are defined on each hierarchical level with well-defined decomposition conditions, then to argue that the composite component behaves according to the contract, we should explicitly argue over the component decomposition (Figure 38). We build on the contract-driven assurance argument patterns proposed as a part of SafeCer and SYNOPSIS [53] projects and adapt them to the compositional systems whose modelling and verification is facilitated in AMASS.

Just as the argument pattern for assurance of the application of an architectural pattern, these patterns share the same goal promoted through ISO/IEC 15026, to identify the uncertainties involved in assurance of the top-level claims. In this case, the component contracts are used to gather the evidence that a particular

requirement allocated to a component is satisfied. The presented patterns try to capture the uncertainties that need to be addressed to increase the confidence demonstrated through contracts that a particular requirement is satisfied by the component.

### 3. Design Level (\*)

In this chapter the design of the AMASS logical tool architecture and the metamodel supporting the concepts presented in the previous chapter are presented.

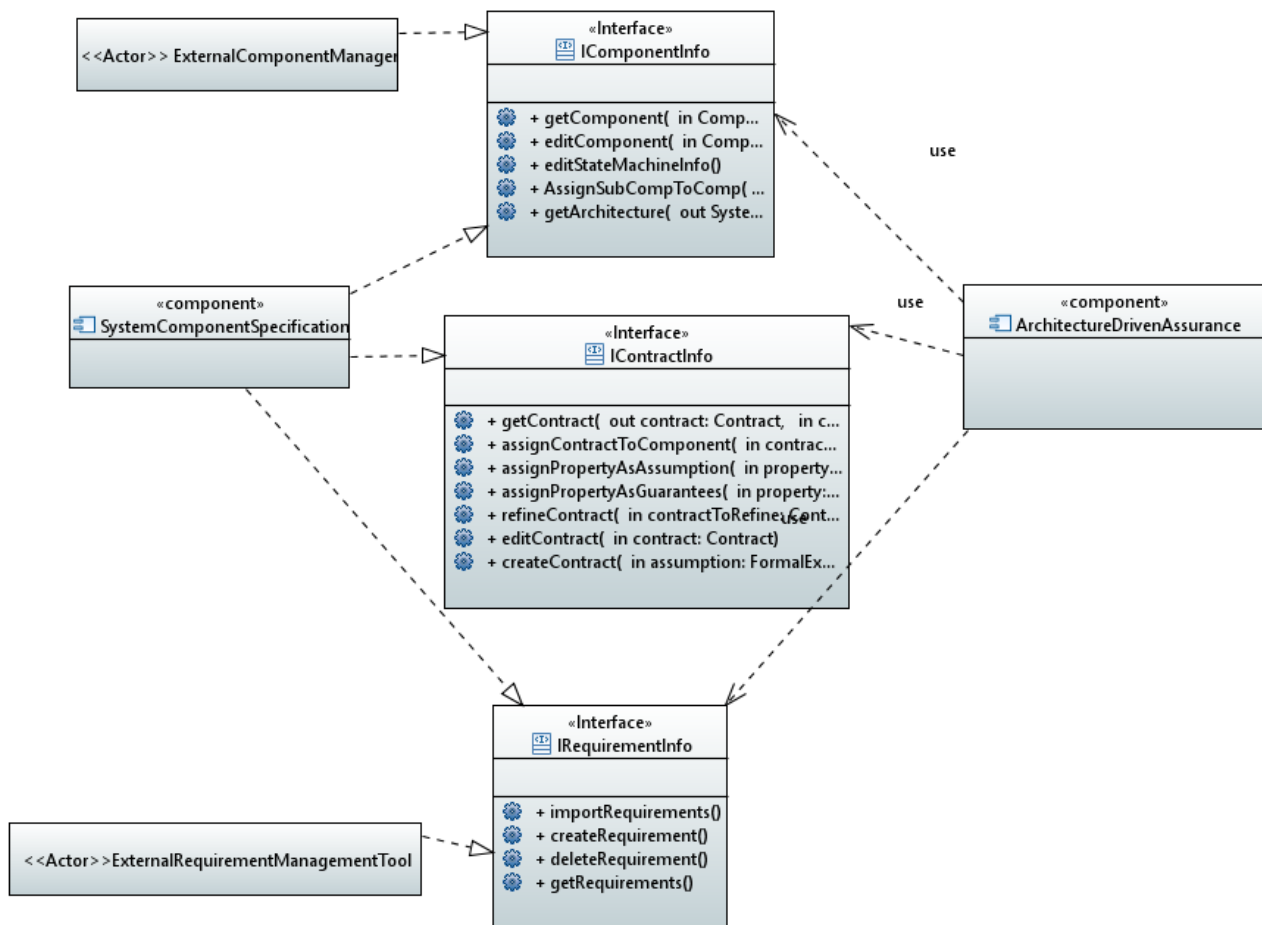
#### 3.1 Functional Architecture for Architecture Driven Assurance

This section illustrates the functional architecture in charge to support the approaches for architecture-driven assurance presented in the previous chapter. In particular, logical components supporting the features discussed at conceptual level have been identified. Standard UML notation is used to represent components with provided and required interfaces.

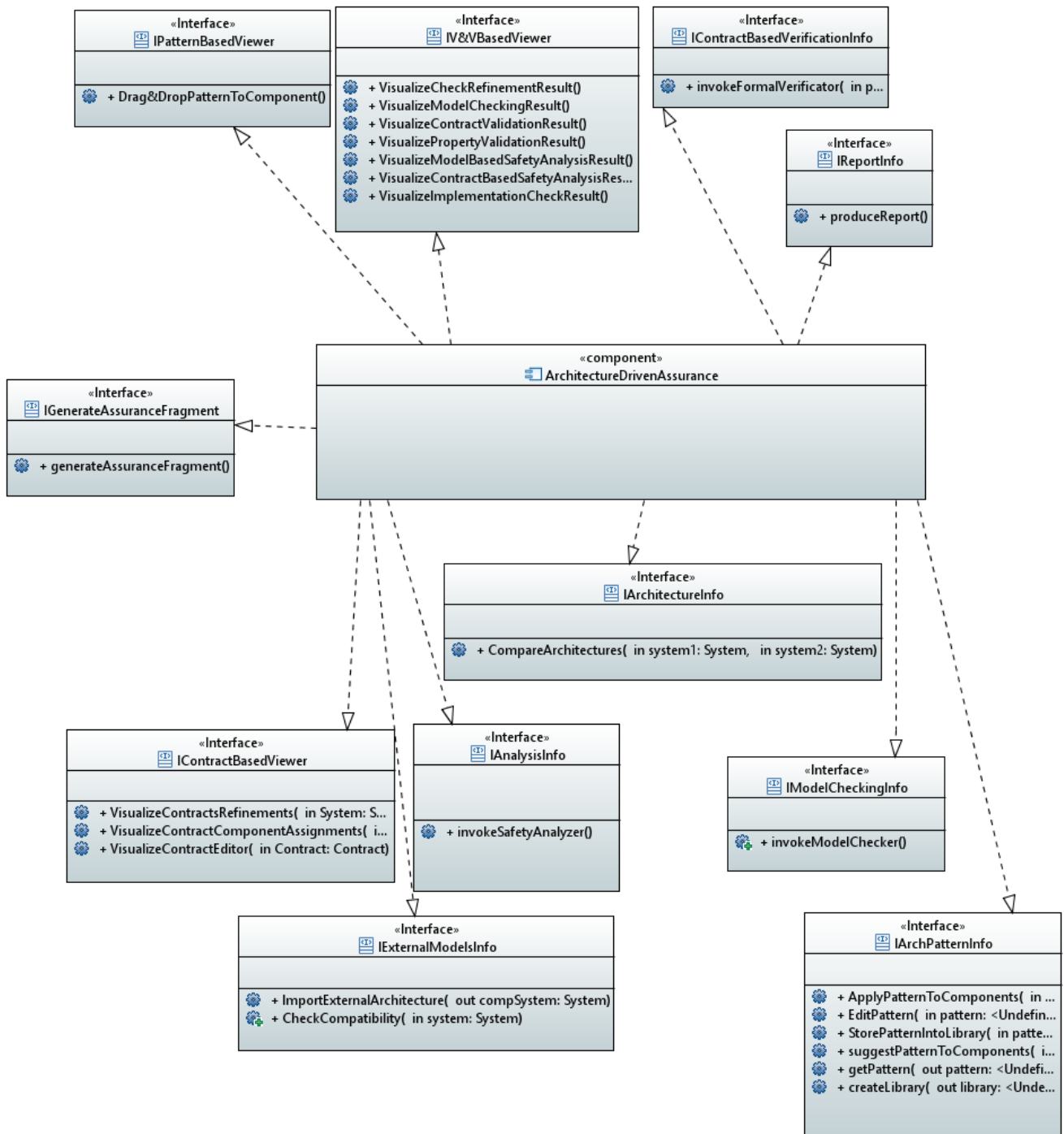
One of the main goals of the logical architecture regards the definition of the interfaces that will be provided/required by each logical component, so to clarify the responsibilities and collaboration between the components themselves, and possibly between the components here defined and the ones defined in the others technical WPs. The logical components here identified, together with their provided and required interfaces, will guide the implementation steps and the design of the overall AMASS logical architecture to be defined in WP2. Indeed, it is expected that a subset of the components and interfaces here presented will be referred in WP2 for the definition of the overall logical architecture of the AMASS reference tool architecture.

Figure 39 shows the *SystemComponentSpecification* basic building block, responsible for the management of the components and contracts modelling (a first implementation of this component based on Papyrus+CHESS has been provided in the context of the AMASS Core prototype release). *ArchitectureDrivenAssurance* represents the component that it is in charge to implement the services of the ARTA; these services are represented in Figure 40 as a set of provided interfaces; the services listed in the interfaces basically reflect the features presented in chapter 2, like the possibility to apply architectural patterns, V&V activities and contract based approach facilities.

*ArchitectureDrivenAssurance* component provides its features by using services provided by the *SystemComponentSpecification* (see Figure 39) and by using services provided by components external to the ARTA (see Figure 41) (in the diagrams the external components are tagged as actors, according to the UML semantics). We expect that the *ArchitectureDrivenAssurance* will be able to interface with external tools dedicated to system component specification: this is reflected in Figure 39 by the *ComponentManager* entity.

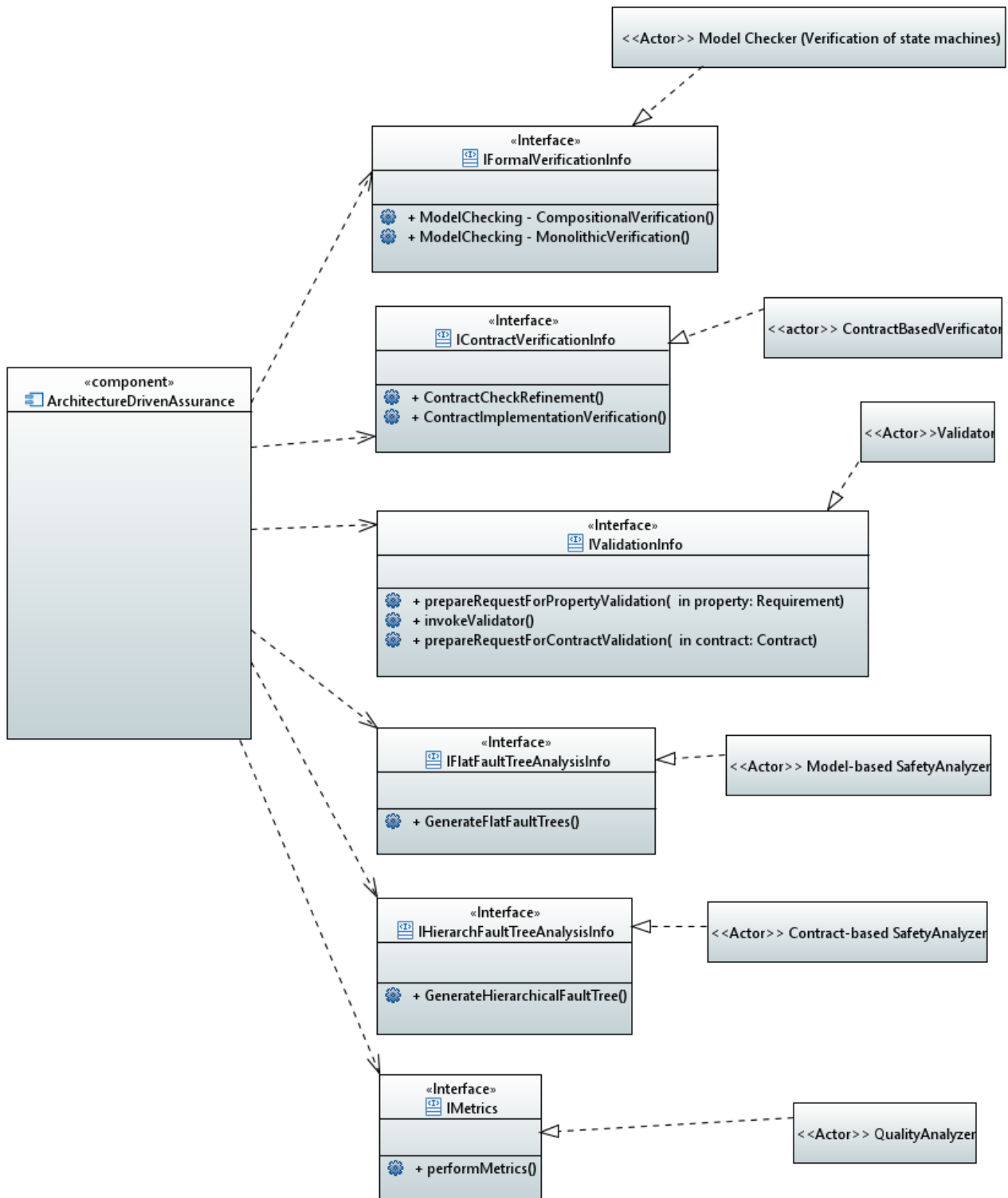


**Figure 39.** ARTA SystemComponentSpecification and ArchitectureDrivenAssurance components



**Figure 40.** ArchitectureDrivenAssurance components provided interfaces

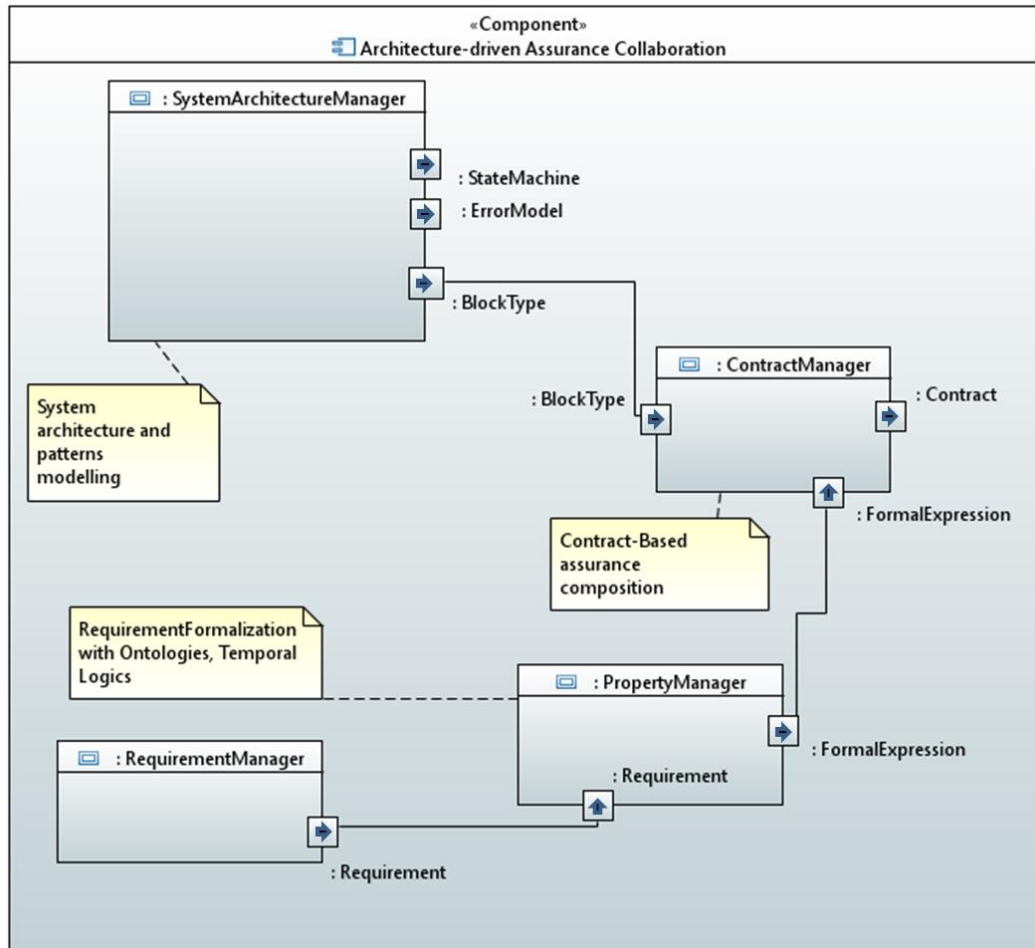




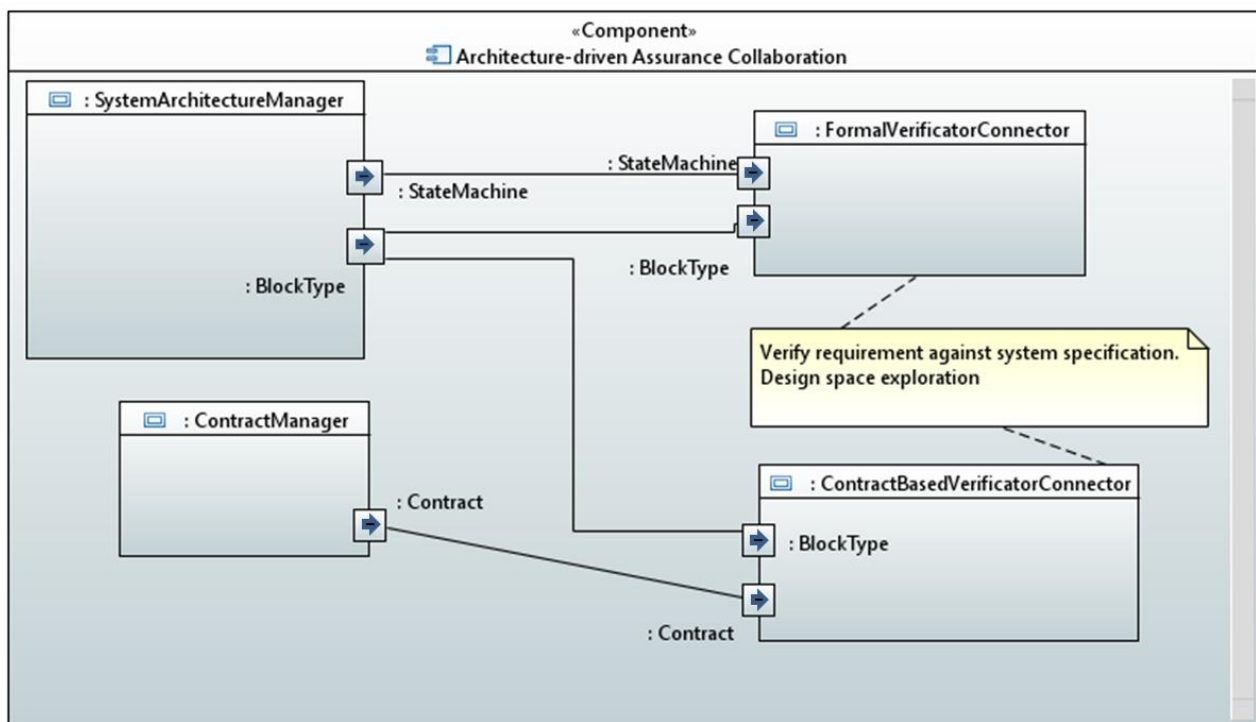
**Figure 41.** ARTA ArchitectureDrivenAssurance components and external actors/tools

A more detailed picture of the logical architecture is provided in Appendix B: Architecture-driven Assurance logical architecture (\*), where sub-components decomposing the components introduced above have been designed. The information about the WP3 requirements satisfied by each logical component is also provided in the diagrams.

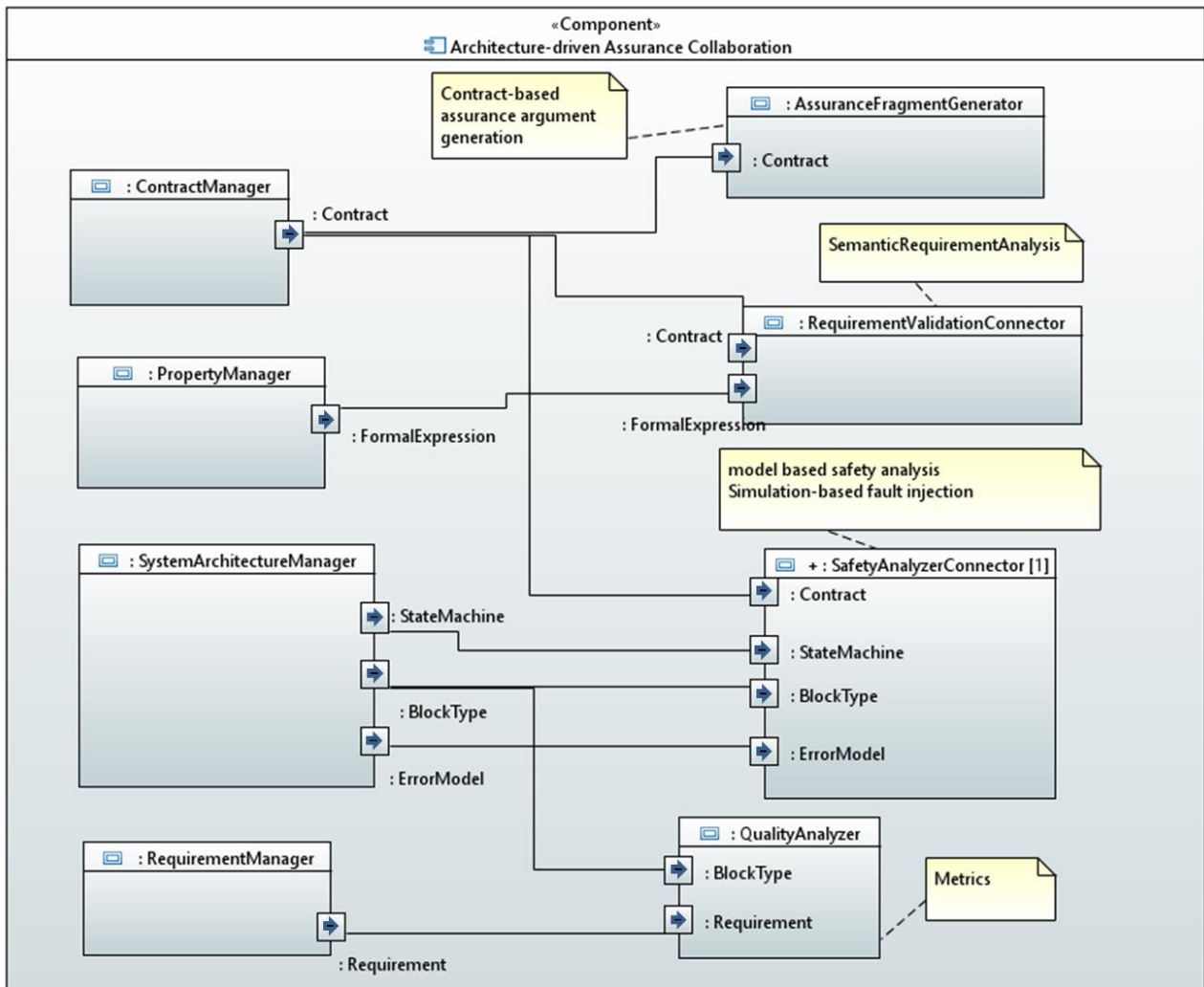
The collaboration diagrams provided in Figure 44 shows the aforementioned sub-components realizing the features presented Section 2; the data flow between the sub-components shows at logical level the dependencies of the realized features.



**Figure 42.** Logical Components Collaboration - part1



**Figure 43.** Logical Components Collaboration - part2



**Figure 44.** Logical Components Collaboration – part3

## 3.2 System Component Metamodel for Architecture-driven Assurance

This section illustrates the AMASS Component MetaModel supporting Architecture-driven assurance (CMMA). The metamodel is a review of the SafeCer metamodel [11] and embeds results from the SEooCMM metamodel [12].

This is an abstract metamodel, in the sense that it is used to elaborate the domain needs in an easier way (e.g. without the need to introduce and face with the complexity of existing standard component metamodels, like UML). The concepts available in CMMA will be made available to the modeller by using standard/existing metamodel(s) properly adapted, like UML and SysML tailored with the CHES profile (see Section 3.3).

The metamodel basically provides general architectural entities commonly available in standard modelling languages for system architectures, like UML/SysML and AADL. Moreover, it covers concepts related to contract based design approach, like the ones presented in Section 2.1.1. It is worth noting that the main goal of CMMA is not to provide a unified metamodel for system component, failure behaviour specification, etc. but to identify the links between architectural-related entities and the other parts of the CACM (e.g. argumentation, evidences), so to provide the model-based support for the architecture driven assurance approach.

### 3.2.1 Elaborations

This section presents some support that has been made available in the metamodel to cover the conceptual approaches discussed in chapter 2. The full CMMA specification is then given in Section 3.2.2. Further extensions of CMMA related to WP4 and WP6 conceptual level needs will be documented in D4.3 and D6.3 respectively, if needed.

#### 3.2.1.1 System Architecture Modelling

As stated above, the CMMA metamodel basically provides general architectural entities commonly available in standard modelling languages, like views, structured system components, components ports, connections, etc. This part is then enriched with other kind of constructs (like contracts) and traceability links to the CACM to support the architecture-driven approach.

#### 3.2.1.2 Representing Analysis

Section 2.1.1 elaborates about the need to represent specific analysis at model level in order to support argumentation reasoning.

A dedicated entity named *AnalysisContext* representing a specific analysis run is defined in CMMA; *AnalysisContext* allows to refer the entities in the model (e.g. architectural entities, failure behaviours) to be given in input to a given analysis tool, also together with specific information needed to set the specific analysis (e.g. analysis parameters), if needed.

*AnalysisContext* can be then traced to the artefact produced by the analysis and to the CACM entity representing the tool which has been used to run the analysis.

#### 3.2.1.3 Failure Behaviour Specification

As elaborated in Section 2.1.1, failure behaviour specification plays an important role in the modelling of the system components, e.g. to support model based V&V activities. In the context of CMMA, generic entities related to safety behaviour are introduced, mimicking the ones introduced in Section 2.1.1. More fine-grained support for failure behaviour specification is expected to be available in “concrete” metamodel for system component specification (like UML+CHES dependability profile, AADL error model annex). Direct connections between failure behaviour specification and other CACM entities have not been introduced; e.g., as presented in Section 2.1.1, the connection between the safety case and the entities of the failure behaviours (see Figure 5) can be derived from the FailureAnalysis concept, which in CMMA is represented by the *AnalysisContext* concept.

#### 3.2.1.4 Link to assurance case, evidence and process models

To support the link between system architecture and assurance, the following information has been addressed in the CMMA:

- As discussed in Section 2.1.1, the idea is that any instance in the architecture could reference to its associated arguments included in a component assurance case; to support this idea, the CMMA metamodel identifies the connection between an instance-entity of the system architecture (BlockInstance, Section 3.2.2.3.2) and the argumentation entity.
- Contract and assurance case: as discussed in section 2.5, the contract-based requirements satisfaction pattern is proposed in AMASS as general argument strategy. To support this approach at model level, several traceability links have been identified between contract, assurance case and evidence entities (see Section 3.2.2.4). These links facilitate connecting the system model with the assurance case such that it makes available the information needed for automated instantiation of argumentation patterns for contract-based assurance of requirements.
- There is a clear relationship between the design architecture and the work-products defined at the process level; these relationships must be identified to support the demonstration of the

compliance of the architecture with respect to a given process. In this regard CMMA identifies traceability links between system architecture to CACM process activities, i.e. the executed process. Concerning the planned process, we could have a set of information related to a tool/system methodology for system design which can be related to a process plan/standard and possibly reused when the tool/methodology is applied in the context of a given system design; this traceability aspect addressing the planned process will be investigated in the context of WP6.

### 3.2.1.5 Support for architectural patterns

CMMA introduces a basic support for patterns definition and instantiation, while focusing on the contract-based approach applied to patterns and then the enabling of the assurance patterns application (as discussed in Section 2.5.1).

## 3.2.2 CMMA Metamodel specification

### 3.2.2.1 Modelling out of context

This part of the metamodel concerns the constructs that can be used to model entities out of a given context, i.e. reusable units.

#### 3.2.2.1.1 Block Type

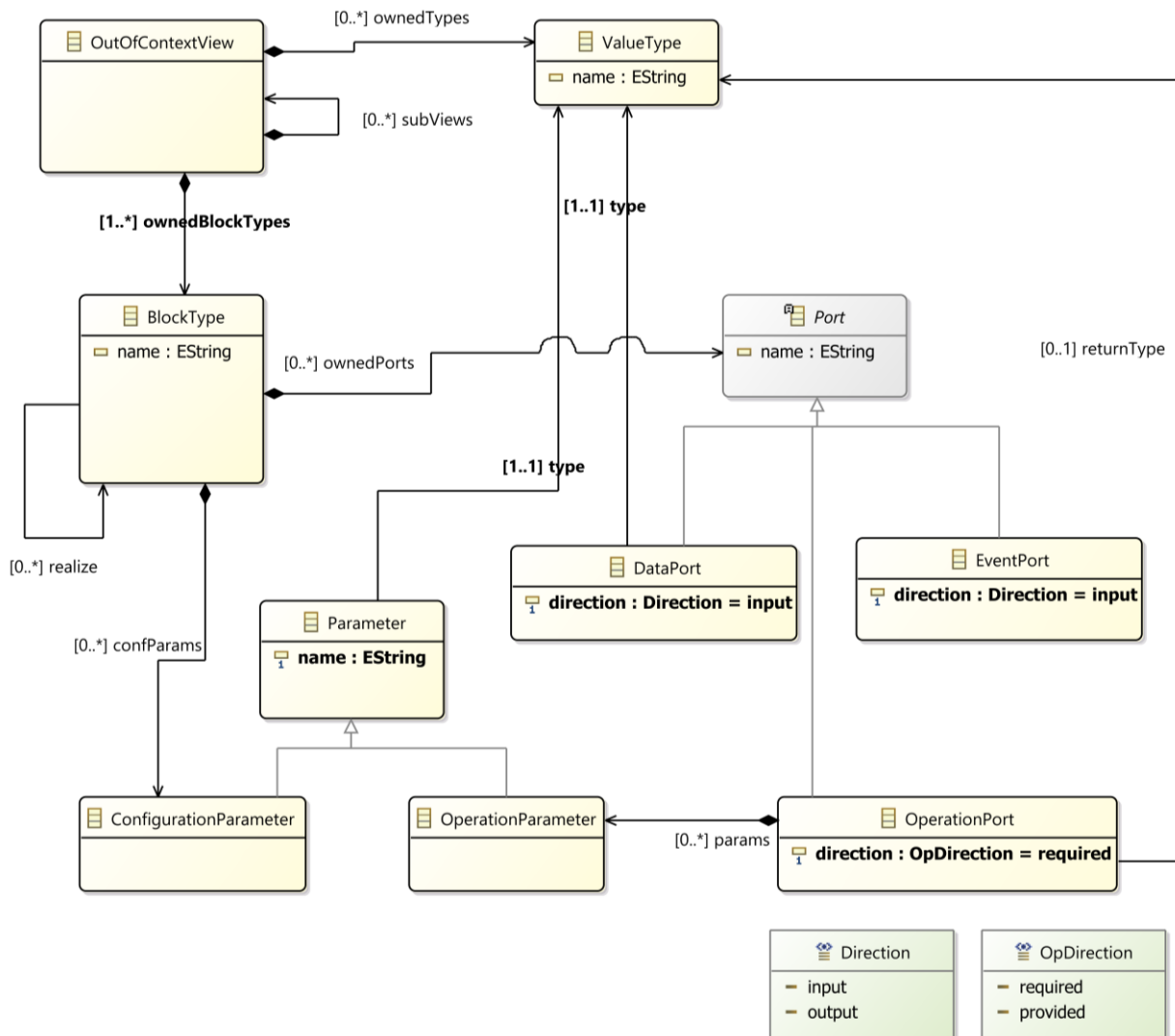
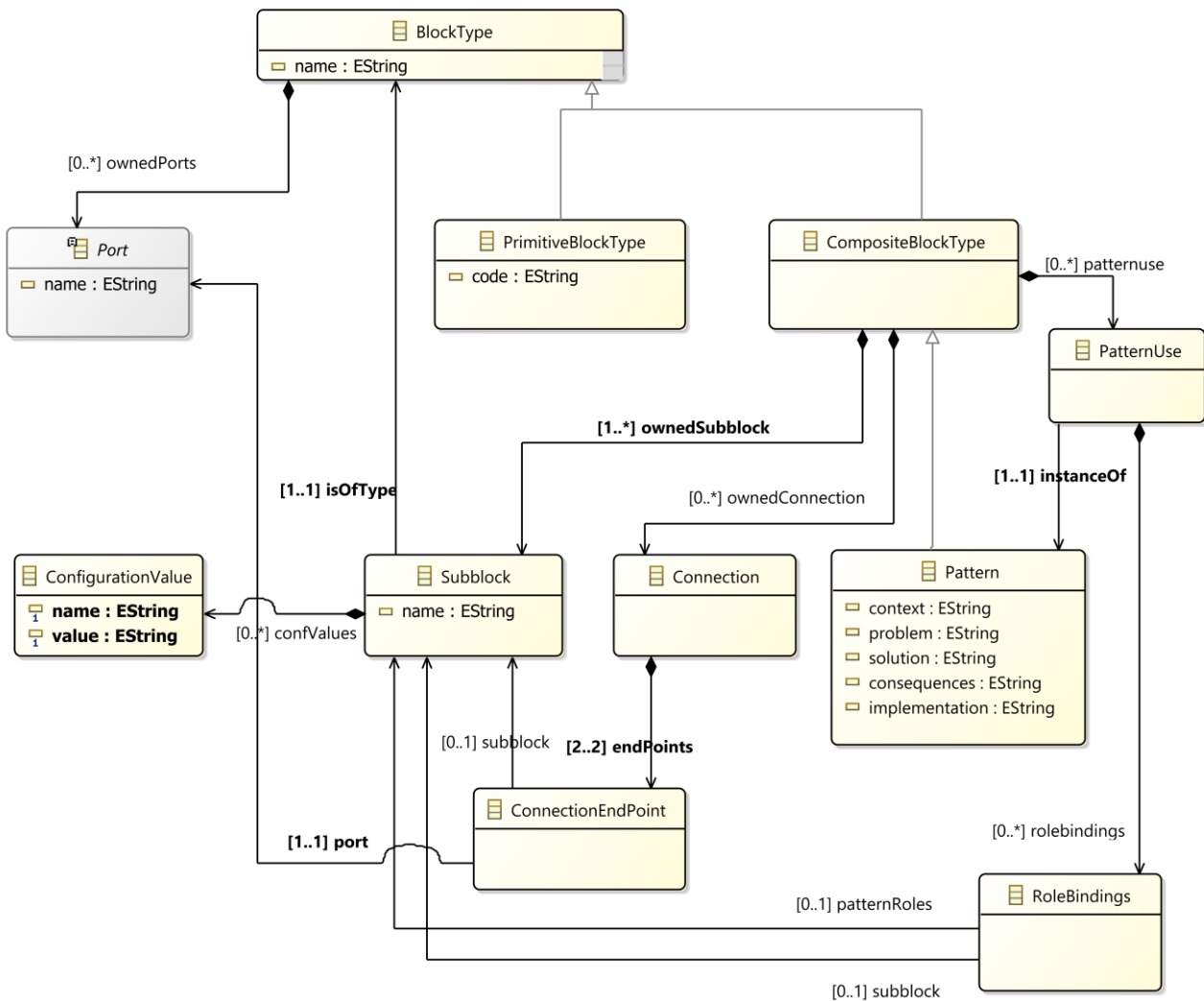


Figure 45. BlockType



**Figure 46.** Composite BlockType

A BlockType (Figure 45) represents a reusable unit out-of-context, i.e., a collection of features that are constant regardless of the context in which it is used. It can be used to represent any kind of system entities, e.g. HW, SW, functional, human.

The *realize* relationships allows to model that a block implements the functionality provided by other (more abstract) blocks, e.g. to model function blocks realized through SW/HW blocks.

### 3.2.2.1.2 Port

A Port (Figure 45) represents an interaction point through which data can flow between the block and the context where it is placed. This is an abstract meta-class.

There are three types of ports: Data, Event and Operation ports. Each port also has a specified direction.

- **Data port:** A Data port is a point of interaction where typed data can be sent or received by the block. The direction of a data port is either output (sending data) or input (receiving data).
- **Event port:** An Event port is a point of interaction where events can be sent or received by the block. The direction of an event port is either output (sending events) or input (receiving events).
- **Operation port:** An Operation port is a point of interaction corresponding to a function or method, with a number of typed parameters and return type.



#### **3.2.2.1.3 ConfigurationParameter**

Configuration parameters (Figure 45) represent points of variability in a BlockType. They allow formulation of more detailed contracts by including them in assumptions or in the form of parametric contracts. For example, a contract could specify that the component requires at most  $10+5*queue\_length$  units of memory, where *queue\_length* is one of the configuration parameter defined for the component type.

It can be set when the BlockType appears as Subblock or when it is instantiated (see BlockInstance) in a given system.

#### **3.2.2.1.4 Subblock**

Subblock (Figure 46) represents a part of a decomposed BlockType. A Subblock is an occurrence of a given BlockType inside a parent BlockType.

Subblock is different from the BlockInstance concept (see 3.2.2.3.2) since Subblock is an occurrence of a BlockType in the context of a BlockType, while a BlockInstance is an occurrence of a BlockType in the context of a System.

When a composite BlockType is instantiated in a given system, its Subblocks are instantiated as well; in this way the Subblocks can be further configured at instance level.

Subblocks of the same parent BlockType can be connected together through ports. Also, Subblocks ports can be connected to the ports of the parent BlockType.

#### **3.2.2.1.5 Connection and ConnectionEndPoint**

Connection and ConnectionEndPoint (Figure 46) allow to connect Subblocks through the ports defined for the corresponding/typing BlockTypes.

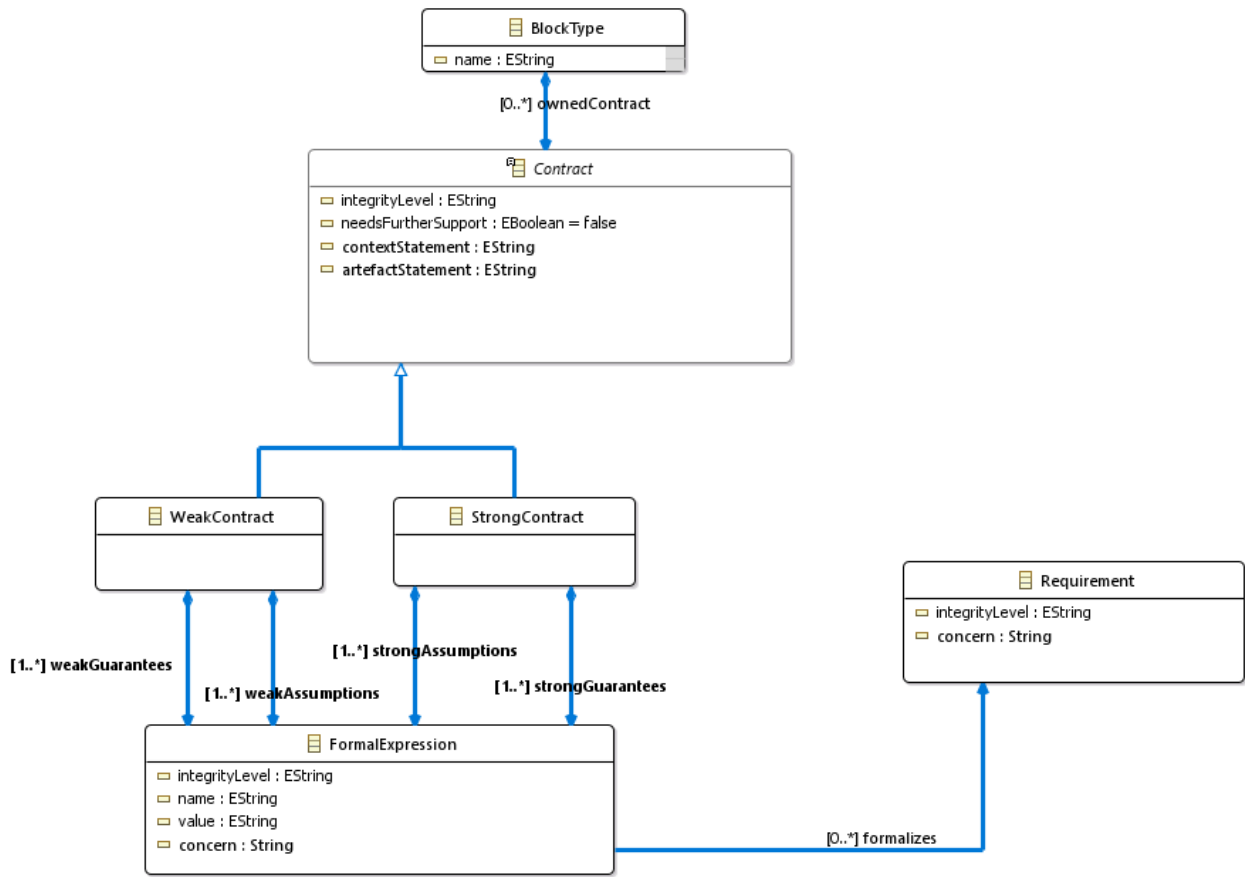
#### **3.2.2.1.6 Patterns**

Patterns are represented in the metamodel as a kind of composite block (Figure 46), where the internal part (Subblocks) represent the roles of the pattern. Pattern comes with the set of attributes presented in Section 2.2.1.1. The link between Patterns and Contracts is derived through BlockType (see Section 3.2.2.2).

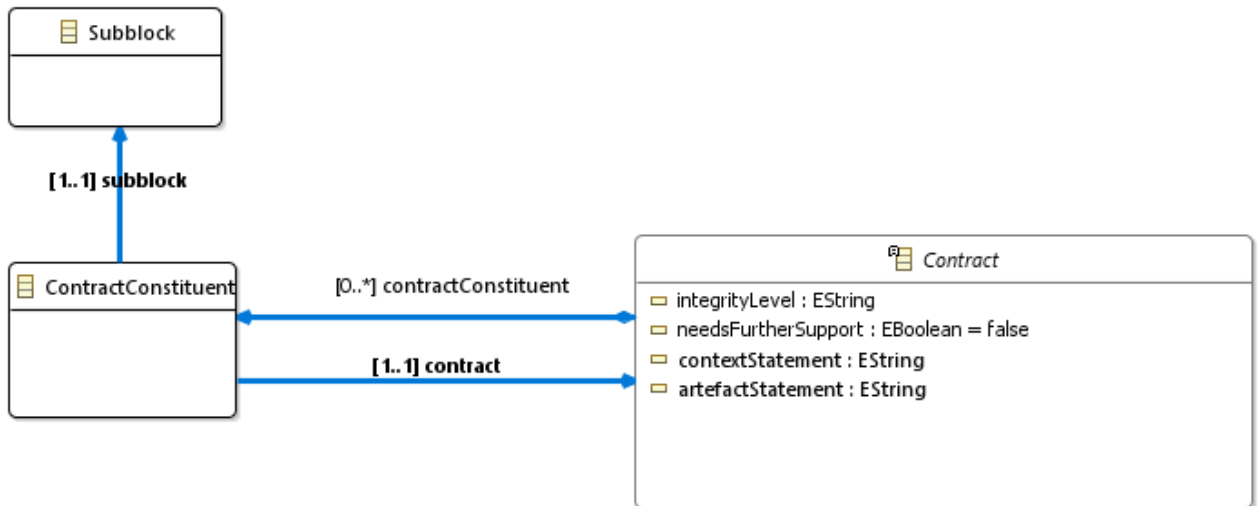
PatternUse (Figure 46) represents the application of a given Pattern in the context of a composite block; PatternUse owns the traceability information about the parts of the composite block playing the roles owned by the instantiated pattern.

### **3.2.2.2 Contracts**

This part of the metamodel regards the constructs which enable contract-based design.



**Figure 47.** Contract



**Figure 48.** Contract refinement

### 3.2.2.2.1 Contract

Contracts (Figure 48) represent information about the block type, bundled together with explicit descriptions of the assumptions under which the information is guaranteed.

The general format of a contract can be defined as:

<A, G, {<B1, H1>, ... , <Bn, Hn>}>

Where:

- *A* defines the strong assumptions that must hold in any context where the component type is used.
- *G* defines strong guarantees that always hold with no additional assumptions.
- *Bi* are weak assumptions that describe specific contexts where additional information is available.
- *Hi* are weak guarantees that are guaranteed to hold only in contexts where *Bi* hold.

A block type should never be used in a context where some strong assumptions are violated, but if some weak assumptions do not hold, it just means that the corresponding guarantees cannot be relied on.

Contract can have an integrity level stating the level of argumentation to be provided about the confidence in the contract; better semantic can be provided for integrity level according to adopted safety standard. Integrity level can be inherited by the Requirements associated to the Contract through FormalExpression.

The *needsFurtherSupport* Boolean attribute indicates if the contract is fully validated; if it is false, only partial evidence is provided with the contract and additional evidence should be provided.

#### **3.2.2.2.2 FormalProperties**

FormalProperty (Figure 48) represents a formalization of a requirement; it appears as assumption or guarantee of a Contract.

#### **3.2.2.2.3 ContractConstituent**

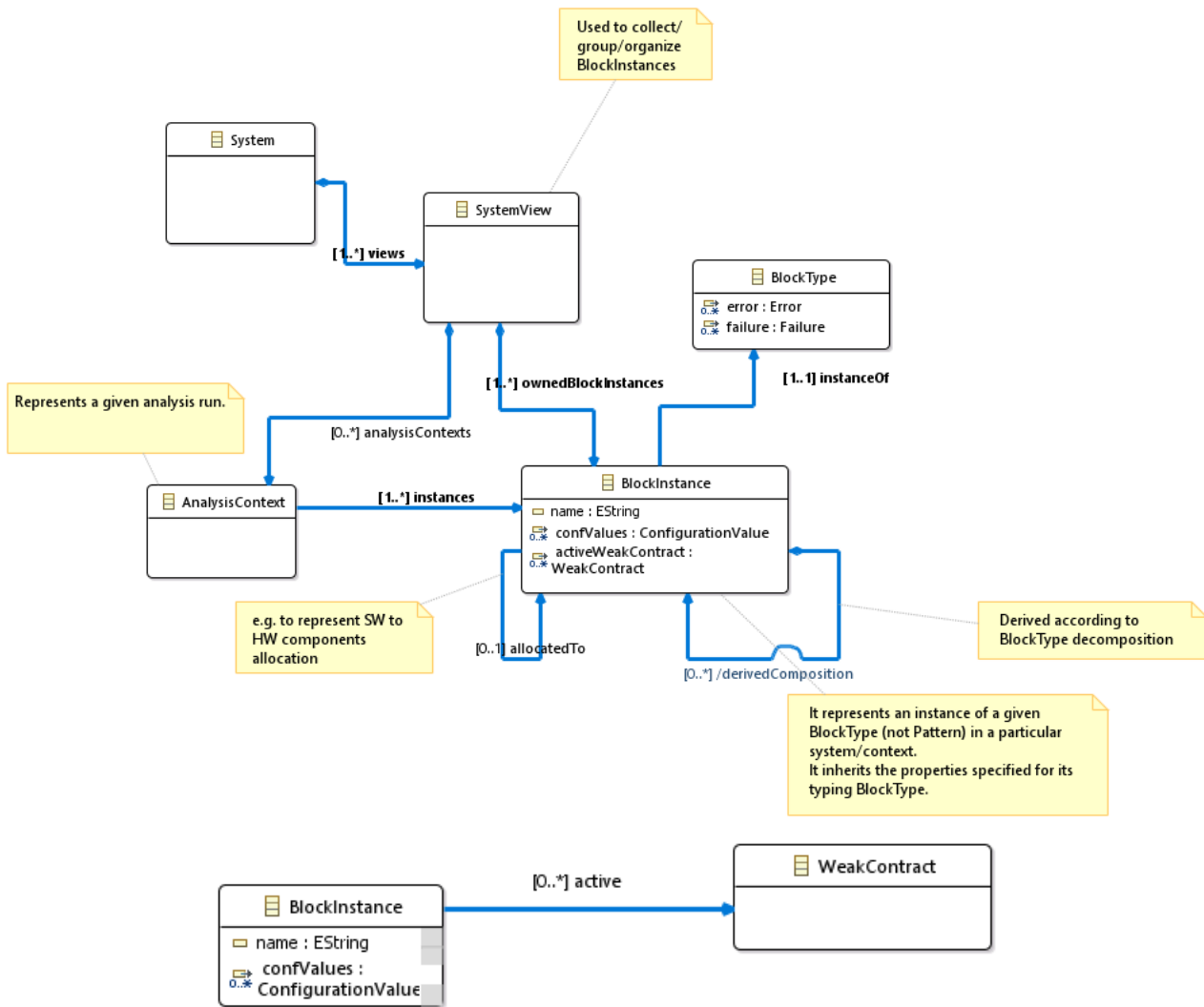
ContractConstituent (Figure 48) is used to model contract refinement along the blockType decomposition, i.e. between BlockType and its parts (Subblocks).

E.g.: supposed to have contract C1 associated to BlockType B1, and B1 is decomposed into B1\_1 and B1\_2 Subblocks. B1\_1 has contract C2 and B1\_2 has contract C3 associated.

Then, ContractConstituent allows modelling that contract C1 is decomposed by the B1\_1.C2 and B1\_2.C3 contracts. In particular the “contract provided by a given Subblock” (e.g. B1\_1.C2) is the kind of information stored in ContractConstituent.

#### **3.2.2.3 Modelling in a given context**

This part of the metamodel regards the constructs that can be used to model entities placed in a given context/system.



**Figure 49. System**

### 3.2.2.3.1 System

A System (Figure 49) represents a given cyber-physical system under design. Hold references to owned block instances through *software* and *platform* association; typically, the latter are created by instantiating a root composite BlockType.

### 3.2.2.3.2 BlockInstance

BlockInstance (Figure 49) represents an instance of a given BlockType in a particular system/context; it inherits the properties (ports, parameters, contracts, subblocks) as specified for its typing BlockType. In particular the decomposition structure defined for the typing BlockType is replicated at instance level through the *derivedComposition* link.

It has *allocatedTo* relationship to be used to model allocation of block instances, for instance like SW to HW instance blocks deployment.

The *active* link on the BlockInstance allows to specify the weak contracts associated to the typing BlockType which hold for a given block instance. Note that this can have impact on the modelled contract refinement. E.g., if a weak contract has been used to decompose a parent strong contract, then if the weak contract does not hold in a given context, then the contract refinement is invalid for that particular context.

A BlockInstance inherits the links to the evidence and assurance entities available for:

- the StrongContracts associated to the typing BlockType
- the WeakContracts referred through the *active* relationships

### 3.2.2.3.3 AnalysisContext

AnalysisContext (Figure 49) allows to collect all the information required to run a given analysis execution; in particular it allows to refer the (sub)set of block instances to be analysed.

### 3.2.2.4 Failure Behaviour

This part of the metamodel regards the definition of the failure behaviour for a given BlockType. The entities depicted in the following figure mimics the ones presented in Section 2.1.1. Basically, a BlockType can be decorated with a set of possible faults effecting the BlockType itself; the faults can be linked to failures of the given BlockType. Each failure can be described with a given failure mode affecting the port of the BlockType.

It is expected that actual metamodels (e.g. provided by CHESSE or Medini, see Figure 6) will provide additional constructs to enrich the failure behaviours modelling (like about the impact of a failure mode on the nominal behaviours, quantitative value or qualitative expression for faults and failures occurrence).

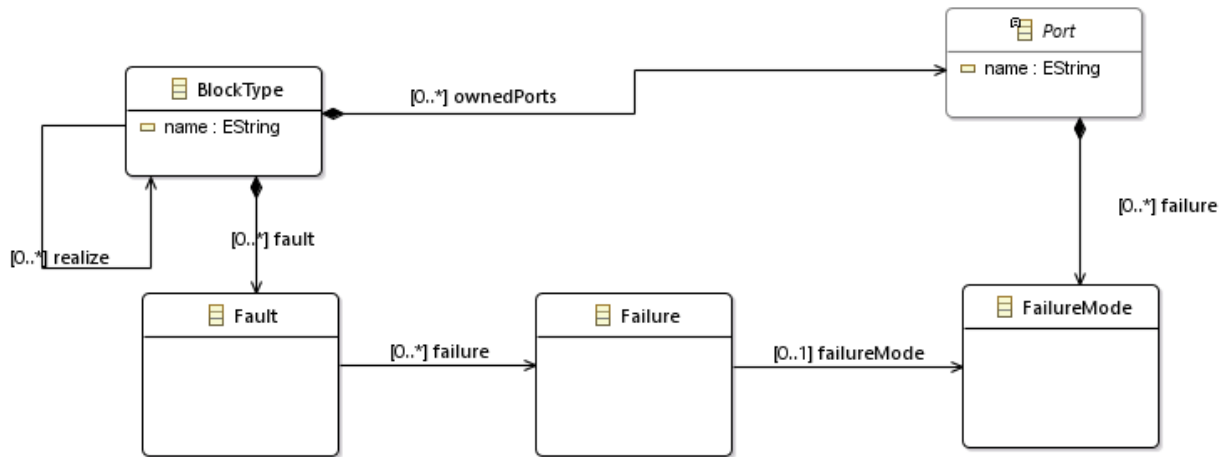
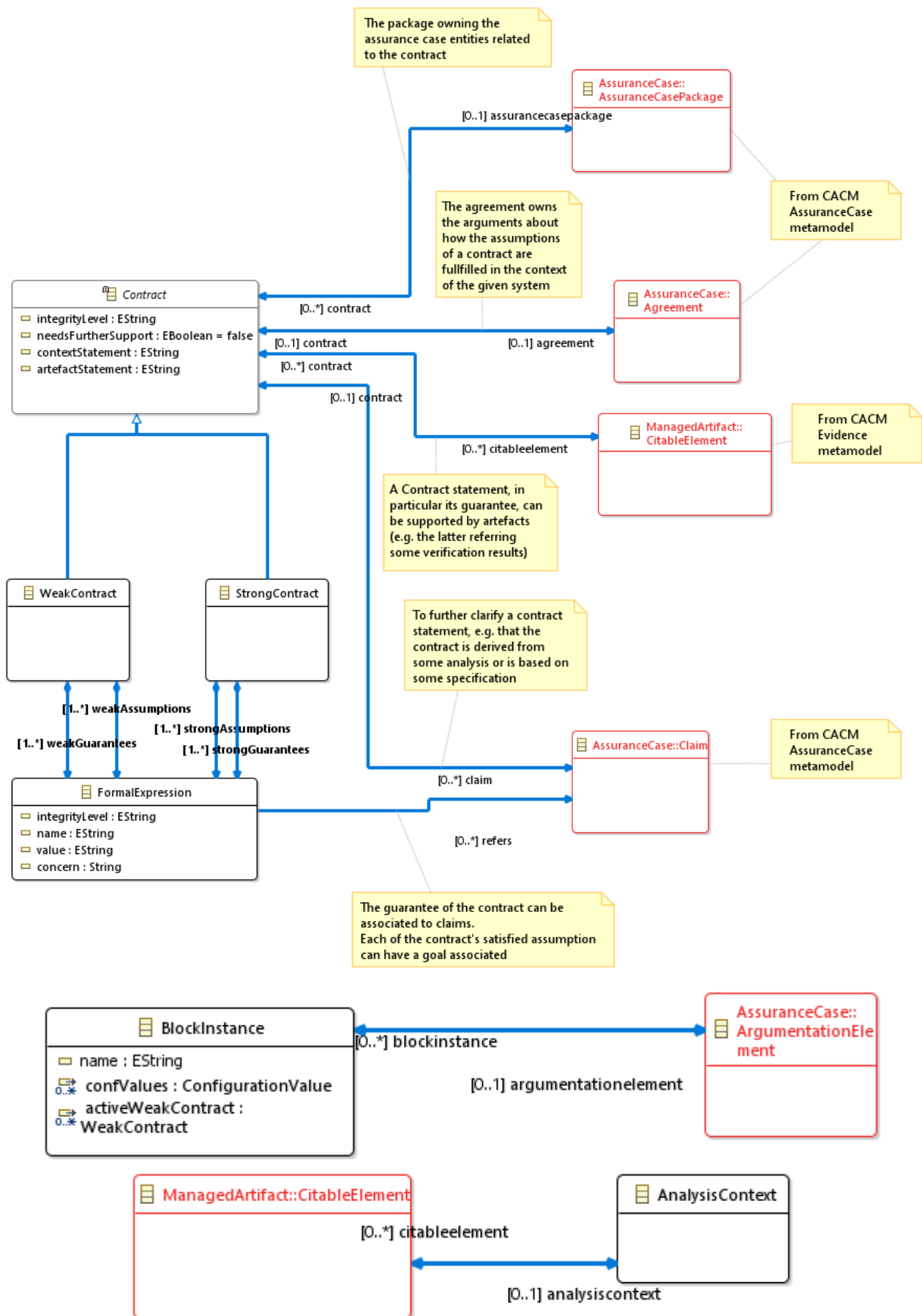


Figure 50. Failure Behaviour

### 3.2.2.5 Link to evidence and assurance cases

This part of the metamodel regards the connection to the assurance-related entities. Note that the connections are bidirectional, to allow navigation of the information from all the different perspectives (architecture, assurance case, evidence models).



**Figure 51.** Artefact and assurance-related entities connections

#### **3.2.2.5.1 CitableElement**

Imported from AMASS CACM Evidence Metamodel (see AMASS D2.2 [3] and updates).

#### **3.2.2.5.2 Claim**

Imported from AMASS CACM Assurance Case Metamodel (see AMASS D2.2 [3] and updates).

#### **3.2.2.5.3 AssuranceCasePackage**

Imported from AMASS CACM Assurance Case Metamodel (see AMASS D2.2 [3] and updates).

#### **3.2.2.5.4 Agreement**

Imported from AMASS CACM Assurance Case Metamodel (see AMASS D2.2 [3] and updates).

#### **3.2.2.5.5 ArgumentationElement**

Imported from AMASS CACM Assurance Case Metamodel (see AMASS D2.2 [3] and updates).

#### **3.2.2.5.6 BlockInstance**

The BlockInstance entity (see 3.2.2.3.2) is extended with the following relationship:

- referenceArgumentation: ArgumentationElement
  - the arguments associated to the block instance

#### **3.2.2.5.7 Contract**

The Contract entity (see 3.2.2.2.1) is extended with the following relationships:

- assuranceCase: AssuranceCasePackage
  - the package(s) owning the assurance case entities related to the contract
- agreement: Agreement
  - the agreement owns the arguments about how the assumption of a contract are fulfilled in the context of the given system
- supportedBy: CitableElement
  - allows to model that a Contract statement, in particular its guarantees, can be supported by artefacts (e.g. the latter referring some verification results)
- claim: Claim
  - the referred claim allows to further clarify a contract statement; e.g. that the contract is derived from some analysis or is based on some specification

The Contract entity is extended with the following attributes:

- contextStatement: String
  - store the informal description of what the contract means (which would be the context statement in the corresponding argumentation)
- artefactStatement: String
  - explain how a particular artefact relates to the contract (e.g., whether a contract is derived from the artefacts or the artefacts support that the implementation behaves according to the contract, etc.).

#### **3.2.2.5.8 FormalExpression**

The FormalProperty entity (see 3.2.2.2.2) is extended with the following relationships:

- Refers: Claim
  - Allows to map the guarantees of the contract to claims
  - Allows to associate a claim (e.g. GSN away goal) to each of the contract's assumptions.



### 3.2.2.5.9 AnalysisContext

The AnalysisContext (see 3.2.2.3.3) is extended with a relationship to the artefacts produced by the corresponding analysis execution.

### 3.2.2.6 Link to executed process

This part of the metamodel regards the connection with the CACM executed process.

The subset of the CMMA entities that can play the role of input or output artefact for a given process activity is shown in Figure 52.

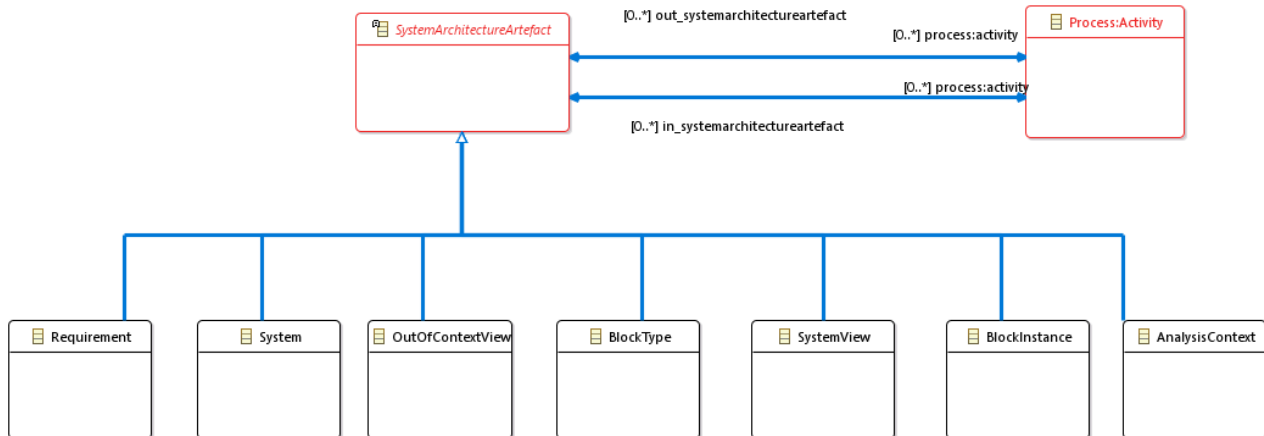


Figure 52. Links to the executed process

## 3.3 CHES Modelling Language

CHES UML/SysML<sup>10</sup>/MARTE<sup>11</sup> profile has been adopted in the AMASS reference tool architecture, as “instantiation” of the CMMA abstract metamodel, as part of the System Component Specification basic building block.

UML and SysML allow modelling the architectural design of the system, so covering the CMMA parts related to the out of context and in-context architectural parts.

CHES profile provides support for contract-based design: an introduction of the CHES profile related to the support for contract-based design has been reported in AMASS D3.1 [2] Appendix B; this part of the profile will be extended in the context of the second prototype iteration to support the new metamodel concepts introduced in Section 3.2.1.

Regarding the AnalysisContext introduced in 3.2.1.2, here the support coming from the MARTE profile will be evaluated. MARTE already provides constructs to support generic concepts for types of analysis based on system execution behaviour (see in particular MARTE Generic Quantitative Analysis Modelling (GQAM) sub-profile from the MARTE specification).

Regarding failure behaviour specification, CHES comes with a dedicated profile for dependability for modelling safety aspects related to the system architecture. The metamodel from which the CHES dependability profile has been derived is the SafeConcert metamodel [47] resulting from the ARTEMIS JU CONCERTO<sup>12</sup>; this metamodel is presented in Appendix C: SafeConcert metamodel, together with considerations about similarities with what has been presented in Section 2.1.1 about failure behaviour

<sup>10</sup> The OMG systems Modelling Language [www.omg.org](http://www.omg.org)

<sup>11</sup> UML Profile for MARTE: Modelling and Analysis of Real-time Embedded Systems.  
<http://www.omg.org/spec/MARTE/1.1>

<sup>12</sup> [www.concerto-project.org/](http://www.concerto-project.org/)

specification. Information about the CHES profile definition will be covered in the context of AMASS task T3.3.

## 4. Way forward for the implementation (\*)

The table below represents the requirements to be implemented in WP3, as derived from D2.1 [1]. A column here has been added to link the requirements to the sections where elaboration has been provided.

It is worth noting that some elaborations and approaches presented in Chapter 2 will be implemented as part of the third prototype (Prototype P2); the “Prototype N” column in the table below has not to be considered final here, it is currently under elaboration and its final version will be documented in the context of Task 3.3.

**Table 5.** WP3 requirements coverage

ID	Short Description	Description	Prototype Nº	Priority	Elaborated in section
WP3_APL_001	Drag and drop an architectural pattern	The system shall be able to instantiate in the component model and architectural pattern selected from the list of patterns stored.	Prototype P1	Should	2.2
WP3_APL_002	Edit an architectural pattern	The system should be able to edit, store and retrieve architectural patterns.	Prototype P1	Should	2.2
WP3_APL_003	Use of architectural patterns at different levels	The system shall be able to apply to the component model architectural patterns at different levels: AUTOSAR, IMA, Safety/Security Mechanisms (security controls).	Prototype P1	Should	2.2
WP3_APL_004	Architectural Patterns suggestions	The system could provide the user suggestions about a certain safety/security mechanisms stored as architectural patterns.	Prototype P2	Could	2.2
WP3_APL_005	Generation of argumentation fragments from architectural patterns/decisions	The system shall be able to generate arguments fragments based on the usage of specific architectural patterns in the component model.	Prototype P2	Should	2.1.3, 2.5.1
WP3_CAC_001	Validate composition of components by validating their assurance contract	The system shall be able to validate the composition of two or more components by validating the compatibility of the component contracts.	Prototype P1	Shall	2.3
WP3_CAC_002	Assign contract to component	The system shall allow associating a contract to a component. Then, the system shall allow dropping a contract from a component.	Core Prototype	Shall	2.3.1
WP3_CAC_003	Structure properties into contracts (assumptions/guarantees)	The system shall be able to support the extraction of assumptions and guarantees to be used in component contracts based on component properties.	Core Prototype	Must	2.3.1

WP3_CAC_004	Specify contract refinement	The system shall enable users to specify the refinement of the contract along the hierarchical components architecture.	Core Prototype	Shall	2.3.1
WP3_CAC_005	General management of contract-component assignments	The system should enable users to have a view of the association between contracts and components for the entire system architecture (thus, not only a view on the single contract assignment for each component).	Prototype P1	Should	2.3.1
WP3_CAC_006	Refinement-based overview	The system should enable users to have a hierarchical view of the contract refinements along the system architecture.	Prototype P1	Should	2.3.1
WP3_CAC_007	Overview of check refinements results	The system should enable users to have an overview in terms of status of check refinement of all the defined contracts.	Prototype P1	Should	2.3.1
WP3_CAC_008	Contract-based validation and verification	The system must provide support for contract-based system validation and verification, including refinement checking, compositional verification of behavioural models, and contract-based fault-tree generation.	Prototype P1	Must	2.4.3.6
WP3_CAC_009	Improvement of Contract definition process	The operation of contract definition should be improved in terms of time spent.	Prototype P1	Should	2.3.1
WP3_CAC_011	Overview of contract-based validation for behavioural models	The system could enable users to have an overview of the validation of a contract over a state-machine. In case of failure, the system could enable users to have information about the trace that does not fulfil the contract.	Prototype P1	Could	2.3.1
WP3_CAC_012	Browse Contract status	The user shall be able to browse the contracts associated within a component and their status (fulfilled or not).	Core Prototype	Must	2.3.1
WP3_CAC_013	Specify contracts defining the assumption and the guarantee elements	The system shall provide the capability to create a contract defining two new properties (assumptions/guarantees) implicitly associated to that contract.	Core Prototype	Should	2.3.1
WP3_SAM_001	Trace component with assurance assets	The supplier of a component shall be able to trace all the assurance information with the specific component.	Core Prototype	Must	3.2

WP3_SA M_002	Impact assessment if the component changes	The system shall provide the capability for a component change impact analysis.	Prototype P2	Shall	2.1.2 This requirement will be also supported by the work performed in WP5 related to traceability support.
WP3_SA M_003	Compare different architectures according to different concerns which have been specified before	The system shall be able to compare different system architectures based on predefined criteria, like dependability or timing concerns.	Prototype P2	Could	2.4.6
WP3_SA M_004	Integration with external modelling tools	The system could interact with external tools for system design and development (e.g., Rhapsody, Autofocus, Compass) to get the system architecture.	Prototype P2	Could	2.1.4
WP3_SC_001	System abstraction levels browsing	The user shall be able to browse along the different abstractions levels (system, subsystem, component).	Core Prototype	Must	2.1.4
WP3_SC_002	System abstraction levels editing	The user shall be able to move and edit along the different abstractions levels (system, subsystem, component).	Core Prototype	Must	2.1.4
WP3_SC_003	Modelling languages for component model	The system shall be able to support different modelling languages to model the component/subsystem/system.	Prototype P2	Should	2.1.4
WP3_SC_004	Formalize requirements with formal properties	The system shall be able to specify requirements about a component in a formal way.	Core Prototype	Must	2.4.2.4
WP3_SC_005	Requirements allocation	The system shall provide the capability for allocating requirements to parts of the component model. More in general, requirements traceability shall be enabled.	Core Prototype	Must	2.4.2.4
WP3_SC_006	Specify component behavioural model (state machines)	The system shall be able to specify the component behavioural model.	Core Prototype	Must	Covered by the Core Prototype

WP3_SC_007	Fault injection (include faulty behaviour of a component)	The system shall have fault injection capabilities.	Core Prototype	Must	2.1.1, 2.4.7, 2.4.8
WP3_VVA_001	Traceability between different kinds of V&V evidence	The system shall provide the ability to trace immediate evidence (obtained during the execution of the left-hand side of the V-model) with direct evidence (obtained during the execution of the right-hand side of the V-model). For instance, a contract-based, component-based specification should be traced with the corresponding analysis-results.	Core Prototype	Should	2.1.2, 3.2.1.2
WP3_VVA_002	Trace model-to-model transformation	The system shall be able to trace all component model transformations executed during V&V model-based analysis.	Prototype P1	Must	2.1.2
WP3_VVA_003	Validate requirements checking consistency, redundancy, ... on formal properties	The system shall be able to validate formal requirements/properties.	Prototype P1	Must	2.4.3.6
WP3_VVA_004	Trace requirements validation checks	The system shall be able to trace requirements validations.	Core Prototype	Must	2.4.3.6
WP3_VVA_005	Verify (model checking) state machines	The system shall be able to verify that the component behavioural model matches with the specification.	Prototype P1	Must	2.4.5.3
WP3_VVA_006	Automatic provision of HARA/TARA-artefacts	The system shall provide the capability for automating HARA (Hazard Analysis Risk Assessment) / TARA (Threat Assessment & Remediation Analysis)-related artefacts (e.g., FTA, FMEA, and attack trees).	Prototype P2	Must	2.4.8
WP3_VVA_007	Generation of reports about system description/ verification results ....	The system shall generate reports about system/subsystem/component verification results.	Prototype P2	Must	2.1
WP3_VVA_008	Automatic test cases specification from assurance requirements specification	The system should be able to generate automatically the test cases specification based on the requirements definition.	Prototype P2	Shall	This requirement has been dropped

WP3_VVA_009	Capability to connect to tools for test case generation based on assurance requirements specification of a component/system	The system shall be able to connect to external tools to execute the test cases already specified.	Prototype P2	Shall	This requirement has been dropped
WP3_VVA_010	Model-based safety analysis	The system shall allow the user to generate fault trees and FMEA tables from the behavioural model and the fault injection.	Prototype P1	Must	2.4.8
WP3_VVA_011	Simulation-based Fault Injection	The system should allow the user to generate fault injection simulations from the fault trees and FMEA tables.	Prototype P2	Should	2.4.7
WP3_VVA_012	Design Space Exploration	The system could support the design space exploration of a system for a certain safety/security criticality level.	Prototype P2	Could	2.4.6
WP6_PPA_004	The AMASS tools must support specification of variability at the component level	The system shall enable users to specify what varies (and what remains unchanged) from one component and its evolved version at component level.	Prototype P1	Shall	Addressed in the context of WP6
WP6_RA_003	Reusable off the shelf components	The system shall provide the capability for reuse of pre-developed components and their accompanying artefacts.	Core Prototype	Must	2.3.2

## 4.1 Feedback from Core/P1 prototype evaluation

Deliverable D1.4 [6] and D1.5 [10]<sup>13</sup> report about the results of evaluating, respectively, the AMASS Core and P1 prototypes by industrial partners.

While Papyrus and CHESSE have been used by some use cases, other use cases have highlighted the need to use external tools for architecture specification like Rhapsody, Model-based Requirement Management Tool or Simulink. In WP3 we already have a dedicated requirement covering this aspect, about integration with external modelling tools, and in this deliverable we propose some approaches for its coverage, e.g. by proposing functionalities for traceability management with external tools (see Section 2.1.2), the latter under development in the context of WP5. For instance, support for traceability from AMASS platform to requirements managed in DOORS will be available by using an extension of the Capra [59] traceability tool (see AMASS deliverable D5.5 [9]).

Dedicated import facilities from external modelling tool to AMASS internal modelling tool are available in Papyrus (the system modelling tool adopted by the AMASS platform, together with the CHESSE extension) like the import from Rhapsody models. Section 2.1.4 also is related to the description of specific importers that will be developed in AMASS.

<sup>13</sup> At the time of writing this deliverable D1.5 is under finalization.



## 5. Conclusions (\*)

In this deliverable, we have elaborated our conceptual approach for architecture-driven assurance. In particular, in chapter 2 several approaches and features planned to be supported by the AMASS tool platform, also by using external tools, have been presented, together with argumentation fragments related to the system architecture modelling and verification and validation activities.

Then in chapter 3 we have presented the logical architecture and system component metamodel that will allow to support the aforementioned features and that will guide the implementation phase, discussed in chapter 4.

In the next period of the AMASS project we will focus on the finalization of the implementation of the presented approaches, to be made available as part of the AMASS final prototype, and on the definition of the methodological guidelines. We will analyse the feedback from the use cases implementation, in particular related to the application of the proposed argumentation fragments related to the contract-based approach. We will continue the investigation around assurance of patterns; for this goal, we will try to identify specific safety, security or technological patterns that could be of interest for the AMASS use cases first, and then work on that specific examples, by trying to apply the pattern contract-based assurance approach proposed in this deliverable.

## Abbreviations and Definitions

AADL	Architecture Analysis and Design Language
ADA	Architecture-Driven Assurance
ADAM	Architecture-Driven Assurance Metamodel
API	Application Programming Interface
ARTA	AMASS Reference Tool Architecture
ARTEMIS	ARTEMIS Industry Association is the association for actors in Embedded Intelligent Systems within Europe
ASIC	Application Specific Integrated Circuit
AUTOSAR	AUTomotive Open System Architecture
AV	Acceptance Voting
CACM	Common Assurance and Certification Metamodel
CBD	Contract-Based Design
CCL	Common Certification Language
CDO	Connected Data Objects
CHESSML	CHESS Modelling Language
CMMA	Component MetaModel for architecture-driven Assurance
CPS	Cyber-Physical Systems
CRC	Cyclic Redundancy Check
DIA	Development Interface Agreement
E2E	End to End
ECSEL	Electronic Components and Systems for European Leadership
EDC	Error-Detection-Correction codes
ETCS	European Train Control System
FI	Fault Injection
FMEA	Failure Modes and Effects Analysis
FMEDA	Failure Modes Effects and Diagnostics Analysis
FTA	Fault Tree Analysis
GQAM	Generic Quantitative Analysis Modelling
GSN	Goal Structured Notation
HARA	Hazard Analysis Risk Assessment
HAZOP	HAZard and OPerability study
HRELTL	Hybrid Linear Temporal Logic with Regular Expressions
HTTP	Hypertext Transfer Protocol
HW	Hardware
IEC	International Electro Technical Commission
IMA	Integrated Modular Avionics
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
LF	Latent Fault
LTL	Linear Temporal Logic
MA	Monitor-Actuator
MARTE	Modelling and Analysis of Real Time and Embedded systems
MCS	Minimal Correction Set

ML	Modal Logic
MoonN	M-out-of-N Pattern
MSS	Maximal Satisfiable Subset
MTL	Metric Temporal Logic
MUS	Minimal Unsatisfiable Subset
NLP	Natural Language Processing
OCRA	Othello Contracts Refinement Analysis
OEM	Original Equipment Manufacturer
OMG	Object Management Group
OSLC	Open Services for Lifecycle Collaboration
PSAC	Plan for Software Aspects of Certification
PUS	Packet Utilization Standard
RE	Requirement Engineering
RQA	Requirements Quality Analyzer
RSHP	Relation SHiP
RTCA	Radio Technical Commission for Aeronautics
RVS	Rapita Verification Suite
SCC	Semantic Clusters Nouns
SCS	System Component Specification
SCM	System Conceptual Model
SCV	Hierarchical Views Nouns
SE	System Engineering
SEooCMM	Safety Element out-of-context Metamodel
SLIM	System-Level Integrated Modelling Language
SMUT	System Model Under Test
SPF/LF	Single Point Failure/Latent Fault
SQA	System Quality Analyzer
SW	Software
SysML	System Modelling Language
TARA	Threat Assessment & Remediation Analysis
TLE	Top Level Event
UI	User Interface
UML	Unified Modelling Language
V&V	Verification and Validation
WP	Work Package
XML	eXtensible Markup Language

## References

- [1] [AMASS D2.1 Business cases and high-level requirements](#), 28<sup>th</sup> February 2017
- [2] [AMASS D3.1 Baseline and requirements for architecture-driven assurance](#), 30<sup>th</sup> September 2017
- [3] AMASS D2.2 AMASS reference architecture (a), 30<sup>th</sup> November 2016
- [4] AMASS D4.2 Design of the AMASS tools and methods for multiconcern assurance (a), 30<sup>th</sup> June 2017
- [5] AMASS D6.2 Design of the AMASS tools and methods for cross/intra-domain reuse (a), 31<sup>st</sup> October 2017
- [6] [AMASS D1.4 AMASS demonstrators \(a\)](#), 30<sup>th</sup> April 2017
- [7] AMASS D3.2 Design of the AMASS tools and methods for architecture-driven assurance (a), 30th June 2017
- [8] AMASS D2.3 AMASS reference architecture (b), 31<sup>st</sup> January 2018
- [9] [AMASS D5.5 Prototype for seamless interoperability \(b\)](#), 30<sup>th</sup> November 2017
- [10] AMASS D1.5 AMASS demonstrators (b), 16<sup>th</sup> April 2018
- [11] SafeCer Deliverable D132.2, Generic component meta-model v1.0, 2014-12-19
- [12] [http://www.es.mdh.se/pdf\\_publications/4435.pdf](http://www.es.mdh.se/pdf_publications/4435.pdf)
- [13] M. Bozzano, A. Cimatti, C. Mattarei, S. Tonetta: Formal Safety Assessment via Contract-Based Design. ATVA 2014: 81-97
- [14] A. Pnueli. The temporal logic of programs. In Proceedings of 18th IEEE Symp. on Foundation of Computer Science, pages 46–57, 1977.
- [15] Alessandro Cimatti, Marco Roveri, Angelo Susi, and Stefano Tonetta: Validation of Requirements for Hybrid Systems: a Formal Approach. ACM Transactions on Software Engineering and Methodology, Volume 21, Issue 4.
- [16] S. Schwendimann. A New One-Pass Tableau Calculus for PLTL. TABLEAUX 1998, 277-292.
- [17] P. Wolper. The Tableau Method for Temporal Logic: An Overview. Logique et Analyse 28.110-111 (1985).
- [18] K. Rozier and M. Vardi. LTL Satisfiability Checking. STTT 12.2 (2010).
- [19] M. Fisher: A Resolution Method for Temporal Logic. IJCAI 1991, 99-104.
- [20] M. Fisher, C. Dixon, and M. Peim. Clausal temporal resolution. ACM Trans. Comput. Log. 2(1): 12-56.
- [21] Thomas Arts, Stefano Tonetta: Safely Using the AUTOSAR End-to-End Protection Library. SAFECOMP 2015: 74-89
- [22] J. Barnat, L. Brim, J. Beran, and T. Kratochvila, "Partial Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs," in Submitted to FMICS., 2012.
- [23] Llorens, J., Fuentes J.M., Diaz I. RSHP: A Scheme to Classify Information in a Domain Analysis Environment. IEEE-AAAI- ICEIS 2001. International Conference on Enterprise Information Systems. Actas del congreso, pp. 686-690. Setúbal, Portugal. 2001
- [24] Llorens, Juan, Jorge Morato, and Gonzalo Genova. "RSHP: an information representation model based on relationships." In Soft computing in software engineering, pp. 221-253. Springer Berlin Heidelberg, 2004.
- [25] Génova, G., Fuentes, J.M., Llorens, J., Hurtado, O., Moreno, V. (2013). A Framework to Measure and Improve the Quality of Textual Requirements. Requirements Engineering Journal, 18(1):25-41.
- [26] Parra, E., Dimou, C., Llorens, J., Morento, V. and Fraga, A. (2015). A methodology for the classification of quality of requirements using machine learning techniques. Information and Software Technology 67:180–195.
- [27] System Quality Analyzer (SQA). The Reuse Company. Accessed February 15th, 2018. <https://www.reusecompany.com/system-quality-analyzer-sqa>
- [28] H. Ziade, R. Ayoubi, R. Velazco, "A survey on fault injection techniques", Int. Arab J. Inf. Technol., 1(2):

- 171--186. 2004.
- [29] A. Benso and D. C. S., "The art of fault injection," *Journal of Control Engineering and Applied Informatics*, pp. 9–18, 2011.
- [30] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE Trans. Softw. Eng.*, vol. 16, pp. 166–182, Feb. 1990.
- [31] A. Pena, I. Iglesias, J. Valera, and A. Martin, "Development and validation of Dynacar RT software, a new integrated solution for design of electric and hybrid vehicles," *EVS26 Los Angeles, California*, 2012.
- [32] Marco Gario, Alessandro Cimatti, Cristian Mattarei, Stefano Tonetta, Kristin Yvonne Rozier: Model Checking at Scale: Automated Air Traffic Control Design Space Exploration. *CAV (2) 2016*: 3-22
- [33] Standardized E-Gas Monitoring Concept for Gasoline and Diesel Engine Control Units V6.0, EGAS Workgroup\_en/2015-07-13.
- [34] "ISO 26262: Road vehicles Functional safety," International Organization for Standardization, Geneva, Switzerland, Standard, 2011.
- [35] Ashraf Armoush, Falk Salewski, Stefan Kowalewski, "Design Pattern Representation for Safety-Critical Embedded Systems", *J. Software Engineering & Applications*, April 2009.
- [36] C. Alexander, "A Pattern Language: Towns, Buildings, Construction," New York: Oxford University Press, 1977.
- [37] J. Barnat, P. Bauch, N. Benes, L. Brim, J. Beran, and T. Kratochvila. Analysing Sanity of Requirements for Avionics Systems. In *Formal Aspects of Computing (FAoC'16)*, volume 1, pages 1--19, 2016.
- [38] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser. DiViNE 3.0 --- An Explicit-State Model Checker for Multithreaded C&C++ Programs. *CAV*, pp 863—868, LNCS, vol. 8044, 2013.
- [39] R. Bloem, R. Cavada, I. Pill, M. Roveri, A. Tchaltsev. RAT: A Tool for the Formal Analysis of Requirements. *CAV 2007*: 263-267.
- [40] N. Soundarajan and J. O. Hallstrom. Responsibilities and rewards: Specifying design patterns. In *26th International Conference on Software Engineering*, 23-28 May 2004, Edinburgh, UK, pages 666{675. IEEE Computer Society, 2004.
- [41] MOGENTES, Model-based Generation of Tests for Dependable Embedded Systems, D3.1b: "Model-Based Generation of Test-Cases for Embedded Systems. Fault Models".
- [42] D. Domis, "Integrating fault tree analysis and component-oriented model-based design of embedded systems," *Verl. Dr. Hut*, 2011.
- [43] B. Kaiser, R. Weber, M. Oertel, E. Böde, B. Monajemi Nejad, J. Zander, "Contract-Based Design of Embedded Systems Integrating Nominal Behavior and Safety", in *Complex Systems Informatics and Modeling Quarterly CSIMQ*, 2015.
- [44] Massif: MATLAB Simulink Integration Framework for Eclipse. <https://github.com/viatra/massif>
- [45] H. L. V. de Matos, A. M. da Cunha and L. A. V. Dias, "Using design patterns for safety assessment of integrated modular avionics" *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, Colorado Springs, CO, 2014, pp. 4D1-1-4D1-13.
- [46] C. Rescher, N. Kajtazovic and C. Kreiner, "System of safety-critical embedded Architecture Patterns", in *EuroPLOP*, 2013.
- [47] L. Montecchi and B. Gallina. SafeConcert: a Metamodel for a Concerted Safety Modeling of Socio-Technical Systems. *5th International Symposium on Model-Based Safety and Assessment (IMBSA)*, Trento, Italy, September, 2017.
- [48] R. Rosner, *Modular synthesis of reactive systems*, Ph.D. thesis, Weizmann Institute of Science, 1992.
- [49] MOGENTES, Model-based Generation of Tests for Dependable Embedded Systems, <http://www.mogentes.eu>.
- [50] CRYSTAL-ARTEMIS. 2016. "CRITICAL sYSTEM Engineering AccELeration (CRYSTAL EU Project)." Accessed

September 4. <http://www.crystal-artemis.eu>.

- [51] J. M. Alvarez-Rodríguez, J. Llorens, M. Alejandres, and J. Fuentes, "OSLC-KM: A knowledge management specification for OSLC-based resources," *INCOSE Int. Symp.*, vol. 25, no. 1, pp. 16–34, 2015.
- [52] I. Sljivo, B. Gallina, J. Carlson, H. Hansson and S. Puri. "A Method to Generate Reusable Safety Case Argument-Fragments from Compositional Safety Analysis". *Journal of Systems and Software: Special Issue on Software Reuse*, July 2016.
- [53] SYNOPSIS - Safety Analysis for Predictable Software Intensive Systems.  
<http://www.es.mdh.se/SYNOPSIS>.
- [54] Alessandro Cimatti, Stefano Tonetta: A Property-Based Proof System for Contract-Based Design. *EUROMICRO-SEAA 2012*: 21-28
- [55] A. Avizienis, J.C. Laprie, B. Randell, C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. In: *IEEE Trans. Dependable Sec. Comput.*1(1): 11-33, 2004
- [56] OPENCROSS Project, Common Certification Language: Conceptual Model Deliverable D4.4, Version 1 June 2013, Version 1.1 December 2013, Version 1.2 Forthcoming Summer 2014
- [57] Irfan Sljivo. Facilitating reuse of safety case artefacts using safety contracts. Licentiate thesis. Mälardalen University, June 2015.
- [58] Foundation. Eclipse process framework (epf) composer 1.0 architecture overview.  
<http://www.eclipse.org/epf/composer/.architecture/>. Accessed: 2016-08-29
- [59] Capra – traceability management project: <https://projects.eclipse.org/projects/modeling.capra>

## Appendix A: LTL/MTL

In this appendix, we define the syntax and semantics of LTL and MTL, which are used by different architecture-drive assurance functionalities.

### LTL

LTL formulas are built with the following grammar:

$$\phi := true \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid X\phi \mid \phi_1 U \phi_2 \mid F\phi \mid G\phi$$

where  $a$  ranges over a given set of predicates  $PRED(V)$  (which can be for example the set of linear arithmetic expressions over integer and/or real variables in  $V$ ). While the semantics of a predicate is defined in terms of assignment to the variables in  $V$ , the semantics of LTL formulas is defined over sequences of assignments, also called traces. Thus, if  $s$  is an assignment to the variables in  $V$  and  $a$  is a predicate in  $PRED(V)$ , we assume to have defined if  $s(a)$  is true or false, i.e. if substituting the variables in  $a$  with the value assigned them by  $s$ , the expression is evaluated to true or to false. Then, if  $\sigma = s_0 s_1 s_2 \dots$  is an infinite sequence of assignments to variables in  $V$ , the relation  $\sigma \models \phi$  (in words,  $\sigma$  satisfies  $\phi$ ) is defined as follows:

$\sigma \models true$
$\sigma \models a$ iff $s_0(a)$ is true
$\sigma \models \phi_1 \wedge \phi_2$ iff $\sigma \models \phi_1$ and $\sigma \models \phi_2$
$\sigma \models \neg\phi_1$ iff $\sigma \not\models \phi_1$
$\sigma \models X\phi$ iff $\sigma^1 \models \phi$
$\sigma \models \phi_1 U \phi_2$ iff $\exists j \geq 0$ s.t. $\sigma^j \models \phi_2$ and $\sigma^i \models \phi_1$ , for all $0 \leq i < j$
$\sigma \models F\phi$ iff $\exists j \geq 0$ s.t. $\sigma^j \models \phi$
$\sigma \models G\phi$ iff $\forall j \geq 0$ s.t. $\sigma^j \models \phi$

where  $\sigma^h$  represents the sub-sequence  $s^h s^{h+1} \dots$  that starts from  $s^h$ .

### PLTL

PLTL is an extension of LTL introducing past operators, allowing the logic to reason over past events in a trace. For PLTL, the grammar is defined as follows:

$$\phi := true \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid X\phi \mid \phi_1 U \phi_2 \mid F\phi \mid G\phi \mid Y\phi \mid Z\phi \mid O\phi \mid H\phi \mid \phi_1 S \phi_2$$

The new operators are:

- $Y$ (esterday): the dual of  $X$ , referring to the previous instant (and false initially, at time 0);
- $Z$ : similar to  $Y$ , but true at time instant 0;
- $O$ (nce): dual of  $F$ , is true iff the argument is true in a point in the past;
- $H$ (istorically): dual of  $G$ , is true iff the argument is true in all points of the past;
- $S$ (ince): dual of  $U$ , holds if the first argument was true somewhere in the past, and the second argument was true from then up until now.

Assuming a trace  $\pi$ , and a current time instant  $i$ , the semantics of these operators are defined as follows:



$(\pi, i) \models Y\phi \text{ iff } i > 0 \wedge (\pi, i - 1) \models \phi$
$(\pi, i) \models Z\phi \text{ iff } i = 0 \vee (\pi, i - 1) \models \phi$
$(\pi, i) \models O\phi \text{ iff } \exists 0 \leq j \leq i. (\pi, j) \models \phi$
$(\pi, i) \models \phi_1 S \phi_2 \text{ iff } \exists 0 \leq j \leq i. (\pi, j) \models \phi_1 \wedge \forall j < k \leq i. (\pi, k) \models \phi_2$

## MTL

The grammar of Metric Temporal Logic (MTL) extends the LTL one by attaching time intervals to temporal operators as follows:

$$\phi := \text{true} \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \phi_1 U_I \phi_2$$

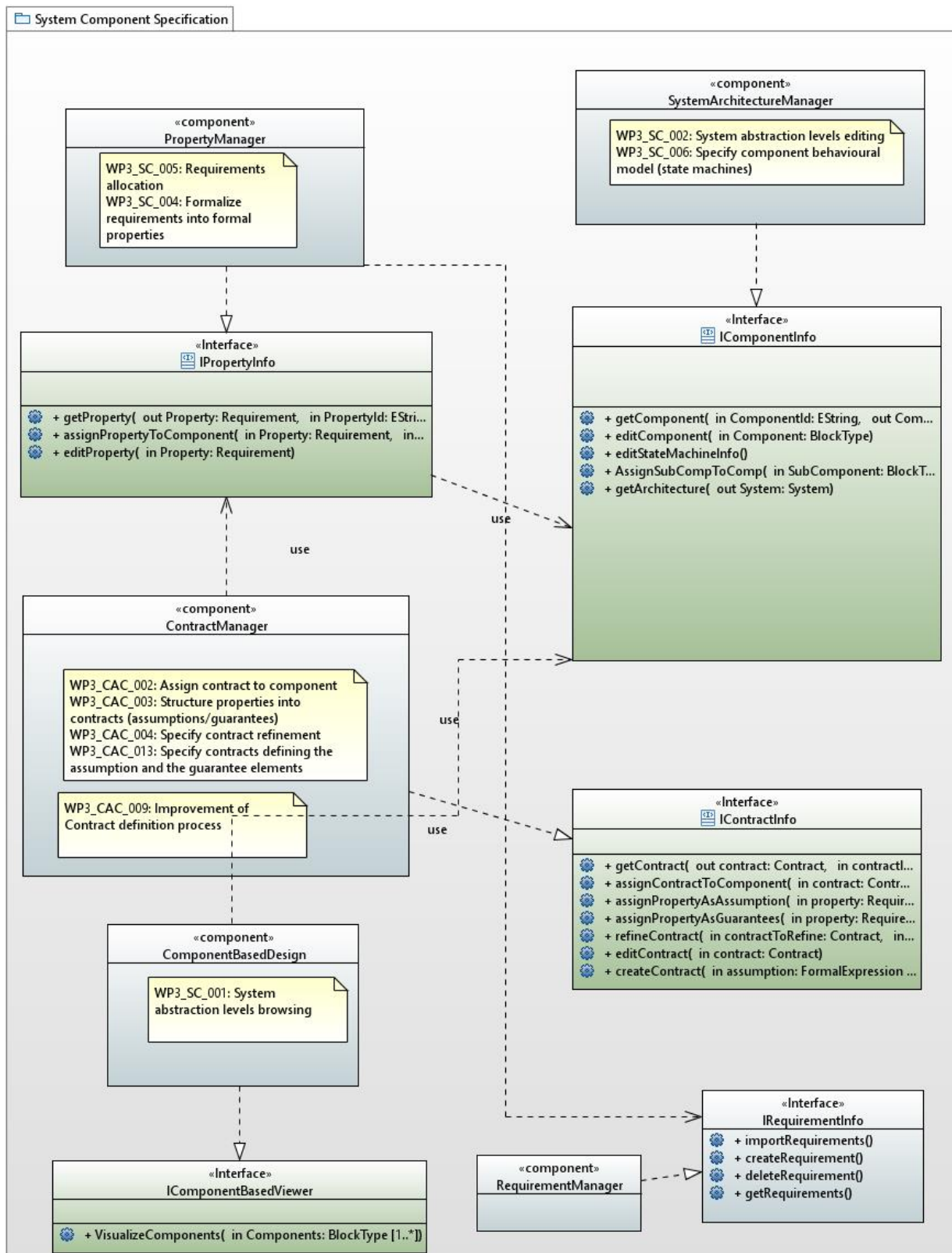
The semantics of MTL formulas is defined in terms of timed state sequences, which enrich traces with a sequence of intervals  $I_0 I_1 I_2 \dots$  cover the real axis that represents the time points. The (continuous) semantics of MTL are defined by the following satisfaction relations ( $\models$ ). If  $f: \mathbb{R}_+ \rightarrow 2^P$  is the function mapping the current time to the set of propositions that hold at that time, then the following hold (with  $f^t(s) = f(s + t)$ ):

$\sigma, t \models \text{true}$
$\sigma, t \models a \text{ iff } t \in I_i, s_i(a) \text{ is true}$
$\sigma, t \models \phi_1 \wedge \phi_2 \text{ iff } \sigma, t \models \phi_1 \text{ and } \sigma, t \models \phi_2$
$\sigma, t \models \neg\phi_1 \text{ iff } \sigma, t \not\models \phi_1$
$\sigma, t \models \phi_1 U_I \phi_2 \text{ iff } \exists t' \in I + t. \sigma, t' \models \phi_2 \wedge \forall t'' \in (t, t'). \sigma, t'' \models \phi_1$

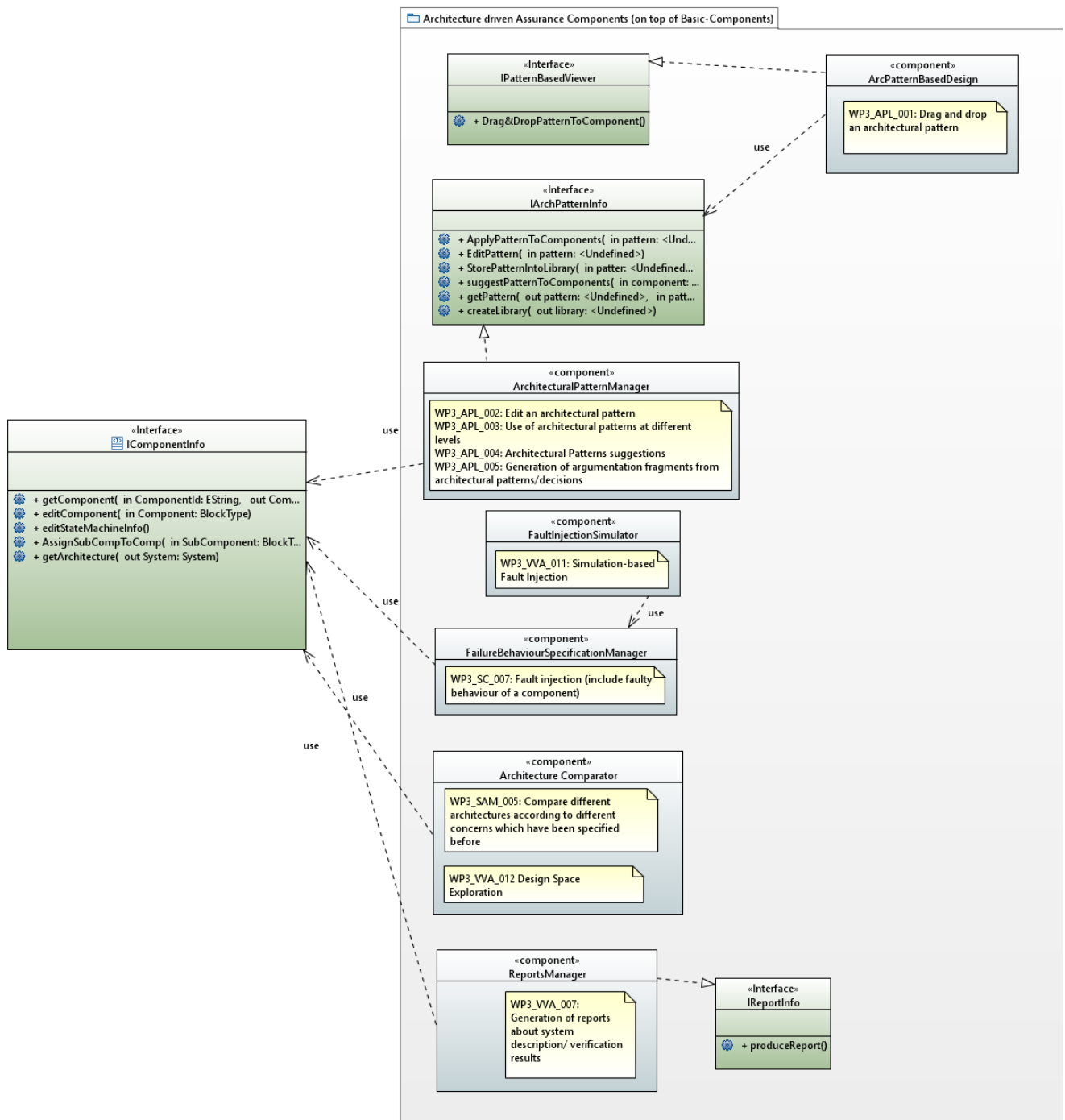
where  $I + t$  is the interval obtained from  $I$  by shifting the endpoints by  $t$ .

## **Appendix B: Architecture-driven Assurance logical architecture (\*)**

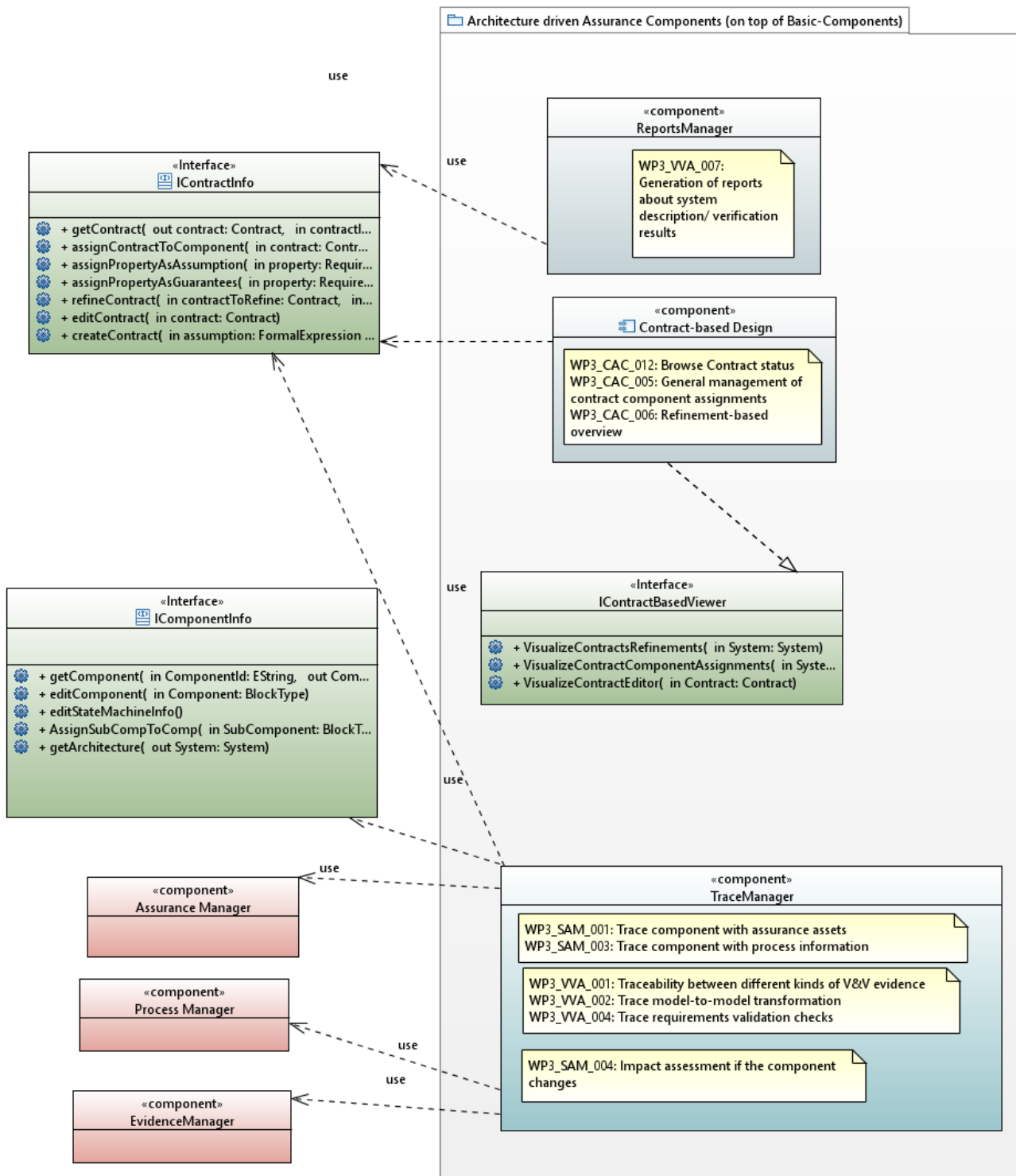
This section shows the detailed logical architecture by using UML diagrams. In the following figures, blue blocks are components, green blocks are interfaces, yellow comments contain the functional requirements and red blocks are the components related to the other AMASS technical work packages.



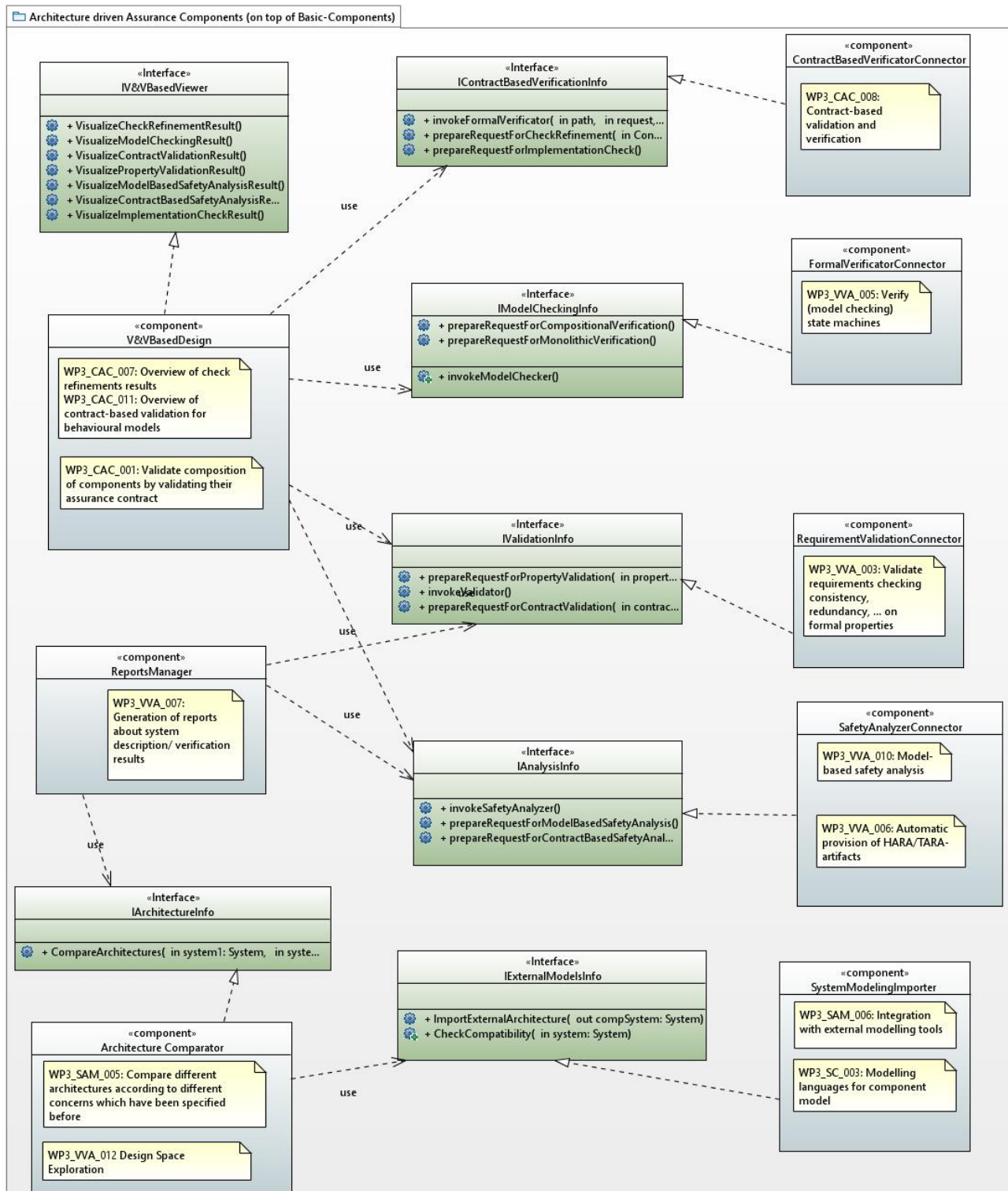
**Figure 53.** SystemComponentSpecification internal design



**Figure 54.** ArchitectureDrivenAssurance component internal structure - part1, with interface required from SystemComponentSpecification components

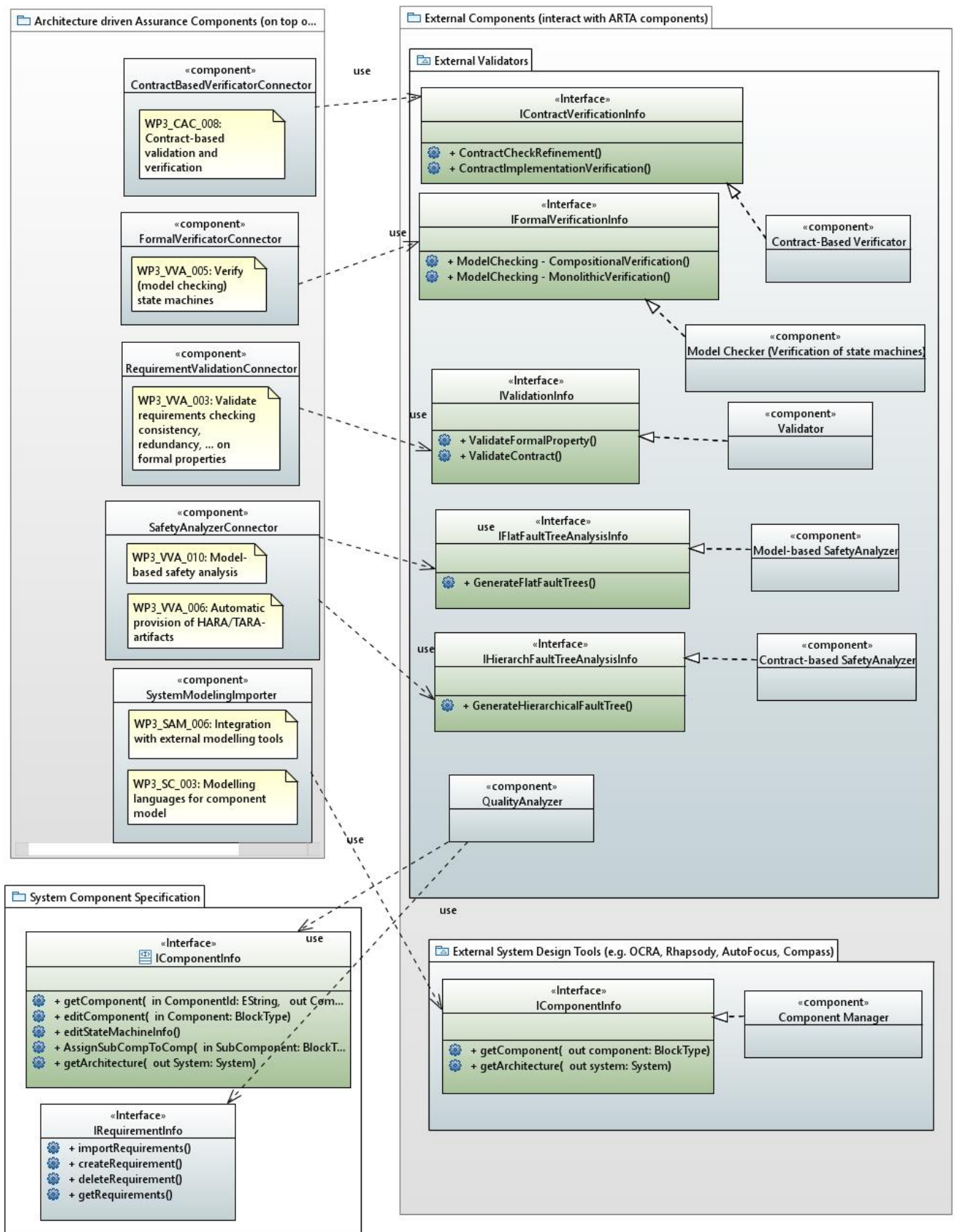


**Figure 55.** ArchitectureDrivenAssurance component internal structure – part2, with realized interface and interfaces required from SystemComponentSpecification components and from AMASS WP5 and WP6 technical work packages



**Figure 56.** ArchitectureDrivenAssurance component internal structure – part3, with realized interfaces





**Figure 57.** ArchitectureDrivenAssurance component internal structure – part4, with interfaces required from external tools



## Appendix C: SafeConcert metamodel

### Failure Modes and Criticality

Here the support for specification of FailureMode is provided; as also supported by the metamodel discussed in 2.1.1, FailureMode here are attached to Ports.

The metamodel allows specification of information related to hazards and criticality levels; in particular regarding criticality levels, the modelling of different classification according to the domain/standard of interested is supported.

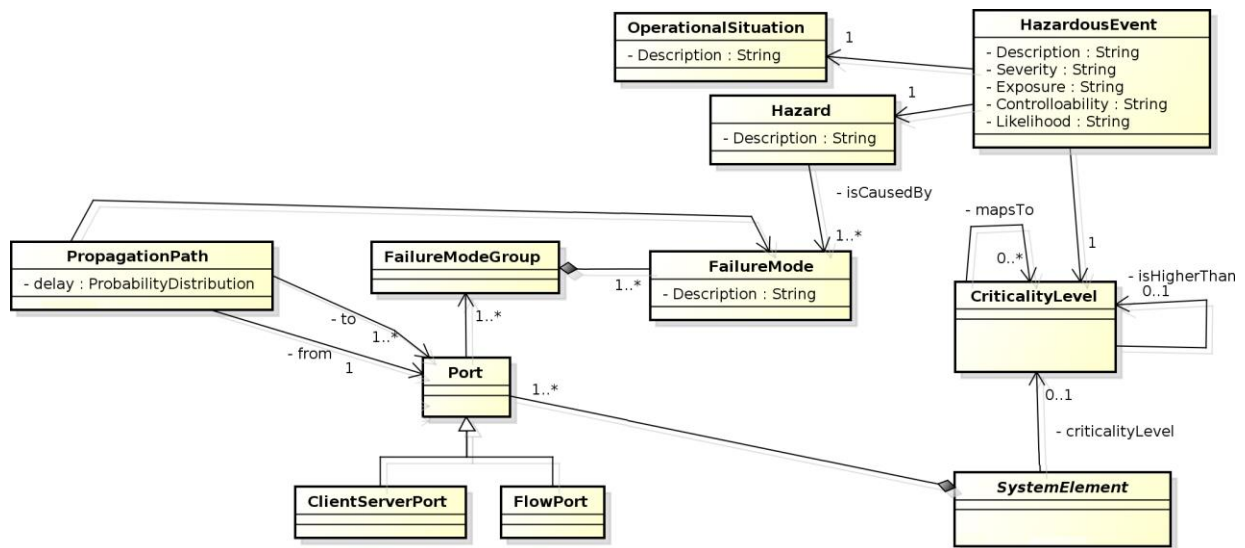


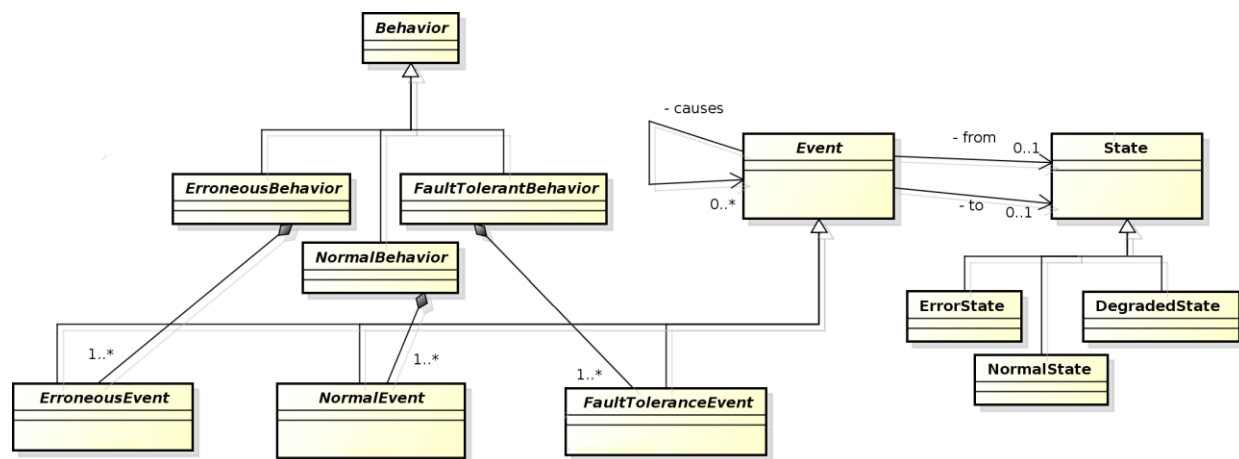
Figure 58. Failure Modes and Criticality

### Failure Behaviours

Section 2.1.1 introduces the concept of events and event occurrences associated to failures: these aspects are made available here through the notion of state machine. Moreover, here the concept of events is refined to consider different kind of events.

The failure behaviour of system elements is defined as a state machine. Then three kinds of states are considered: NormalStates, i.e., states that belong to the nominal behaviour of the component; DegradedStates, i.e., states where the service provided by the component is degraded, but still following the specifications, and ErroneousStates, i.e., states which deviate from the correct behaviour of the component.

Events are classified with respect to two dimensions: i) location, i.e., whether they are InputEvents, InternalEvents, or OutputEvents for the component, and ii) type of behaviour, i.e., whether they are NormalEvents, ErroneousEvents, or FaultToleranceEvents.



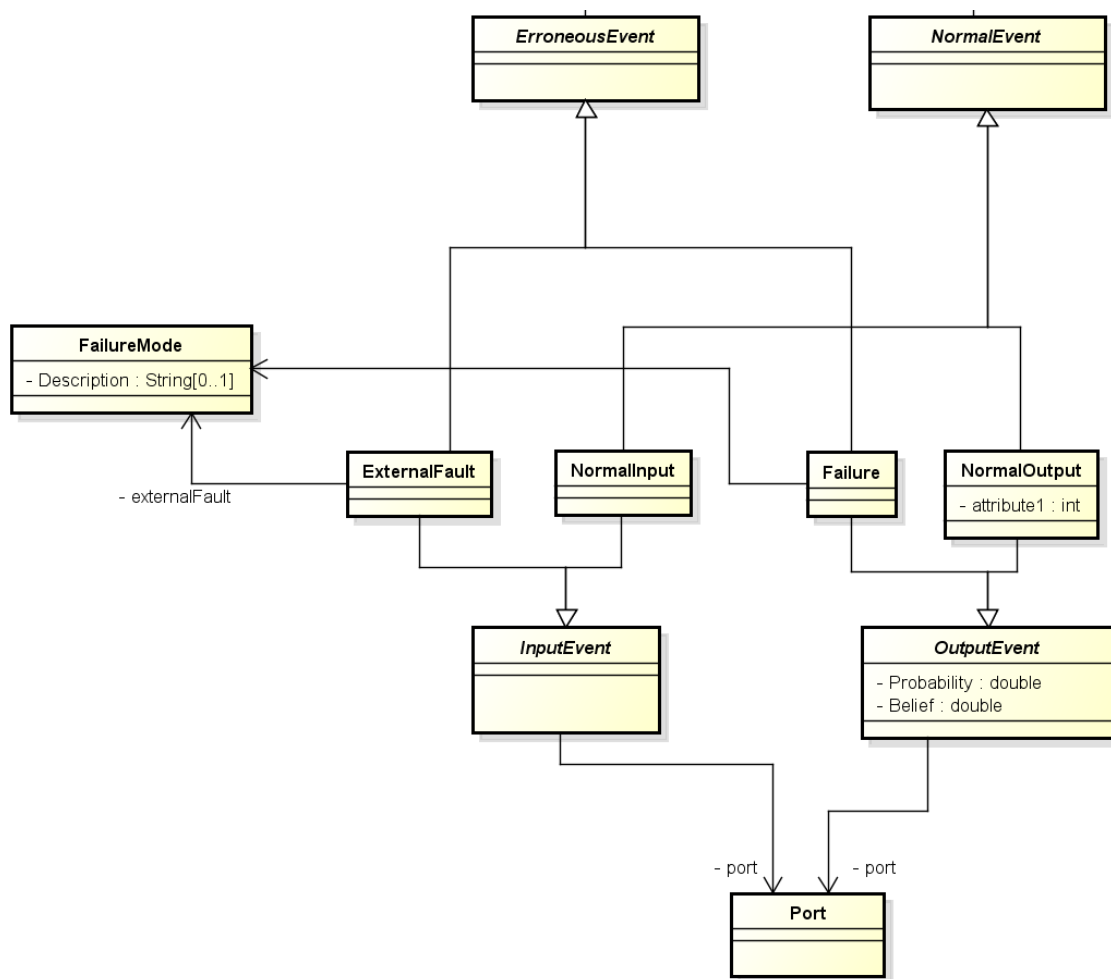
**Figure 59.** Failure Behaviours

## Input and Output Events

External events are either InputEvents or OutputEvents. In both cases, they are events that are occurring on the ports of a system element.

There are essentially two kinds of events that may occur as an input event: a NormalInput event, meaning that the component has received a normal input on one of its ports, or an ExternalFault, meaning that the component has received an input that deviates from the specification, i.e., the service it receives from another entity of the system is not correct.

Similarly, output events are also of two different kinds: NormalOutput, i.e., the component provides a normal output (i.e., a correct service) on the involved port, or Failure, meaning that the service provided by the component on the involved port has become incorrect.

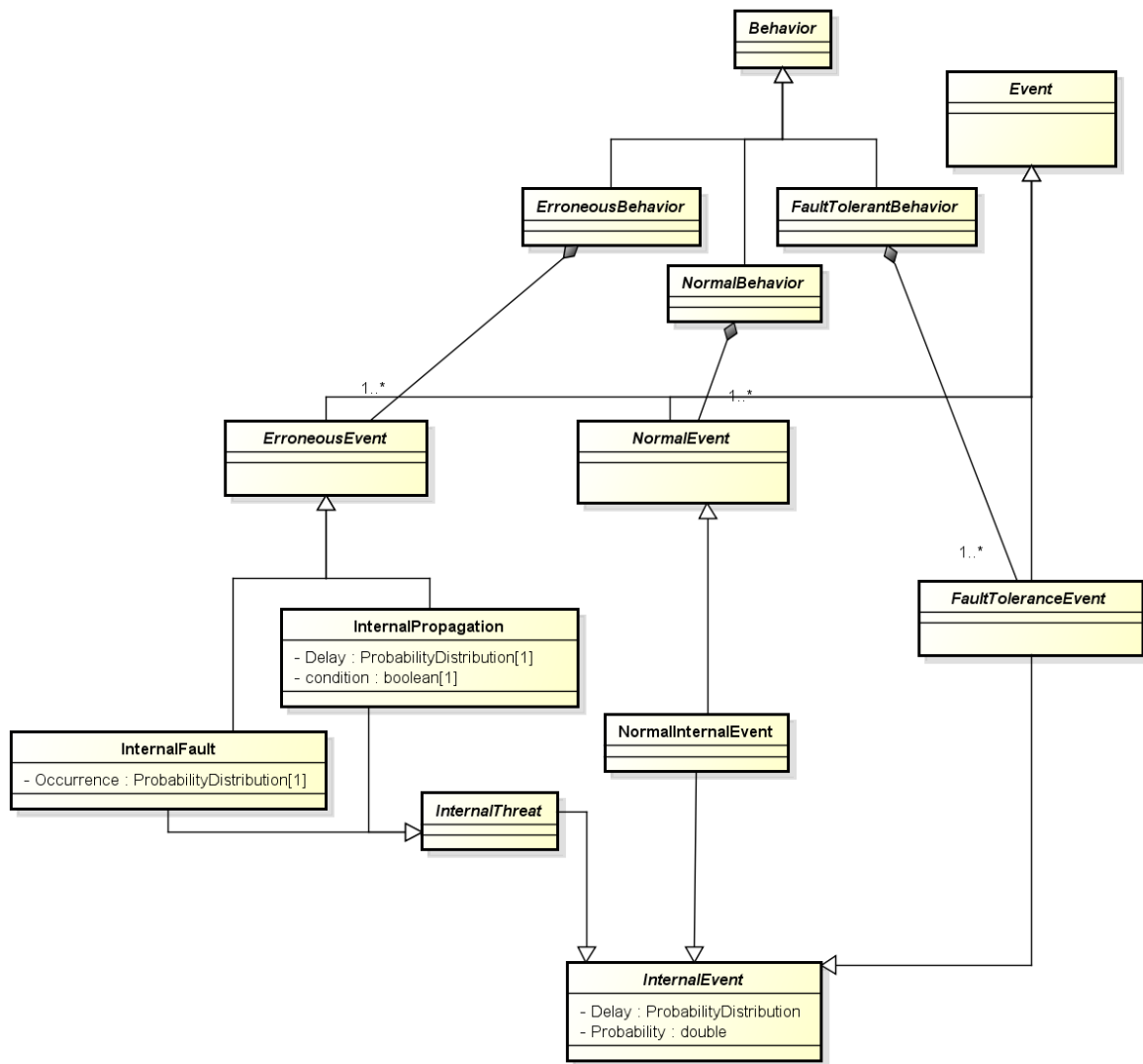


**Figure 60.** Input and Output Events

## Internal Events

Internal events are those events that are internal to the block. We distinguish essentially between three macro-categories of internal events: the NormalInternalEvent, the InternalThreats, and the FaultToleranceEvents.

An internal event may have associated a delay, expressed by a probability distribution, and a probability, which specifies the relative probability of occurrence the event, in case others are supposed to occur at the same time.



**Figure 61.** Internal Events

A **NormalInternalEvent** is an event that was foreseen by the specification of the component, e.g., a switch to a “power saving” mode due to a low battery level. For the purpose of safety analysis, it is important to take into account such aspects of a component’s behaviour: the occurrence of failures, as well as their effects and criticality, may depend, for example, on the current system operational mode (e.g., take off, cruise, and landing for an aircraft).

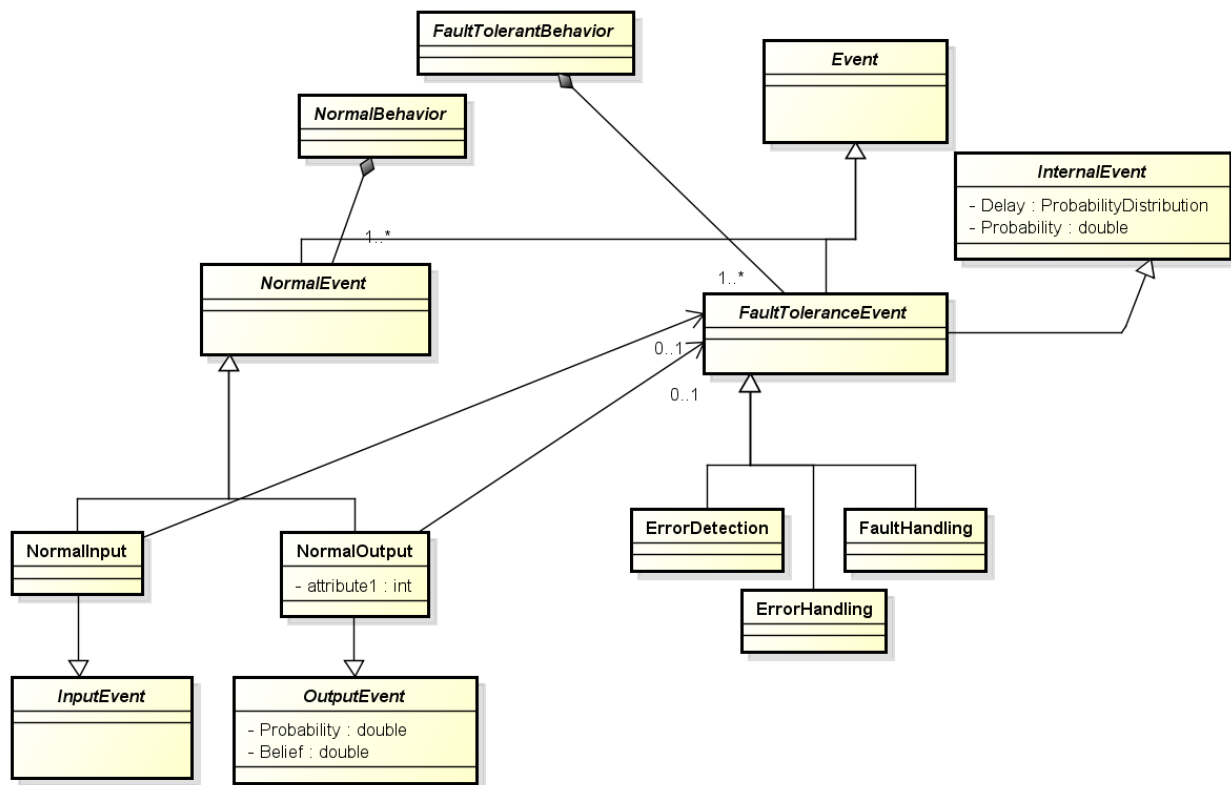
An **InternalThreat** is essentially an **InternalFault**, or an **InternalPropagation**. An **InternalFault** element represents a fault that occurs spontaneously within the component, e.g., an electrical fault, or that is pre-existing and dormant [55] within the component, e.g., a software fault. The occurrence attribute can be used to specify a probabilistic delay, after which the fault manifests itself in the state of the component, i.e., after which the fault gets activated [55].

The **InternalPropagation** concept serves to the purpose of defining how input events, or combination of thereof, affect the internal state of the component. The condition that triggers the propagation is specified by the condition attribute. This attribute is essentially a Boolean expression over **InputEvent** elements, which specifies which combination of events on the ports of the involved component cause that particular internal propagation event.

## Fault-tolerance events

A particular kind of internal events are those events constituting the fault-tolerance behaviour of system components. Three different events of this kind are considered. The classical taxonomy of dependable computing essentially classifies fault tolerance techniques in three main groups: error detection, error handling, and fault handling.

This can be related with what discussed in Section 2.1.1 about Safety Measures.



**Figure 62.** Fault-tolerance events

The *ErrorDetection* event represents the detection of an error in the state of the component, to which different actions may follow. Those actions can be defined by either associating a state transition with the event (e.g., to a safe state), or by specifying that additional events are triggered by the error detection event (e.g., reconfiguration events).

The *ErrorHandling* event represents an event for which an existing error in the state of the component is eliminated, thus bringing the component to an error-free state. Depending on the actual technique, the state can be a previous state (rollback), or a new state (roll forward). In some cases, the error can also be corrected due to redundancy in the component state (compensation).

Fault handling (*FaultHandling* event) consists in preventing existing faults for being reactivated again. For this purpose, several techniques can be adopted, including for example fault isolation (faulty components are excluded from the service delivery) and reconfiguration (the system configuration is modified e.g., switching in spare components).

## Relation to Contracts

Section 2.1.1 addresses the possibility to model traceability between failure modes and contracts; here we have failure modes traced to hazards, and the possibility to model requirements and formal properties traceability. By tracing safety requirement to hazard then traceability between contract and failure modes can be derived.

## Appendix D: Design patterns for fault tolerance applied to technology according to ISO 26262

Safety Mechanism/Measure	Purpose/objective of the Safety Mechanism/Measure	Typical Diagnostic Coverage considered achievable (PatternGuarantee)	Notes (PatternAssumption)
<b>Systems</b>			
Failure Detection by on-line monitoring	To detect failures by monitoring the behaviour of the system in response to the normal (on-line) operation	Low	Depends on diagnostic coverage of failure detection
Comparator	To detect, as early as possible (non-simultaneous) failures in independent hardware or software	High	Depends on the quality of the comparison
Majority voter	To detect and mask failures in one of at least three channels	High	Depends on the quality of the voting
Dynamic principles	To detect static failures by dynamic signal processing	Medium	Depends on diagnostic coverage of failure detection
Analogue signal monitoring in preference to digital on/off states	To improve confidence in measured signals	Low	————
Self-test by software cross exchange between two independent units	To detect, as early as possible, failures in the processing unit consisting of physical storage (for example registers) and functional units (for example, instruction decoder).	Medium	Depends on the quality of the self test
<b>Electrical elements</b>			
Failure detection by on-line monitoring	To detect failures by monitoring the behaviour of the system in response to the normal (on-line) operation	High	Depends on diagnostic coverage of failure detection
<b>Processing units</b>			
Self-test by software: limited number of patterns (one channel)	To detect, as early as possible, failures in the processing unit and other sub-elements, using special hardware that increases the speed and extends the scope of failure detection.	Medium	Depends on the quality of the self test
Self-test by software cross exchange between two independent units	To detect, as early as possible, failures in the processing unit, by dynamic software comparison.	Medium	Depends of the quality of the self test
Self-test supported by hardware (one-channel)	To detect, as early as possible, failures in the processing unit and other sub-elements, using special hardware that increases the speed and extends the scope of failure detection.	Medium	Depends on the quality of the self test

Software diversified redundancy (one hardware channel)	The design consists of two redundant diverse software implementations in one hardware channel. In some cases, using different hardware resources (e.g. different RAM, ROM memory ranges) can increase the diagnostic coverage.	High	Depends on the quality of the diversification. Common mode failures can reduce diagnostic coverage
Reciprocal comparison by software	To detect, as early as possible, failures in the processing unit, by dynamic software comparison.	High	Depends on the quality of the comparison
HW redundancy (e.g. Dual Core Lockstep, asymmetric redundancy, coded processing)	To detect, as early as possible, failures in the processing unit, by step-by-step comparison of internal or external results or both produced by two processing units operating in lockstep.	High	It depends on the quality of redundancy. Common mode failures can reduce diagnostic coverage
Configuration Register Test	To detect, as early as possible, failures in the configuration registers of a processing unit. Failures can be hardware related (stuck values or soft errors induced bit flips) or software related (incorrect value stored or register corrupted by software error).	High	Configuration registers only
Stack over/under flow Detection	To detect, as early as possible, stack over or under flows	Low	Stack boundary test only
<b>Non-volatile memory</b>			
Parity bit	To detect a single corrupted bit or an odd number of corrupted bits failures in a word (typically 8 bits, 16 bits, 32 bits, 64 bits or 128 bits).	Low	—
Memory monitoring using error-detection-correction codes (EDC)	To detect each single-bit failure, each two-bit failure, some three-bit failures, and some all-bit failures in a word (typically 32, 64 or 128 bits).	High	The effectiveness depends on the number of redundant bits. Can be used to correct errors
Modified checksum	To detect each single bit failure.	Low	Depends on the number and location of bit errors within test area
Memory Signature	To detect each one-bit failure and most multi-bit failures.	High	—
Block replication	To detect each bit failure.	High	—
<b>Volatile memory</b>			



RAM pattern test	To detect predominantly static bit failures.	Medium	High coverage for stuck-at failures. No coverage for linked failures. Can be appropriate to run under interrupt protection
RAM March test	To detect predominantly persistent bit failures, bit transition failures, addressing failures and linked cell failures.	High	Depends on the write read order for linked cell coverage. Test generally not appropriate for run time
Parity bit	To detect a single corrupted bit or an odd number of corrupted bits failures in a word (typically 8 bits, 16 bits, 32 bits, 64 bits or 128 bits).	Low	—
Memory monitoring using error-detection-correction codes (EDC)	To detect each single-bit failure, each two-bit failure, some three-bit failures, and some all-bit failures in a word (typically 32, 64 or 128 bits).	High	The effectiveness depends on the number of redundant bits. Can be used to correct errors
Block replication	To detect each bit failure.	High	Common failure modes can reduce diagnostic coverage
Running checksum/CRC	To detect single bit, and some multiple bit, failures in RAM.	High	The effectiveness of the signature depends on the polynomial in relation to the block length of the information to be protected. Care needs to be taken so that values used to determine checksum are not changed during checksum calculation Probability is 1/maximum value of checksum if random pattern is returned
<b>Analogue and digital I/O</b>			
Failure detection by on-line monitoring (Digital I/O)	To detect failures by monitoring the behaviour of the system in response to the normal (on-line) operation.	Low	Depends on diagnostic coverage of failure detection
Test pattern	To detect static failures (stuck-at failures) and cross-talk.	High	Depends on type of pattern
Code protection for digital I/O	To detect random hardware and systematic failures in the input/output dataflow.	Medium	Depends on type of coding
Multi-channel parallel output	To detect random hardware failures (stuck-at failures), failures caused by external influences, timing failures, addressing failures, drift failures and transient failures.	High	—
Monitored outputs	To detect individual failures,	High	Only if dataflow changes

	failures caused by external influences, timing failures, addressing failures, drift failures (for analogue signals) and transient failures.		within diagnostic test interval
Input comparison/voting (1oo2, 2oo3 or better redundancy)	To detect individual failures, failures caused by external influences, timing failures, addressing failures, drift failures (for analogue signals) and transient failures.	High	Only if dataflow changes within diagnostic test interval
<b>Communication bus (serial, parallel)</b>			
One-bit hardware redundancy	To detect each odd-bit failure, i.e. 50 % of all the possible bit failures in the data stream.	Low	—
Multi-bit hardware redundancy	To detect failures during the communication on a bus and in serial transmission links.	Medium	—
Read back of sent message	To detect failures in bus communication.	Medium	—
Complete hardware redundancy	To detect failures during the communication by comparing the signals on two buses.	High	Common mode failures can reduce diagnostic coverage
Inspection using test patterns	To detect static failures (stuck-at failure) and cross-talk.	High	—
Transmission redundancy	To detect transient failures in bus communication.	Medium	Depends on type of redundancy. Effective only against transient faults
Information redundancy	To detect failures in bus communication.	Medium	Depends on type of redundancy
Frame counter	To detect frame losses. A frame is a coherent set of data sent from one controller to other controller(s). The unique frame is identified by a message ID.	Medium	—
Timeout monitoring	To detect loss of data between the sending node and the receiving node.	Medium	—
Combination of information redundancy, frame counter and timeout monitoring		High	For systems without hardware redundancy or test patterns, high coverage can be claimed for the combination of these safety mechanisms

Power supply			
Voltage or current control (input)	To detect as soon as possible wrong behaviour of input current or voltage values.	Low	—
Voltage or current control (output)	To detect as soon as possible wrong behaviour of output current or voltage values.	High	—
Program sequence monitoring/Clock			
Watchdog with separate time base without time-window	To monitor the behaviour and the plausibility of the program sequence.	Low	—
Watchdog with separate time base and time-window	To monitor the behaviour and the plausibility of the program sequence.	Medium	Depends on time restriction for the time-window
Logical monitoring of program sequence	To monitor the correct sequence of the individual program sections.	Medium	Only effective against clock failures if external temporal events influence the logical program flow. Provides coverage for internal hardware failures (such as interrupt frequency errors) that can cause the software to run out of sequence
Combination of temporal and logical monitoring of program sequence	To monitor the behaviour and the correct sequence of the individual program sections.	High	—
Combination of temporal and logical monitoring of program sequences with time dependency	To monitor the behaviour, correct sequencing and the execution time interval of the individual program sections.	High	Provides coverage for internal hardware failures that can cause the software to run out of sequence.  When implemented with asymmetrical designs, provides coverage regarding communication sequence between main and monitoring device  NOTE Method to be designed to account for execution jitter from interrupts, CPU loading, etc.
Sensors			
Failure detection by on-line monitoring	To detect failures by monitoring the behaviour of the system in response to the normal (on-line) operation.	Low	Depends on diagnostic coverage of failure detection

Test pattern	To detect static failures (stuck-at failures) and cross-talk.	High	—
Input comparison/voting (1oo2, 2oo3 or better redundancy)	To detect individual failures, failures caused by external influences, timing failures, addressing failures, drift failures (for analogue signals) and transient failures.	High	Only if dataflow changes within diagnostic test interval
Sensor valid range	To detect sensor shorts to ground or power and some open circuits.	Low	Detects shorts to ground or power and some open circuits
Sensor correlation	To detect sensor-in-range drifts, offsets or other errors using a redundant sensor.	High	Detects in range failures
Sensor rationality check	To detect sensor-in-range drifts, offsets or other errors using multiple diverse sensors.	Medium	—
<b>Actuators</b>			
Failure detection by on-line monitoring	To detect failures by monitoring the behaviour of the system in response to the normal (on-line) operation.	Low	Depends on diagnostic coverage of failure detection
Test pattern	To detect static failures (stuck-at failures) and cross-talk.	High	—
Monitoring (i.e. coherence control)	To detect the incorrect operation of an actuator.	High	Depends on diagnostic coverage of failure detection
<b>Combinatorial and sequential logic</b>			
Self-test by software	To detect, as early as possible, failures in the processing unit and other sub-elements consisting of physical storage (for example, registers) or functional units (for example, instruction decoder or an EDC coder/decoder), or both, by means of software.	Medium	—
Self-test supported by hardware (one-channel)	To detect, as early as possible, failures in the processing unit and other sub-elements, using special hardware that increases the speed and extends the scope of failure detection.	High	Effectiveness depends on the type of self-test. Gate level is an appropriate level for this test
<b>On-chip communication</b>			
One-bit hardware redundancy	To detect each odd-bit failure, i.e. 50 % of all the possible bit failures in the data stream.	Low	—

Multi-bit hardware redundancy	To detect failures during the communication on a bus and in serial transmission links.	Medium	Multi-bit redundancy can achieve high coverage by proper interleaving of data, address and control lines, and if combined with some complete redundancy, e.g. for the arbiter.
Complete hardware redundancy	To detect failures during the communication by comparing the signals on two buses.	High	Common failure modes can reduce diagnostic coverage
Test pattern	To detect static failures (stuck-at failures) and cross-talk.	High	Depends on type of pattern

## Appendix E: Massif Metamodel

Massif [44] is an Eclipse feature that allows to import and export capabilities of MATLAB Simulink models to/from eclipse EMF models. Massif is used in the context of the simulation-based fault infection framework implementation described in section 2.4.7.

The EMF metamodel used by Massif to represent MATLAB Simulink models is showed in Figure 63.

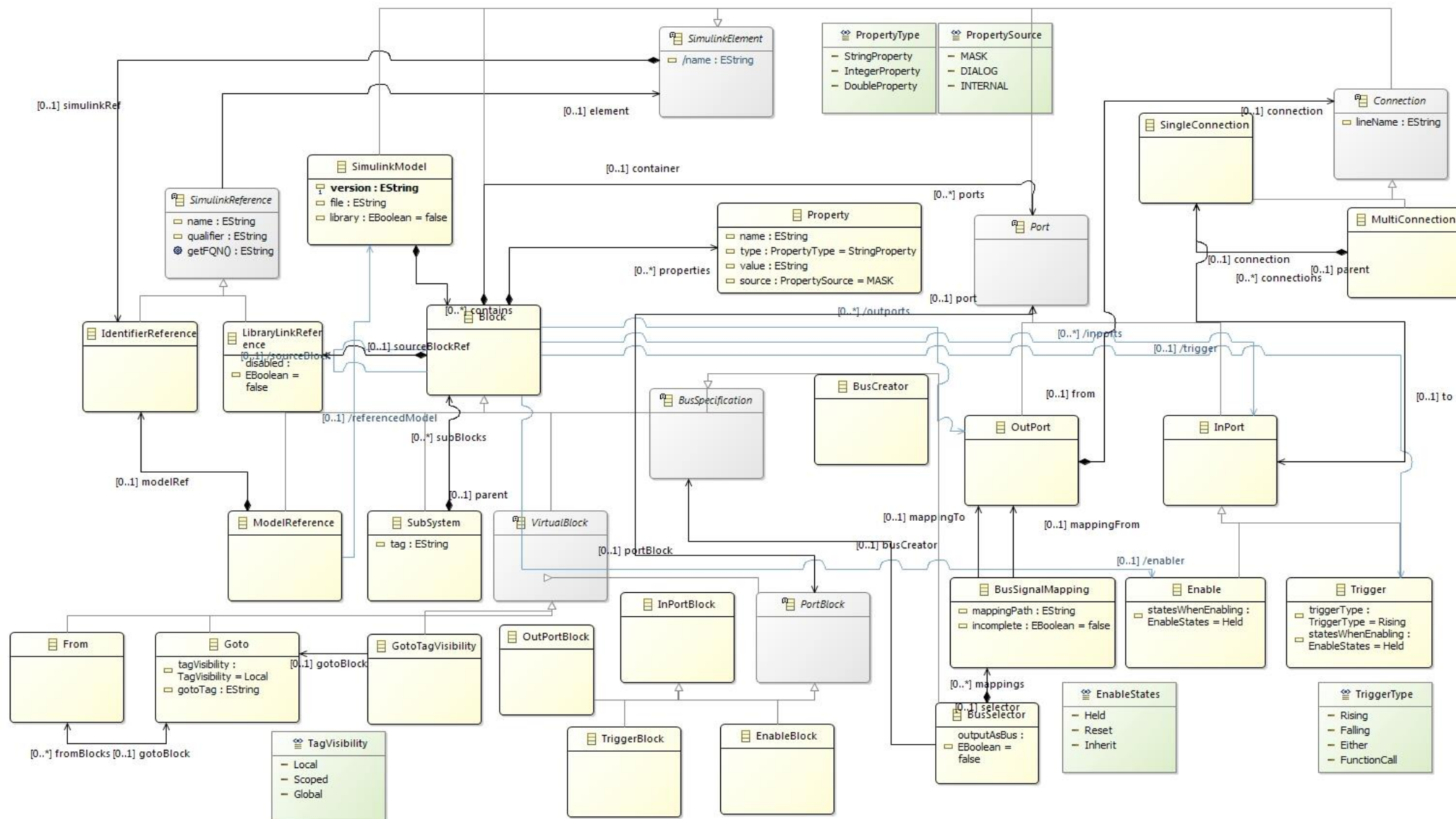


Figure 63. Massif meta-model.



## Appendix F: Document changes respect to D3.2 (\*)

New Sections:

Section	Title
2.2.2	Parametrized architectures for architectural patterns
2.4.4.4	Metrics checklists
2.4.7.1	Sabotage architecture
2.4.8.1	Employment in AMASS
2.5	Assurance Patterns for Contract-Based Design

Modified Sections (in the Section column, sections applicable to D3.2 only have “D3.2-” as prefix):

Section	Title	Change
	Executive Summary	Updated
1	Introduction	Updated
2.1.2	Tracing CACM with results from external safety analysis tools	Clarifications provided. Added information about Capra traceability metamodel
D3.2-2.2.2	Application of Architectural Patterns	Removed
D3.2-2.2.3	Assurance of Architectural Fault Tolerant Patterns	Moved as section 2.5.1
2.3.2	Reuse of Component	The focus on library definition has been removed. Library of components, as general concept, is supported by the AMASS approach and prototype; libraries of specific components can be defined according to specific needs.
2.3.3	Contract-Based Assurance Argument Generation	Minor changes
2.4	Activities Supporting Assurance Case	Chapter renamed
2.4.3.5	Checking Realisability of Requirements	Updated
2.4.4.3	Metrics for models	New correctness metrics provided
2.4.6	Design Space Exploration	Minor restructuring. D3.2 sections 2.4.6.1 and 2.4.6.2 have been merged as 2.4.6 text. Added section 2.4.6.1 Employment in AMASS
2.4.7	Simulation-Based Fault Injection Framework	Updated
D3.2 - 2.4.7.1	Employment in AMASS	Updated Moved as section 2.4.7.2
3.1	Functional Architecture for Architecture Driven Assurance	Functional Logical Architecture has been updated. Coverage of P2 prototype
3.2	System Component Metamodel for Architecture-driven Assurance	Extended The CMMA metamodel presented in D3.2 has been updated. The concept of Pattern definition and usage have been added.
4	Way forward for the implementation	Updated requirements coverage.
5	Conclusions	Updated



Appendix B	Architecture-driven Assurance logical architecture	Functional Logical Architecture has been updated. Coverage of P2 prototype
------------	--	---