

## ECSEL Research and Innovation actions (RIA)



# AMASS

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

# Methodological Guide for Architecture-Driven Assurance (b) D3.8

<b>Work Package:</b>	WP3: Architecture-Driven Assurance
<b>Dissemination level:</b>	PU = Public
<b>Status:</b>	Final
<b>Date:</b>	31 <sup>st</sup> October 2018
<b>Responsible partner:</b>	Stefano Tonetta (FBK)
<b>Contact information:</b>	tonettas@fbk.eu
<b>Document reference:</b>	AMASS_D3.8_WP3_FBK_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.

---

## Contributors

Names	Organisation
Pietro Braghieri, Alberto Debiasi, Ramiro Demasi, Stefano Tonetta, Luca Cristoforetti	Fondazione Bruno Kessler (FBK)
Eugenio Parra	Universidad Carlos III de Madrid (UC3)
Luis Alonso	The Reuse Company (TRC)
Dejan Nickovic	AIT Austrian Institute of Technology (AIT)
Estibaliz Amparan, Garazi Juez	Tecnalia Research & Innovation (TEC)
Irfan Sljivo	Mälardalen University (MDH)
Stefano Puri	Intecs (INT)
Tomáš Kratochvíla, Ivana Cerna, Vit Koksa	Honeywell (HON)
Andrea Critelli, Luca Macchi	RINA Services (RIN)
Markus Grabowski	Assystem Germany (formerly: Berner&Mattner Systemtechnik) (B&M)
Bernhard Kaiser	ANSYS medini Technologies AG (formerly: KPIT medini Technologies AG) (KMT)
Morayo Adedjouma, Ansgar Radermacher, Huascar Espinoza, Bernard Botella, Thibaud Antignac,	Commissariat à l’Energie Atomique et aux Energies Alternatives (CEA)

## Reviewers

Names	Organisation
Gabriel Pedroza (Peer reviewer D3.7)	Commissariat à l’Energie Atomique et aux Energies Alternatives (CEA)
Bernhard Winkler (Peer reviewer D3.7)	Virtual Vehicle (VIF)
Elena Alaña Salazar (Peer reviewer D3.8)	GMV (GMV)
Daniele Tornaghi (Peer reviewer D3.8)	Thales Italy (THI)
Cristina Martinez (Quality Manager D3.7 and D3.8)	Tecnalia Research & Innovation (TEC)
Jose Luis de la Vara (TC reviewer D3.7 and D3.8)	Universidad Carlos III de Madrid (UC3)
Barbara Gallina (TC reviewer D3.7 and D3.8)	Maelardalen Hoegskola (MDH)
Alejandra Ruiz (TC reviewer D3.7 and D3.8)	Tecnalia Research & Innovation (TEC)
Isaac Moreno (TC reviewer D3.8)	Thales Alenia Space Spain (TAS)

# TABLE OF CONTENTS

<b>Executive Summary.....</b>	<b>10</b>
<b>1. Introduction.....</b>	<b>11</b>
<b>2. Architecture-Driven Assurance Overview .....</b>	<b>13</b>
2.1 Background.....	13
2.1.1 System Architecture .....	13
2.1.2 Contract-Based Design .....	13
2.1.3 Semi-formal Specification of Requirements and Contracts .....	14
2.1.4 Requirements Validation.....	15
2.1.5 Verification and Validation of Behavioural Models .....	16
2.1.6 Model-Based Safety Analysis.....	16
2.1.7 Safety Case .....	18
2.2 Architecture-Driven Assurance.....	20
2.2.1 Main Idea.....	20
2.2.2 Transfer of the AMASS Main Idea to an Industry-Proof Working Process.....	22
2.2.3 Tool Support Overview.....	28
<b>3. Methodological Guide .....</b>	<b>30</b>
3.1 Workflow Overview .....	30
3.2 System Design.....	36
3.2.1 System Definition.....	36
3.2.2 Hazard Identification.....	39
3.2.3 Requirements Formalization .....	42
3.2.4 Requirements Early Validation .....	51
3.2.5 Functional Refinement.....	77
3.2.6 Component's Nominal and Faulty Behaviour Definition.....	86
3.2.7 Functional Early Verification.....	87
3.3 Safety Analysis .....	94
3.3.1 Simulation-based Fault Injection .....	94
3.3.2 Model-Based Safety Analysis.....	102
3.3.3 Contract-Based Safety Analysis .....	104
3.4 Safety Case .....	105
3.4.1 Evidence Generation .....	105
3.4.2 Link to Architectural Entities .....	107
3.4.3 Document Generation.....	107
3.4.4 Argument Fragments Generation .....	109
3.5 Summary .....	114
<b>4. Case Studies.....</b>	<b>116</b>
4.1 AIR6110 .....	116
4.1.1 Case Study Description.....	116
4.2 ETCS Linking function .....	119
4.2.1 Case Study Description.....	119
4.2.2 System Definition.....	121
4.2.3 Requirements Formalization .....	125
4.2.4 Parametrization of the Architecture .....	127



4.2.5 Error Parametrization.....	129
4.2.6 Result Analysis .....	129
4.2.7 Evidence Generation .....	131
4.3 DC Motor Drive .....	132
4.3.1 Case Study Description.....	132
4.3.2 Experience gained with DC Drive Case Study at the stage of AMASS P2 .....	135
<b>5. Conclusions.....</b>	<b>138</b>
<b>Abbreviations and Definitions.....</b>	<b>139</b>
<b>References .....</b>	<b>141</b>
<b>Appendix A: Document changes respect to D3.7 .....</b>	<b>145</b>



## List of Figures

Figure 1.	Architecture-Driven Assurance in relation to the other work packages.....	11
Figure 2.	MBSA techniques for early safety assessment.....	17
Figure 3.	Safety arguments show the fulfilment of safety goals and other related safety requirements by providing evidences (e.g., ISO 26262 work products) .....	18
Figure 4.	Links between the safety case and the design model .....	20
Figure 5.	Modelling and Early V&V providing input to the assurance case.....	21
Figure 6.	Generic ISO 26262 V-model .....	22
Figure 7.	Zoom-in on left leg of the V-model.....	23
Figure 8.	Workflow in the left V-leg with normal function development and safety activities.....	24
Figure 9.	Dimensions of Refinement .....	25
Figure 10.	Early vs. late formalization .....	26
Figure 11.	V-Model with Monitor Generation, Fault Injection and Simulation.....	27
Figure 12.	Internal and external tools involved in the Architecture-Driven Assurance .....	29
Figure 13.	Main Steps to perform the design and development of the system with the support of the AMASS platform for Architecture-Driven Assurance.....	31
Figure 14.	Steps for the System Design.....	32
Figure 15.	Detailed steps for the System Design with the definition of roles (I) .....	33
Figure 16.	Detailed steps for the System Design with the definition of roles (II) .....	34
Figure 17.	Sub-activities related to the safety analysis currently supported by the AMASS platform .....	35
Figure 18.	Detailed Safety Analysis Process.....	35
Figure 19.	Sub-activities related to the safety case currently supported by the AMASS platform .....	36
Figure 20.	A generic component and the system component created in the Block Definition Diagram..	38
Figure 21.	ISO 26262 HARA Process Flow.....	40
Figure 22.	Example of system description in SysML .....	40
Figure 23.	Example of HAZOP Analysis.....	41
Figure 24.	Example of scenario analysis .....	42
Figure 25.	Formal property created in the diagram editor. The property is associated to one component .....	43
Figure 26.	Formal property created in the diagram editor. The property is not associated to any component .....	44
Figure 27.	Owner component set in the UML tab .....	44
Figure 28.	Property Editor with “completion assistance” .....	45
Figure 29.	Requirements set in the Profile Tab .....	45
Figure 30.	Popup related to the contract definition process.....	46
Figure 31.	Contract editor with “completion assistance” .....	46
Figure 32.	Assertion section of the properties view in SAVONA .....	47
Figure 33.	Assertion-Wizard: Selection of a General Pattern Type to formulate an assertion .....	48
Figure 34.	Assertion-Wizard: Refine the pattern instance with names of available model elements.....	49
Figure 35.	Pattern-suggestion feature of the Assertion Editor.....	49
Figure 36.	Macros Section of the Properties View in SAVONA.....	50
Figure 37.	Data Dictionary View in SAVONA.....	50

Figure 38.	Formal assumptions/guarantees in the contract of a component .....	51
Figure 39.	Invocation of the V&V Manager .....	52
Figure 40.	V&V Result view.....	53
Figure 41.	RQA Connection Window.....	54
Figure 42.	OSLC-KM Connection (SysML CHESS sub-type) .....	55
Figure 43.	RQA Connection Window.....	56
Figure 44.	Information message to select a set of metrics .....	56
Figure 45.	Window with the templates store in RQA that contains the set of metrics .....	57
Figure 46.	RQA ready to assess the quality .....	57
Figure 47.	Detail of the assessment options.....	58
Figure 48.	Quality information of the model in RQA. ....	58
Figure 49.	Connection window to export the evidence in an AMASS repository.....	59
Figure 50.	Information message of the stored process.....	59
Figure 51.	Evidence stored in the assurance project .....	60
Figure 52.	Creation of a new custom-coded consistency metric.....	61
Figure 53.	Custom-coded configuration step .....	62
Figure 54.	Assembly, class and method selection.....	62
Figure 55.	Range of values of K.....	63
Figure 56.	Pattern groups selection .....	63
Figure 57.	Metric ready to assess quality .....	63
Figure 58.	Metric results.....	64
Figure 59.	Correctness metrics related to nouns.....	65
Figure 60.	Correctness metrics related to verbs.....	66
Figure 61.	Database connection .....	67
Figure 62.	Correctness metrics selection .....	68
Figure 63.	Parameters configuration.....	68
Figure 64.	Generate classifier .....	69
Figure 65.	Selecting systems and subsystems from SCM .....	69
Figure 66.	Assessing the quality metrics .....	70
Figure 67.	Saving snapshot with the quality of the project .....	70
Figure 68.	Graphical representation of the quality evolution .....	71
Figure 69.	Information of the snapshot.....	71
Figure 70.	Correctness metrics for model template .....	72
Figure 71.	Assign template to the model .....	72
Figure 72.	Assess quality of the model.....	73
Figure 73.	Creation the checklist metric.....	74
Figure 74.	Complete checklist metric.....	74
Figure 75.	Window to answer the questions.....	75
Figure 76.	Checklist results statistics.....	76
Figure 77.	Checklist results for each requirement .....	77
Figure 78.	Two instances of the component Block1 compose the System component .....	78
Figure 79.	Internal Block Diagram used for architectural refinement .....	79
Figure 80.	Model Explorer showing a valid system architecture hierarchy in SAVONA.....	79
Figure 81.	Design pattern selection dialog .....	81

Figure 82.	Role binding interface .....	82
Figure 83.	Wizard to set the parameters of the parameterized architecture .....	83
Figure 84.	Last page of the wizard to import the instantiated architecture into the current project .....	84
Figure 85.	“Profile” tab to edit the refining contracts.....	84
Figure 86.	Contracts Section in the Properties View of SAVONA .....	85
Figure 87.	Contract Wizard of SAVONA: Assignment of assumptions and guarantees .....	85
Figure 88.	Contract Wizard of SAVONA: Definition of contract refinements .....	86
Figure 89.	Example of component .....	86
Figure 90.	Example of nominal state machine (using SMV as action language).....	87
Figure 91.	Dedicated menu to perform the check of the contract refinements .....	88
Figure 92.	An example of the result of the check contract refinement shown in the Trace View. In this case, it is possible to see that one refinement is failed, and the counter example. ....	88
Figure 93.	An example of the result of the check contract implementation shown in the “Trace” view. It is possible to see that four contracts are not satisfied by the state machines of their associated component and one contract not.....	89
Figure 94.	AMT2.0 STL contract editor.....	90
Figure 95.	AMT2.0 Evaluation view.....	91
Figure 96.	AMT2.0 diagnostics results.....	92
Figure 97.	Weak contract selection for a component instance .....	93
Figure 98.	Performing refinement analysis with strong and weak contracts.....	93
Figure 99.	Integration workflow: from contract-based design to the generation of saboteurs and monitors .....	95
Figure 100.	Sabotage: global vision of the implementation with different tools .....	96
Figure 101.	Select the Simulink file .....	97
Figure 102.	Fault Target Block Selection .....	98
Figure 103.	Fault List Creation .....	98
Figure 104.	Faulty System Generation .....	99
Figure 105.	Loading the CHESS file in Sabotage Framework .....	100
Figure 106.	The possible connections of the system model .....	101
Figure 107.	Assertions of a specific component .....	101
Figure 108.	State machine modelling faulty behaviour .....	102
Figure 109.	A fault tree visualized in the CHESS Editor View; note the probabilities associated to the top and basic events .....	103
Figure 110.	The FMEA table in the CHESS view .....	104
Figure 111.	Dedicated menu to perform the compute the contract-based fault tree .....	105
Figure 112.	Block Definition Diagram exported as vector image .....	108
Figure 113.	Instance of an Internal Block Diagram exported as vector image .....	109
Figure 114.	Popup to set the preferences of the generated document and diagrams.....	109
Figure 115.	Initiating the argument-fragment generation.....	110
Figure 116.	Selecting the source analysis context .....	111
Figure 117.	Selecting the destination assurance case folder on the CDO repository .....	112
Figure 118.	Generation successfully completed with argument-fragments for each block .....	113
Figure 119.	An example of the generated argument-fragment .....	114
Figure 120.	Functional decomposition of the AIR6110 case study.....	117
Figure 121.	WBS architecture overview (MV=Meter Valve; ASV=AntiskidShutoff Valve; W=Wheel) .....	119

Figure 122. Pictorial view of the Linked Balises on the track .....	121
Figure 123. ETCS Linking model - Requirements Diagram .....	121
Figure 124. SysML Block Definition Diagram of the parameterized model.....	123
Figure 125. Internal Block Diagram of parameterized Physical block.....	124
Figure 126. Internal Block Diagram of parameterized “Linking system” block .....	124
Figure 127. Tab Profile to link formal properties to requirements.....	125
Figure 128. Exepctation_window formal property .....	126
Figure 129. Scenarios formalization.....	126
Figure 130. SysML Block Definition Diagram of the instantiated model.....	128
Figure 131. Screenshot of a part of the documentation generated in HTML format .....	132
Figure 132. Architecture Overview of DC Drive Case Study (simplest version).....	134
Figure 133. The physical implementation of the DC Drive Case Study as an experimental setting in AMASS.....	135
Figure 134. A semiformal requirement for the DC Drive .....	136
Figure 135. Example of a macro expansion wizard allowing specification of semantically correct requirements.....	137

## List of Tables

Table 1.	Artefacts produced by analysis.....	106
Table 2.	Summary of modules supporting the different phases of the AMASS architecture-driven assurance.....	114

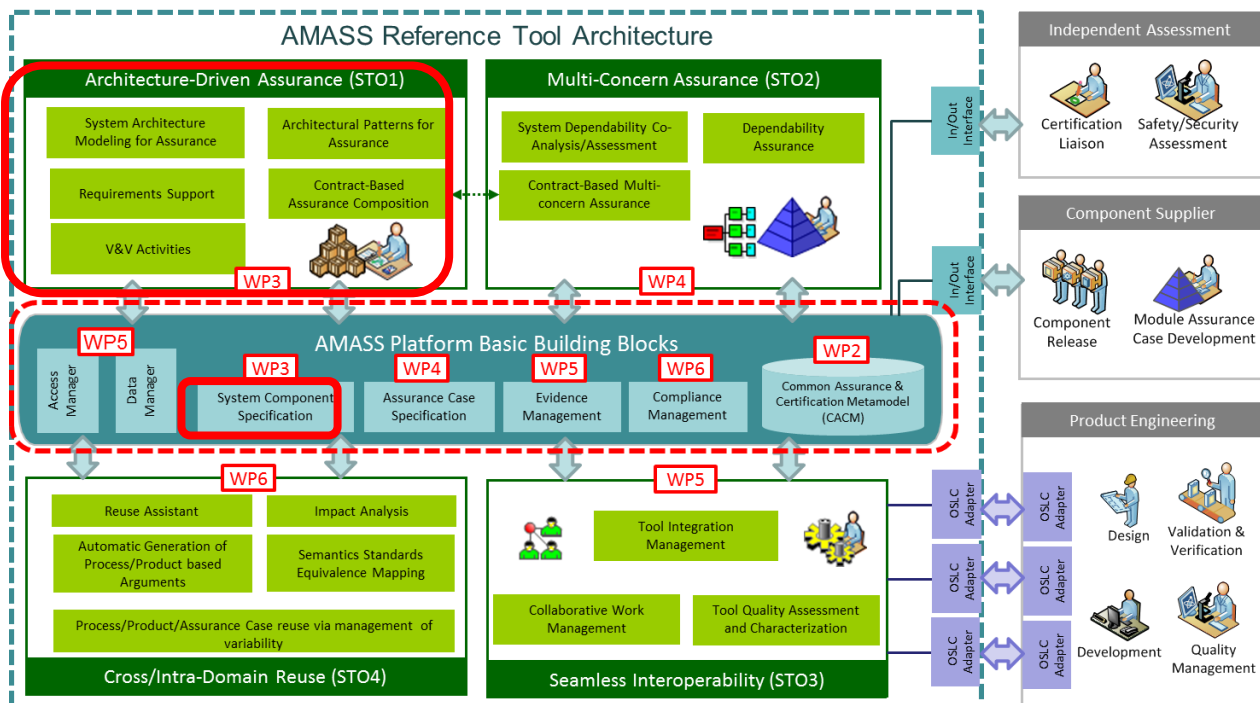
## Executive Summary

This document is the final deliverable associated with the AMASS Task 3.4 Methodological Guidance for Architecture-driven Assurance, which provides the methodological guide for the AMASS Architecture-Driven Assurance approach. This deliverable represents an update of the AMASS D3.7 [30] deliverable released at m20; several sections have been added or modified as summarized in “Appendix A: Document changes respect to D3.7”. While D3.7 was based on the functionality supported by the second prototype (P1) of the AMASS platform, this deliverable, D3.8, is based on the third and final version of the prototype (P2).

This document focuses on the techniques developed in WP3. It guides the users step by step in the usage of the AMASS platform to support the architectural design of a system collecting modelling artefacts and the related results of early validation, verification, and safety analysis to be used in the safety case. To have a more general overview and guide for the AMASS approach, including the methods and techniques provided by other WPs, the reader is referred to D2.5 “AMASS user guidance and methodological framework” [33]. Also, to have a more detailed description of specific functions, the reader is referred to the tool user manual, which is included in the above-mentioned deliverable D2.5.

# 1. Introduction

Embedded systems have significantly increased in technical complexity towards open, interconnected systems. The rise of complex Cyber-Physical Systems (CPS) has led to many initiatives to promote reuse and automation of labor-intensive activities such as the assurance of their dependability. The AMASS project builds on the results of two large-scale projects, namely OPENCOS [49] and SafeCer [48], which dealt with assurance and certification of software-intensive critical systems using incremental and model-based approaches. In particular, SafeCer developed a generic component model and contract-based verification techniques for compositional development and certification of CPS. These have been integrated in the CHES tool support [50]. The AMASS project consolidates and extends such support with a wider range of analysis techniques for the system architecture and combines it with the OPENCOS solutions for building an assurance case. The resulting Architecture-Driven Assurance is further enhanced for multi-concern aspects (in particular, the interplay between safety and security), for reuse of architectural patterns, and exploits tool interoperability mechanisms to interact with external tools for modelling and analysis support.



**Figure 1.** Architecture-Driven Assurance in relation to the other work packages

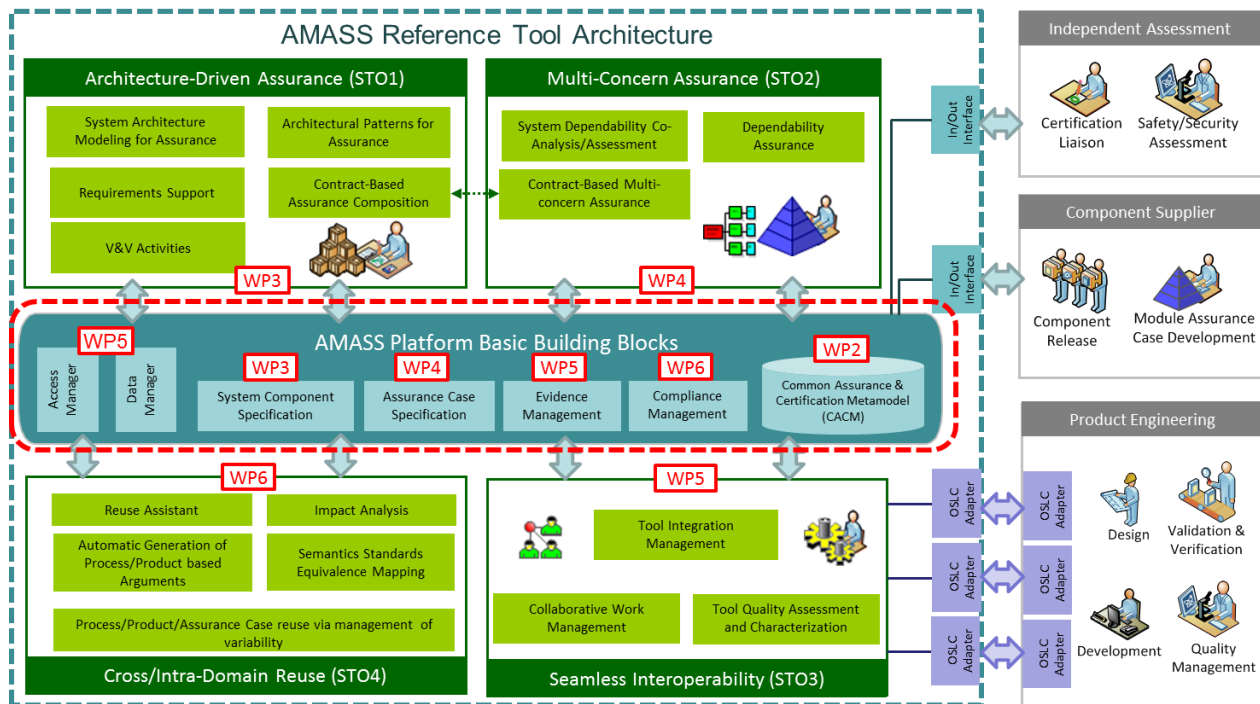


Figure 1 provides a general overview of the different AMASS Scientific Technical Objectives (STOs) and how they are implemented in the AMASS project by specific Work Packages (WPs). This document focuses on methodology guidelines for Architecture-Driven Assurance and the related tools developed in WP3.

Thus, it defines a methodological guide to apply the Architecture-Driven Assurance approach, to use both its conceptual aspects and its software tool support. It first provides an overview of the key concepts, such as system architecture, contract-based design, early verification and validation, and model-based safety analysis. Then, it details what Architecture-Driven Assurance means, the role of the key concepts in the approach, and how the AMASS platform supports it. An overview of the tool architecture is given, in particular of how the core component, CHES, interacts with the external tools. In the rest of the document, the term “tool”, when not more specifically defined by the context, refers to the AMASS platform (of which CHES is a core component).

This guide describes the methodology to follow, detailing the process steps and how to use the tool support. The workflow is presented by means of activity diagrams or sequences of steps to follow, with details on how to use the AMASS platform to perform each step. The steps are meant to give an example of usage of the tool trying to cover all relevant features. The user is referred to the user manual, that is included as an Annex of D2.5 [33], to get a deeper knowledge about the different options.

The guide uses simple case studies to concretely describe the approach. The material (more specifically, the set of CHES projects) of these case studies is released with the tool.



## 2. Architecture-Driven Assurance Overview

### 2.1 Background

This section gives a brief description of the main concepts used in the guide. In each subsection, first, the concept itself is briefly described, and, second, the related tool support is summarized. The reader is referred to D3.3 “Design of the AMASS tools and methods for architecture-driven assurance (b)” [28] for further details on these concepts.

#### 2.1.1 System Architecture

As part of a model-based engineering approach to system development [60], the system architecture plays an important role in the early phases of the system design. The system is first considered as a whole, defining system requirements and boundaries. This initial level of architecture can fulfil the role of an “Item Definition” according ISO 26262 [81]. The system architecture model is then defined to detail how the different parts of the system are connected and interact to fulfil the system requirements. The system architecture model is used to capture a variety of information: the internal hierarchical structure of the system and its components, the system functions and component behaviours, and the component interfaces, their connection and interaction. The model is used for early verification and validation, meant here as model-based techniques to verify that (a certain part of) the model satisfies a higher-level specification and to validate that the model captures what the designer has in mind. The system architecture model is also used for allocation of the safety-related requirements identified during the safety analysis and safety concept creation phases: it includes the safety mechanisms implemented in the system to reduce the risk introduced by system failures, and the model of faults used to derive of the safety mechanisms and assess their effectiveness.

The AMASS platform supports the modelling of the system architecture with SysML and UML diagrams for what regards the modelling of the requirements (SysML Requirement diagrams), the internal hierarchical structure of the system and its components (SysML Block Definition and Internal Block diagrams) and the components behaviours (UML State Machine diagrams). Moreover, the CHES modelling language (CHESML)<sup>1</sup> [61] provides means to extend the aforementioned diagrams to support the modelling of dependability concerns and to apply the contract-based design, the latter introduced in the next Section 2.1.2.

#### 2.1.2 Contract-Based Design

The challenges posed by the design of complex cyber-physical systems [36] pushed the research of contract-based techniques for system design (e.g., [37][38][39][40]). The system architecture model is enriched with expressions asserting the expected properties of the system, its components and environment. In order to allow compositional reasoning, the property of a component may be restricted to its interface considering the component as a black box (without constraining the internal variables of the component) and can be structured into contracts, pairs of properties representing an assumption and a guarantee of the component: an assumption is an assertion on the behaviour of the component environment, while a guarantee is an assertion on the behaviour of the component provided that the entire set of assumptions holds. If assumptions and guarantees are formal properties, which means they are specified in a formal language such as some specific kind of temporal logic, the architectural decomposition can be verified by checking that the contract refinement is correct: this consists of checking that, for all composite components, the contract of

---

<sup>1</sup> CHESML is implemented as UML, SysML and MARTE profile.

the composite component is ensured by the contracts of the subcomponents – considering their interconnection as described by the architecture - and that the assumption of each subcomponent is ensured by the contracts of the other sibling subcomponents and the assumption of the composite component.

The contract specification can further be enriched by categorising contracts into strong and weak [82] to allow for better support for specification of reusable components behaviour. Such components are intended to work in different environments, and often exhibit environment-specific behaviours and assumptions. They have also been proven useful for specifying different behaviours or levels of quality for different configurations, operation modes or degradation levels, which can in particular help specifying safety properties of Systems-of-Systems that reconfigure at runtime (e.g. vehicle platoons), as outlined in [72].

A strong contract, denoted by  $\langle A, G \rangle$ , requires the environment to satisfy the assumption  $A$  so that the component cannot be used in an environment violating  $A$ . On the contrary, a weak contract, denoted by  $\langle B, H \rangle$ , is equivalent to the strong contract  $\langle \text{True}, B \Rightarrow H \rangle$ : the component can be used in all environments, while the guarantee  $H$  is specific for the environments that satisfy  $B$ . In other words, strong contracts must always hold (if their assumptions are violated, the behaviour of the system is completely undefined and the system can even be destroyed), whereas the weak contract may hold or not in certain environments (i.e. in any given environment, only the guarantees will be assured that belong to a contract set of which all assumptions are fulfilled in this environment). In fact, the assumptions of weak contracts may be even in mutual exclusion and only some of them are satisfied by the same environment.

The effectiveness of the contract-based design approach applied to complex cyber-physical systems is faced with several challenges. CPS are heterogeneous systems – they combine software and hardware and even optical or mechanical components, exhibit a combination of discrete event interactions with (often non-linear) continuous control dynamics and interact with an unpredictable physical environment. Application of contract-based design at the system-level of complex CPS requires addressing their heterogeneity, knowing that many basic operations from contract-based design, such as the refinement, are in general undecidable in presence of continuous dynamics. The recent survey [39] on contracts for system design provides an overview of this versatile approach to rich domains, such as real-time and probabilistic systems. More specifically, a contract-based design methodology for developing controllers in CPS is proposed by Nuzzo et al [73]. In this work, the contracts are expressed in Signal Temporal Logic (STL), a temporal assertion language designed to express system-level properties of CPS. In a related work [74], the complexity of CPS is tackled by considering probabilistic contracts (expressed in stochastic variant of STL) and developing algorithms for checking contract operations such as contract compatibility, consistency, and refinement, in a stochastic setting. The contract-based design is increasingly gaining attention in industry, especially in the automotive domain, where companies such as Toyota, Volvo Cars, Bosch, and Boeing used contract languages such as LTL and STL to formalize their functional CPS requirements, and build a rigorous testing methodology around it to check violations of implementation and refinement [75][76][77][78][79][80][19]. To summarize, the adaptation from contract-based design in software to contract-based design in CPS is a vivid area of research, which is also actively studied in AMASS.

The AMASS platform supports the specification and analysis of contracts where assumptions and guarantees are expressed in Linear-time Temporal Logic [52] (LTL with future and past operators [53], first-order constraints [54] such as linear constraints over integer and real numbers, discrete or super-dense time model [55][56]) or Hybrid extension of LTL (HRELTL) [57]. Finally, the external tool AMT2.0 uses contracts specified in Signal Temporal Logic (STL) [59].

### 2.1.3 Semi-formal Specification of Requirements and Contracts

Because formal expressions are hard to write and understand by non-experts, they tend to be avoided in practice. That turns out to be very unfortunate as they provide many striving characteristics, from which requirements engineering processes would benefit. A well-defined syntax and semantics offer only one way to interpret statements, making e.g. automatic verification and tracing possible. Expressions in

unconstrained natural language might be easier to read, but have no constraints in syntax and semantics, resulting in ambiguous statements which make automated processing or verification nearly impossible. In many cases, it takes an expert with appropriate domain knowledge to interpret and validate the expressions correctly, and even then, different experts might disagree on the exact meaning. This can even be caused by the fact, that some relevant details are simply not addressed at all in the human-made natural language specification.

Template Languages [5] can close the gap between purely formal expressions and unconstrained natural language. They provide a well-chosen set of allowed sentence patterns, which results in a constrained natural language featuring a well-defined syntax. Ideally, the template language also has unambiguous semantics, leaving only one way to interpret an expression.

There exist various attempts to semi-formalize requirements. Requirement Boilerplates [5][6][7] offer a set of predefined sentence patterns with placeholders that must be substituted by keywords such as component, interface or function names. Advanced semi-formal specification languages [8][9] feature not only syntactical rules but also semantical meaning to the expressions built. The Requirement Specification Language (RSL) [8], developed in the Artemis project CESAR (Cost-efficient Methods and Processes for Safety-relevant Embedded Systems) [10], allows the semi-formal specification for various types of requirements, such as functional, safety, architecture, etc. The Goal and Contract Specification Language (GCSL) [9] which has been developed during the DANSE (Designing for Adaptability and evolution in System of systems Engineering) project [11] considers the contract paradigm and allows a formal contract structure with semi-formal assumption and guarantee notation. Building on the aforementioned pattern language and some additional considerations inspired by industrial experience, a new pattern language called System Specification Pattern Language (SSPL) [71] has been developed during the AMASS project and applied onto some of the AMASS case studies (in particular, the DC Drive system).

An integration of semi-formal languages and specifications with system modelling tools can greatly improve the development process. Online expression checks on requirements or assertions can be made based on the existing system model used as ontology. If a semi-formal specification is fully translatable to a verifiable formal language such as LTL or HRELTL (perhaps not for all, but just for some of the assertions it can express), the system model can be verified against the specification enabling early V&V.

The current AMASS platform supports the OCRA grammar<sup>2</sup> to formalize requirements. This has a formal semantics, corresponds to LTL and HRELTL but with English words instead of mathematical symbols.

## **2.1.4 Requirements Validation**

Requirements validation is a fundamental step in the development process of software and system design. In fact, requirements are typically specified in natural language, and flaws and ambiguities in the requirements can lead to the development of correct systems that do not do what they were supposed to. The role of requirements validation is to check if requirements are specified correctly. Possible faults in the requirements are conflicts, ambiguities, incorrect values, incomplete cases, missing assumptions, over-specification, etc. Formal methods for requirements validation are being devoted increasing interest (e.g., [43][44][45][46]).

The AMASS platform provides different techniques to validate the requirements either based on quality metrics or on formal semantics analysis (provided that requirements are formalized into formal properties).

---

<sup>2</sup> <http://ocra.fbk.eu>

### 2.1.5 Verification and Validation of Behavioural Models

A behavioural model describes the internal dynamics of a component. The model can describe how the internal state of a component is updated or the functional update of outputs based on the inputs. These behavioural models can be verified by means of model checking [47] against some formal properties in different temporal logics. The formal properties can represent some requirements (e.g., functional or safety-related requirements) or some validation queries such as the reachability of states. Model checking performs an exhaustive search of the state space. However, it suffers of the well-known state space explosion problem. Thus, modern symbolic model checking techniques combine search and deductive techniques.

In combination with contract-based design, the verification can be performed compositionally: state machines are verified separately against the local contracts of the corresponding components and the correctness of the system is implicitly derived by the correctness of the contract refinement and the local state machines.

As an alternative and complementary approach, the properties can be compiled into monitors that observe individual execution traces and check whether they satisfy or violate the specification during simulation or test runs. This pragmatic approach provides a scalable, yet rigorous technique to reason about systems that are too complex for formal verification and model checking. In addition, property-based monitoring techniques can be applied to assess the correctness of black-box systems.

In AMASS, the behaviour of components is specified with either state machines or other external modelling languages such as Simulink. The AMASS platform provides model checking techniques supported by the nuXmv model checker [17] and contract-based reasoning supported by the tool OCRA [16]. These tools are integrated in CHESS as backends. Monitor compilation is instead supported by AMT2.0, which is integrated with the Simulink models.

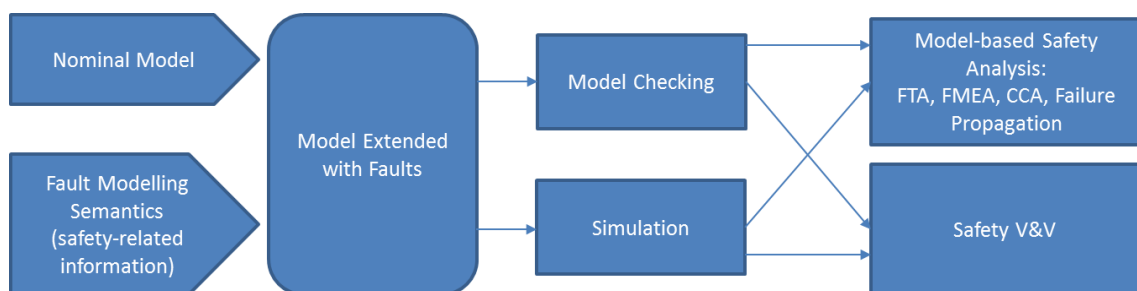
### 2.1.6 Model-Based Safety Analysis

Model-Based Safety Analysis (MBSA) [51] provides a set of techniques aimed at analysing the safety of a system based on the models used for system design and development (also referred to as “nominal models”). A key step of MBSA is **fault injection**, i.e. the deliberate introduction of faults into the system nominal model [21] [22] [23]. This enables the validation (e.g., by means of simulation), verification (e.g., model checking or monitoring), or, more in general, safety analysis (e.g., minimal-cut-sets analysis) of fault-tolerance mechanisms. It basically consists of introducing faults into a system, analysing its behaviour with respect to the introduced faults and determining which kind of actions or measures must further be taken, until a stage is reached where the system can cope with all reasonably foreseeable failure cases. MBSA contributes to the safety analysis phase, which includes the verification and validation of safety concepts and requirements. Some of its most remarkable aims are to support the assessment of implemented safety requirements, and the correct implementation and the effectiveness (diagnostic coverage) of safety or fault tolerant mechanisms. Traditional safety analysis methods such as Fault Tree Analysis (FTA) or Failure Mode and Effect Analysis (FMEA) are typically performed manually and are often not sufficient. Manual reviews are normally needed to prove the completeness and the correctness of those analyses [24]. Furthermore, the failure logic or the effects of certain faults cannot easily be determined by those analysis techniques. A promising approach to overcome this limitation is to combine traditional analysis with MBSA approaches. It is important to understand that MBSA mitigates the new challenges, but cannot replace safety assessments such as FTA or FMEA done in the traditional way. Therefore, MBSA and traditional safety analysis techniques complement each other.

There are different MBSA techniques like symbolic MBSA or simulation-based MBSA. Symbolic MBSA arises as an attempt to introduce formal methods into the area of fault injection in order to evaluate the dependability of safety-critical computer systems. Meanwhile, simulation-based MBSA realises a controlled testing experiment to evaluate the behaviour of the system in the presence of faults.

Fault Injection requires, on the one hand, a formal or an executable design model from which an undesired behaviour is modelled with the system model. On the other hand, safety analysis techniques based on model checkers (e.g. xSAP [25] [26]) or on simulation (e.g. Sabotage [27]) [13], help to analyse the system model extended with faulty behaviour. This helps, for example, to find inconsistencies between the modelled and the safety requirements. As explained in D3.3 [28], when setting up the fault injection environment, it is important to define the fault injection policy which is called fault list [34]. This configuration process includes the definition of fault locations, fault injection times, fault durations, and the input data for the system.

Figure 2 depicts the role of applying multi-techniques with the intention of completing an early safety assessment, showing how they contribute to architecture-driven assurance. The novelty of this approach lies in the combination of simulation and model-checking to automate the safety analysis construction, define the needed safety measures, verify the safety mechanisms and validate if the required level of safety is achieved.



**Figure 2.** MBSA techniques for early safety assessment

#### 2.1.6.1 Simulation-Based MBSA

Among the different MBSA techniques, simulation-based approaches emerge as a promising solution to provide an early safety evaluation and V&V of a system.

Simulation-based MBSA involves the construction of a behavioural model of the system [35]. The simulation models can be developed on different level of abstractions such as Simulink/SCADE or using hardware description languages like Very High speed integrated circuit Hardware Description Language (VHDL). In the context of AMASS, only the first category is considered.

In order to identify differences in the system's behaviour and to automate the fault injection campaigns, the simulation results of a faulty system under test (faulty SUT) or extended system model with faulty behaviour are compared versus a fault free system (golden SUT) under the same workload. Extra model blocks (saboteurs) are injected into the component inputs, which reproduce a certain failure mode. After that, the effect of that fault can be observed in the output by including extra read-out blocks or monitors. These fault injectors simulate failures at input ports and the inclusion of monitors in the output ports in order detect whether and in which ways an output assertion is violated in consequence.

The results can be stored as part of the safety case as applies to the conventional safety analysis techniques.

#### 2.1.6.2 Symbolic MBSA

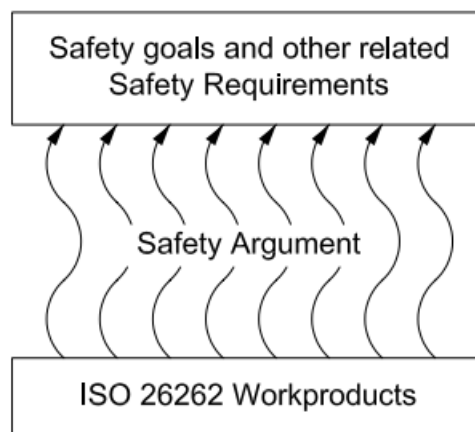
Symbolic MBSA [51] searches for all the possible combinations of faults (minimal cut sets) that may lead to a system failure. The result can be presented in form of a fault tree, where the system failure represents the top-level event and the injected faults are the basic events.

If the system architecture is enriched with contracts, a fault tree can be generated semi-automatically to represent how the failures of components can be propagated and result in a system failure, based on an analysis of the contract refinement. In this case, the failure of a component represents the inability of the component to fulfil any of its guarantees although all assumptions about its environment hold, while the failure in the component's environment represents a violation of any of the component's assumptions. At

every level of the contract refinement, minimal cut sets (or minimal cut sequences in the case of timed behaviour) can be computed to see how the failure of the composite component depends on the failure of the environment and the failure of the subcomponents. The results can be integrated into a hierarchical fault tree that respects the architectural decomposition of the system.

### 2.1.7 Safety Case

In most industries, a well-structured Safety-Case, i.e. a concluding argument that the system to be released for public usage is sufficiently safe, is required by safety standards and certification authorities. Standards mentioning the obligation for a Safety Case include IEC 61508, ISO 26262 and many more. A more comprehensive list and a compilation of notions of Safety Case (and problems with informal definitions what a Safety Case exactly is) is given in [58][65]. The notion of a Safety Case is not formally defined by the standards and therefore varies among different standards, regions and industry branches – it can even depend on the company who creates it or on the safety assessor or local certification authority. Where this term is in use, it refers at least to a collection of all relevant output documents from the safety process, from which an external assessor can conclude that everything has been done to assure that the product is safe in its practical application. A more formal interpretation shaped by Tim Kelly is a structured representation of argument lines that show the fulfilment of every safety goal by providing evidences, see Figure 3.



**Figure 3.** Safety arguments show the fulfilment of safety goals and other related safety requirements by providing evidences (e.g., ISO 26262 work products)

The lines of argumentation may be long and winding, requiring a structured representation, e.g. using the popular Goal-Structuring-Notation (GSN). This way of thinking has influenced British aerospace industries as well as the automotive standard ISO 26262 (but only in its informative part, i.e. as a recommendation). Similar notations are the “Safety Concept Trees” [70] proposed by Fraunhofer IESE or the notation “Claims, Arguments and Evidence (CAE)” suggested by the tool company Adelard<sup>3</sup>, or the graphical structuring of safety concepts provided by the tool medini analyze<sup>4</sup>. All of them have in common that they are tree-style notations that iteratively decompose the safety goal and graphically distinguish between different semantic items like claims, arguments and evidences (with varying terminology).

Looking closer at safety cases from real industries, we can often notice that two levels of abstraction are involved:

1. The process level, where the individual evidences are safety work products (e.g. a FMEA report, a test report, a review report), demonstrating that all necessary process activities have been arranged

<sup>3</sup> See product website: <https://www.adelard.com/asce/choosing-asce/cae.html>

<sup>4</sup> See product website <http://www.medini.eu/index.php/de/products/functional-safety>



in a way as to provide a gap-less argument (e.g. Why are we sure that we have considered all reasonably assumable hardware failures? → Because we have performed an FMEA and had it reviewed by independent experts!).

2. The level of **technical product development**, where individual safety mechanisms (e.g. a range check, a watchdog), their describing requirements, the architectural items to which they are allocated and their corresponding individual verification artefacts (a passed test case, a formal proof for some property, etc.) are all linked to each other, allowing the reader to follow the argument on a technical level (e.g. Why are we sure that the power part is switched off in case that some software routine is trapped in an endless loop? → Because we have implemented a safety mechanism consisting of a software part that monitors the program flow and a hardware watchdog that disables the power part via a dedicated wire if the software is detected to be hanging).

The usage of tree-style notations for safety argument structuring in a technical safety concept, belongs to the second category, whereas the usage of tree notations to structure the process argument in a safety case belongs to the first category. Notations like GSN are applicable for both purposes in industry.

Other classification terminologies have been proposed in literature, all roughly addressing the same semantic difference, e.g.

- Indirect evidences = process level
- Direct evidences = verification (test) results on product level (right side of the V-model)
- Immediate evidences = the design artefacts describing the product technically (left side of the V-model)

However, the distinction in left and right leg of the V-model can be seen problematic these days, because firstly, in the era of agile development and bottom-up system construction by reusing existing parts, the general applicability of the V-model can be doubted at all, and secondly, early prototyping, simulation, analysis and the like happen already in the left leg of the V-model, but provide similar kind of evidences as the test results in the right leg (see Figure 5). More contributions to this classification can be found for instance in [69].

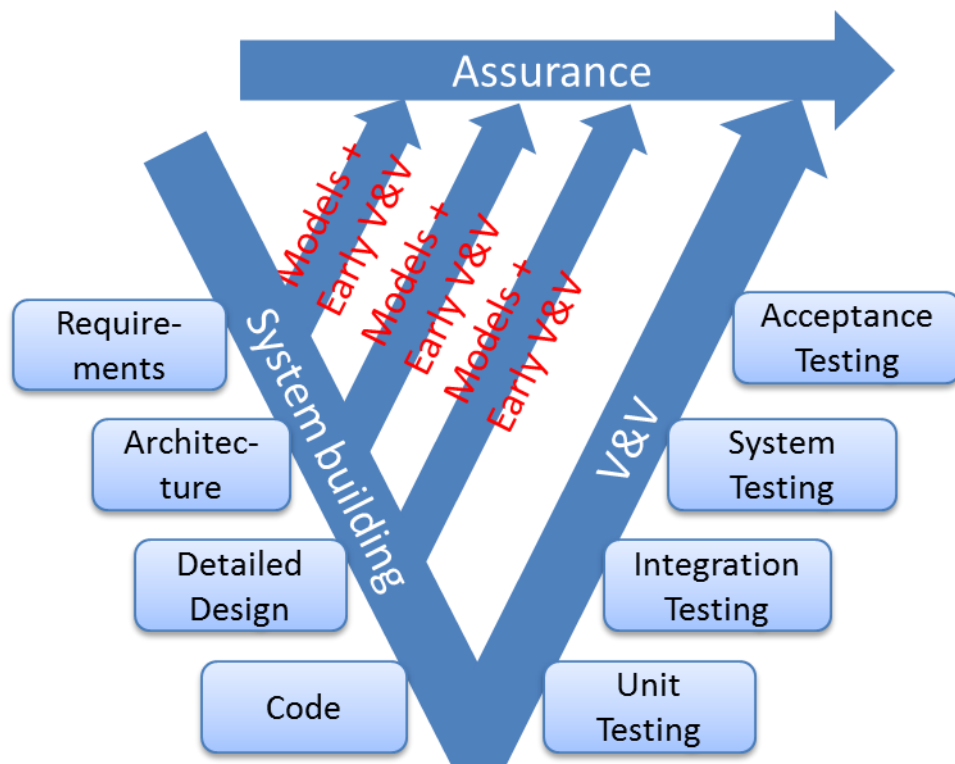
In AMASS, both levels are linked to each other via the meta-model: the model-based artefacts from (1) contain the individual model elements from (2). This is depicted in Figure 4: in the upper section of the figure, the safety artefacts on process level can be seen, whereas the lower section shows architecture, requirements faults/failures and safety mechanisms. The links crossing the boundary between upper and lower section show the containment relationships. The AMASS tool chain adheres to this meta model (see CACM description in AMASS D2.8 “Integrated AMASS Platform (c)”) and allows navigable links in accordance to the meta model links, so traceability is assured from project artefacts constituting the safety case to individual modelling elements in the functional / technical product architecture.





The architectural design phase provides an enormous opportunity for the preparation of the safety case: most design choices, especially related to the safety measures to be included in the system, are made in this phase and it is of paramount importance to explain and justify these design choices in the argumentation, as well as the context in which they were taken; moreover, the availability of models enables the possibility to verify and validate the design earlier than the typical V&V phase; while the main purpose of early V&V is to reduce cost and time discovering problems before the system is implemented and deployed, the generated artefacts provide an amount of evidence for the safety case that is typically not present in the traditional workflow.

The architecture-driven assurance proposes to develop the assurance case along the development process collecting the inputs of the system architecture modelling and early V&V as depicted in Figure 5. Note that the V-Model is a simplification and not meant to be the realistic workflow; moreover the picture does not show the iterations that are necessary in practice, in particular when safety needs to be considered (iterative cycle of design → safety analysis → additional safety requirements → design, until the remaining risk is considered acceptable). It rather highlights the overall idea of early model-based verification and validation, as well as the interaction of the system development with the construction of the assurance case thanks to the model-based system architecture.



**Figure 5.** Modelling and Early V&V providing input to the assurance case

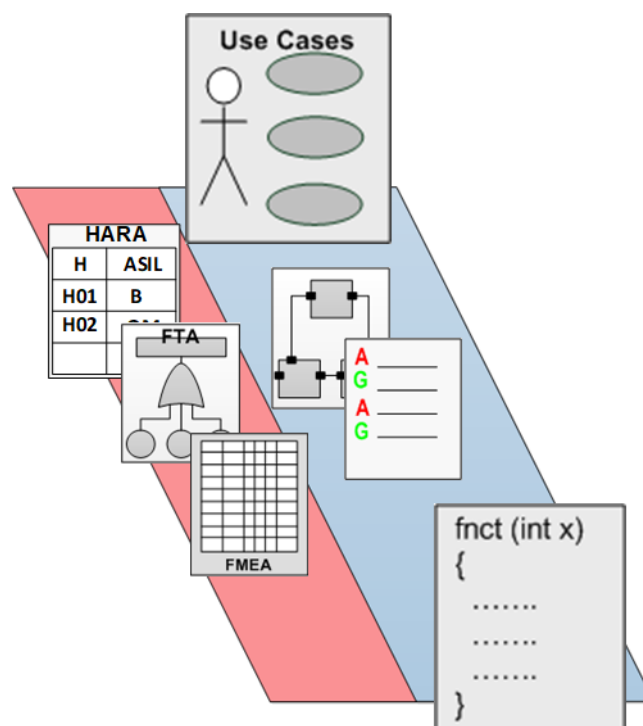
The Prototype P2 of the AMASS platform supports the Architecture-Driven Assurance approach by:

- Providing a rich set of early V&V and model-based safety analysis techniques,
- Collecting the results of the analysis as evidence for the assurance case,
- Linking the modelling elements to the corresponding elements of the assurance case,
- Generating argument fragments from the models.



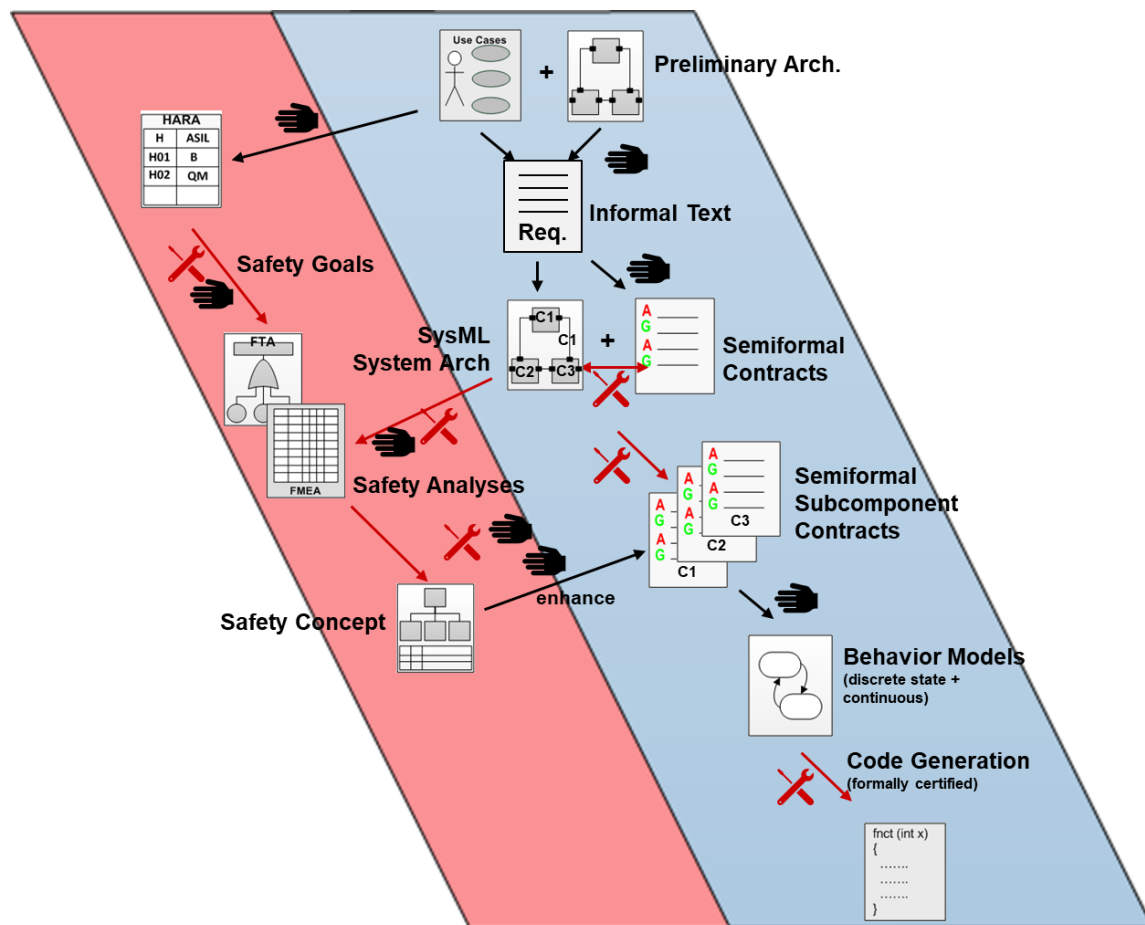
are enabled by the specification, modelling, analysis, verification and validation activities proposed in this methodological guide.

Let us now come one step closer and have a more detailed look on the left leg of the V model, see Figure 7. It is obvious, that the foundations for model-based analysis and V&V are laid on the left leg of the V model. The activities on the left leg are concerned with the transformation of an initial, coarse and informal product idea and item definition (maybe just a list of functions or use cases, maybe a requirements collection in natural language) to more and more detailed models and more formal requirements, which are decomposed onto the subsystems, components, subcomponents etc. of the system-to-be. The quality and degree of formality of these requirements and the models is key for efficient or even automated verification of the refinement, but also of the compliance of the implemented blocks in the end with their specification, and also for systematic safety analysis (see Section 4 of [3]). The desired output of the whole chain of activities is clearly the technical implementation, laid down as mechanical drawings, part lists, electronic hardware circuit diagrams, and code for hardware and software, such as C or VHDL language.



**Figure 7.** Zoom-in on left leg of the V-model

To explain how this transition is performed in detail in the AMASS context, Figure 8 shows more details on the work products and interconnections.



**Figure 8.** Workflow in the left V-leg with normal function development and safety activities

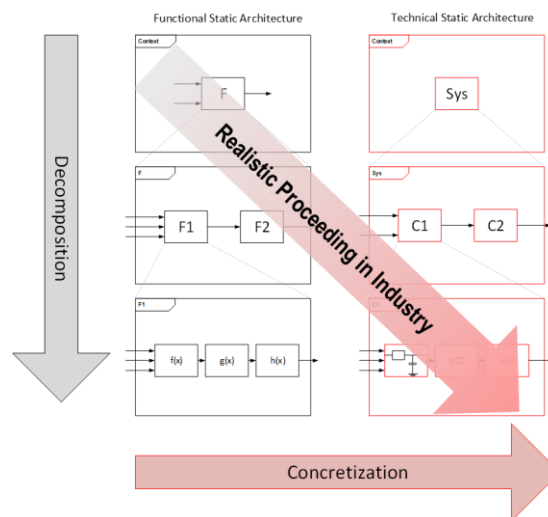
For the initial requirements capturing (e.g. as function lists or use cases) and for the preliminary architecture, no specific tools or notations are assumed (can be even standard office tools), but after that, the requirements will have to be captured in a structured way, allowing their atomization and identification (even if they are still written informally, in plain natural language) and the architecture on the highest level should be modelled in a semi-formal language like SysML, which has at least a well-defined syntax and some modelling guidelines that come with it. As these are standard activities, it does not come as a surprise that there are many different choices for appropriate tools from AMASS and from outside of AMASS, the AMASS tools even partly overlapping on this area. Possible tools for SysML architecture design are CHES/Papyrus, SAVONA, SCADE Architect, and many commercial tools, and also for requirements capturing a variety of tools is on the market (IBM Doors, PTC, Jama, just to name a few of them).

The next, very important step is the stepwise formalization, and the decomposition along with the architecture hierarchy. Doing requirements refinement and architecture design in an intertwined way is one of the key success factors, and not yet supported a lot by commercially available tools. To do so, AMASS proposes the application of the contract-based design paradigm, and the application of template-based languages which allow first to restrict the syntax of the assertions (semi-formal representation) and then, wherever possible, specify them in a language that provides a formal semantics, enabling both

- Verification of the refinement between the levels of the architecture, and
- verification of the implementation at the end of the left V-leg with the respective contracts for each leaf component of the architecture.

When tending to apply formal methods, it should be noticed that the whole process of going from initial specification to technical implementation is often termed “**refinement**”, but in practice it consists of two different dimensions of refinement:

1. **decomposition** (vertically going from components to subcomponents, sub-subcomponents, etc.), and
2. **concretization** (horizontally going from a more abstract and implementation-independent specification to a more technical specification)



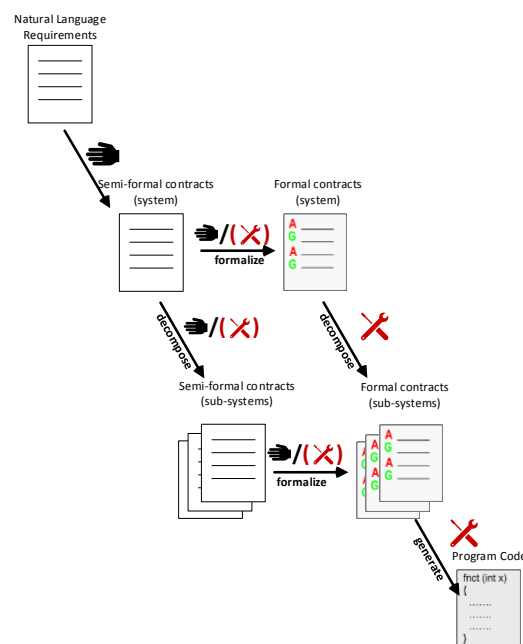
**Figure 9.** Dimensions of Refinement

In industrial practice, both dimensions are often mixed with each other, resulting in a walk roughly along the diagonal arrow in Figure 9. Note that, when choosing the appropriate level of formal vs. semi-formal specification of contracts and models, there is more than one way to go; therefore, the AMASS project partially applied **early formalization** with tools like the CHES (allowing formal verification of the decomposition, but making requirements specification and validation sometimes hard for humans, in languages such as Othello) and partially applied **late formalization**, which means that the specification language (e.g. System Specification Pattern Language / SSPL, see [71]) can offer more expressiveness and flexibility, but at the price that the refinement verification must be performed in a (still computer-assisted) manual process due to the lack of formal semantics. The prototypical tool SAVONA that has been created during the AMASS project provides an assistant function to support manual decomposition verification. The difference is explained in Figure 10: Early formalization means branching to the right on system level (e.g. going from SAVONA tool to CHES tool via the provided interface) and then decomposing, while verifying the correctness of the decomposition automatically using the CHES tool. Late formalization, accordingly, means to stay with SAVONA and decomposing, using the decomposition checker assistant for manual decomposition, and only on a low level formalizing the contracts by switching to the CHES tool (or, for instance, to SCADE suite, which allows generation of formally verified C code out of models). Note that the “hand” symbol in the figures represents manual activities, while the “tools” symbol represents activities that can be automated to a high extent. In some cases, the process must be performed manually, but tool support can help not to get lost in complexity, follow a structured approach and documenting the performed steps for the safety case.

Having explained the details on the process chain from informal requirements to formally verifiable source code, let’s discuss the role of safety analyses in Figure 8: The models created during the described development process are used as a starting point for the different safety analyses, and the contracts that specify the nominal behaviour can be used for systematic analysis of any kind of deviation or malfunction. The top-level system specification, for instance, can take the role of an Item Definition acc. ISO 26262. The

top-level function definitions initially specified in the Item definition, if modified by HAZOP-style guidewords like “more”, “less”, “early”, “late”, “unexpectedly”, “not at all” etc. help the analyst to systematically derive the potential hazards from the nominal functions. From the hazard list, the top-level safety requirements (termed “Safety Goals” in ISO 26262 language) are derived, which are then refined in the same way as the nominal function requirements, using the contract-based approach (see Section 4.1 of [3]). The causes and sub-causes for the hazards can be recursively investigated by means of Fault Tree Analysis, referencing the function blocks of the abstract levels of the model-based architecture. The more detailed and more technical architecture at the lower levels of refinement is usually analysed using the FMEA technique, which can, again, be semi-automatically derived from the technical architecture (e.g. the tool medini analyze automates this step already today). Details on the architecture-focused safety analysis can be found in Section 4.2 of [3]).

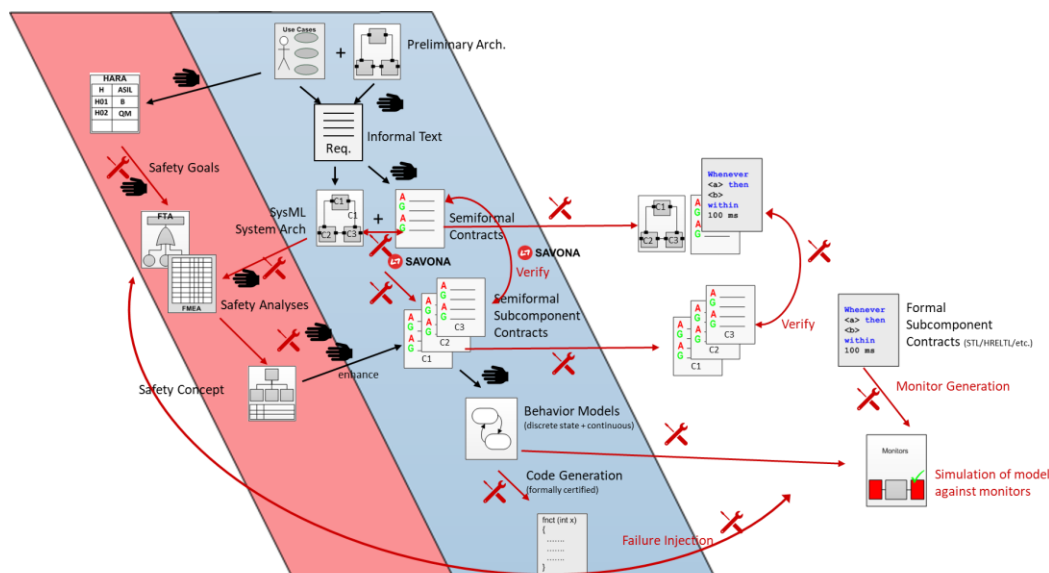
As the top-level claim of the safety case (decomposed using the GSN technique) is to prevent any reasonably foreseeable failures from violating a Safety Goal, it is obvious that the safety process will generate new safety requirements (in the AMASS project denoted as safety contracts, in the same way as functional requirements had been denoted as ordinary functional contracts). The safety contracts look similar to the functional contracts, but consider that there might be failures present in the system, and are qualified with a safety integrity level (e.g. ASIL, SIL or DAL) coming from the hazard they are supposed to prevent. These requirements are fed back into the normal development path (blue ribbon in Figure 6), where they augment the existing requirements set and lead to an enhancement of the architecture by additional blocks that represent safety mechanisms (e.g. a plausibility check for a sensor signal) or paths of redundancy (e.g. a second sensor for the same physical quantity, allowing cross-checking to detect failures), see Section 4.3 of [3]. This is where the iterative proceeding within the V-model becomes visible (loop between red and blue ribbon in Figure 8). This process continues until the analyses show that an acceptable risk level has been reached and the safety case can therefore be completed. Of course, the new requirements not only need to be added to the architecture models iteratively, but also must be implemented and verified (by testing, review, model checking, simulation, etc.) on a maturity level that is described by the safety standard (e.g. ISO 26262) for the applicable safety integrity level (the higher the SIL or ASIL gets, the higher the requirements regarding deeper testing, more formal notations and more independent reviewers and assessors).



**Figure 10.** Early vs. late formalization

The whole argument, based on the provided artefacts centred around the architecture, is finally captured in the safety case for the product. As evidences to support the safety case, not only the safety analyses, but all other verification and validation results are attached to the elements of the architecture, including test and simulation results or reports from model checkers. We will have to accept that many properties of the implementation still cannot be proven formally today. Yet it must be demonstrated in particular that all designed safety mechanisms actually prevent the supposed failures from causing a hazard. Therefore, besides formal verification of the decomposition and implementation, the AMASS proceeding also considers an appropriated semi-formal way of verifying the effectiveness of the safety mechanisms: a combination of monitor generation (again from semi-formal contracts) and fault injection and simulation (or testing), as shown by the additions in Figure 11. The cornerstones of this proceeding, which is supported by the tools AMT and Sabotage from AMASS consortium partners AIT and Tecnia is built on the following cornerstones in an efficient and tool-integrated combination:

- From Safety Analysis and Safety Concept, the failure mode hypotheses and the specifications of the safety mechanisms (acc. Section 4.3 of [3]) as contracts are exported.
- From the contracts (describing the nominal behaviour or the behaviour in presence of failures), which are specified in certain classes of temporal logics (e.g. HRELTL, Othello, STL), monitors are automatically generated. These monitors observe the behaviour of a model (e.g. VHDL, Simulink) during all simulation runs, or they can even be applied to the actual system and observe test runs.
- Test/simulation runs are specified and performed in a suitable model-based simulation framework.
- For all test sequences that shall demonstrate the capability of a safety mechanism to prevent a failure from causing a hazard, the failure is simulated by fault injection at the interfaces of architecture components.
- If all safety contracts hold, even in case of injected failures, the evidence is achieved for the claim in the safety case that all hazards have been sufficiently covered and this branch of the safety case can be considered complete.



**Figure 11.** V-Model with Monitor Generation, Fault Injection and Simulation

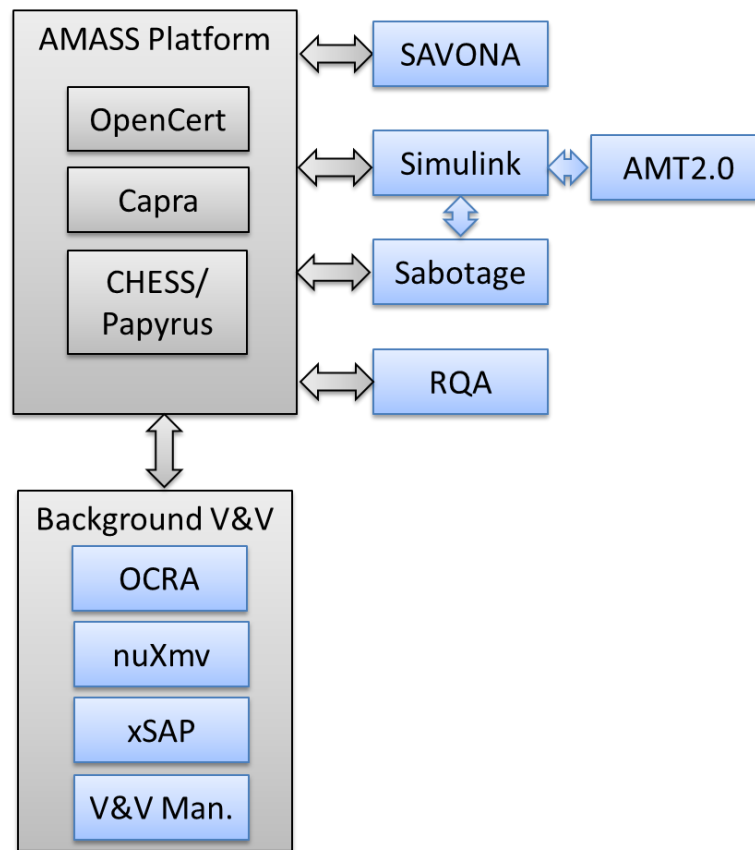


### 2.2.3 Tool Support Overview

The tool support is based on a collection of Eclipse plugins and external tools (also depicted in Figure 12) that provide the different functionalities necessary to perform the Architecture-Driven Assurance approach. In particular, it includes:

- CHESS/Papyrus plugins to model UML/SysML diagrams and to perform contract-based design and different model-based analyses, and OpenCert/Capra plugins to create and link assurance argument fragments and evidences or other traceability links. These plugins are part of the AMASS platform, which provides the user a single user interface hiding the complexity of the underlying tool architecture.
- External backend tools that interact with the AMASS platform to provide analysis results. These external tools include OCRA for contract-based analysis, nuXmv for model checking, xSAP and medini analyze for model-based safety analysis, and V&V Manager for requirements analysis and model checking. They run in background or remotely via OSLC and the user does not interact with them directly.
- Other external tools that can be used for modelling with alternative languages to later import the models into the AMASS platform. These include SAVONA for semi-formal specification and refinement of contracts and SCADE or Simulink for specifying behaviour in their own formal or semi-formal language with the purpose of finally generating the runnable software source code (formally verified in the case of SCADE).
- Similarly, some further external tools can be used to perform external analysis, whose results must be then imported into the AMASS platform. These include RQA for requirements quality analysis, Sabotage for fault injection on Simulink models and AMT2.0 for contract-based offline monitoring of cyber-physical systems. Deliverable D3.6 “Prototype for Architecture-Driven Assurance (c)” [29] describes more in detail the design of the AMASS platform. In particular, Figure 3 of the D3.6 illustrates the interaction among the different components.





**Figure 12.** Internal and external tools involved in the Architecture-Driven Assurance

## 3. Methodological Guide

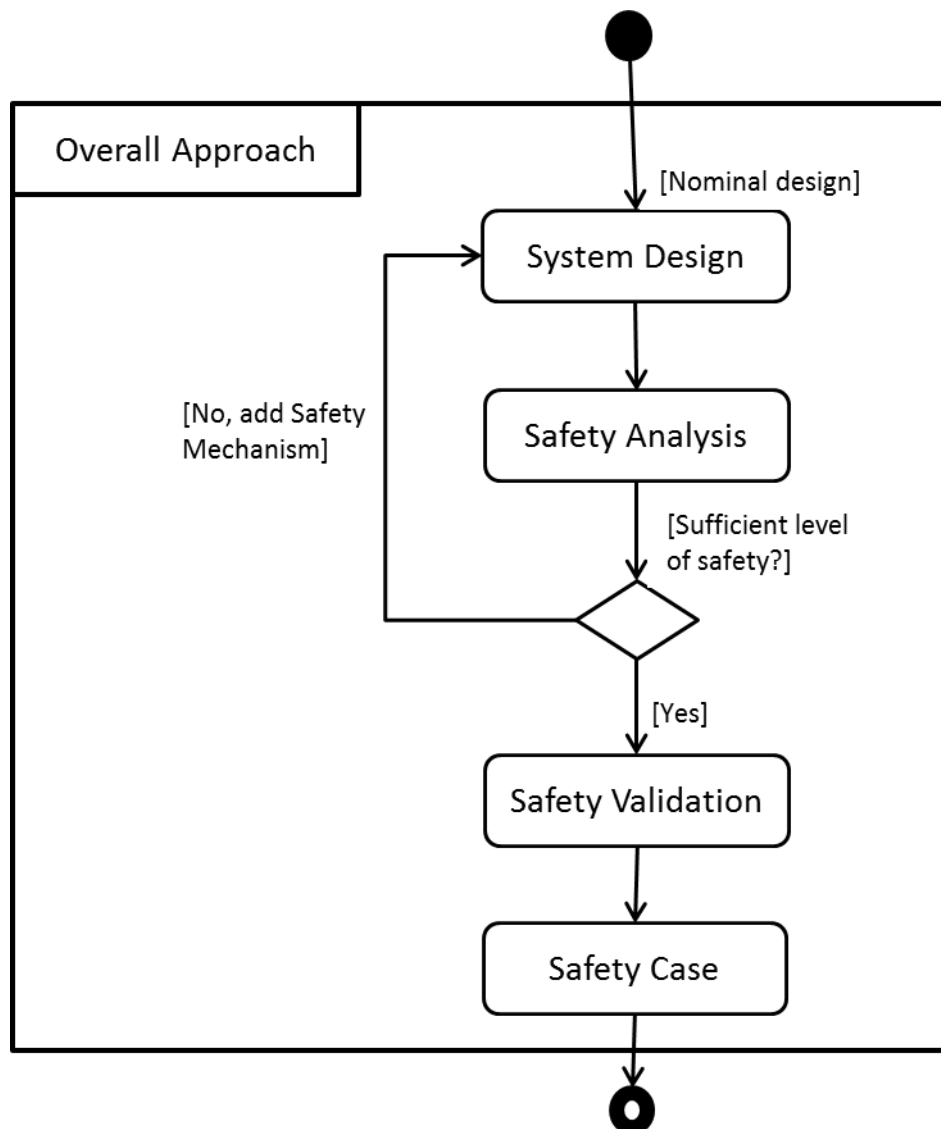
This section contains the actual methodological guide. It first gives an overview of the activities by means of diagrams and then details the activities guiding the user to the usage of the related tool support.

### 3.1 Workflow Overview

The following activity diagrams describe a sequence of steps to perform the design and development of the system with the support of the AMASS platform for Architecture-Driven Assurance.

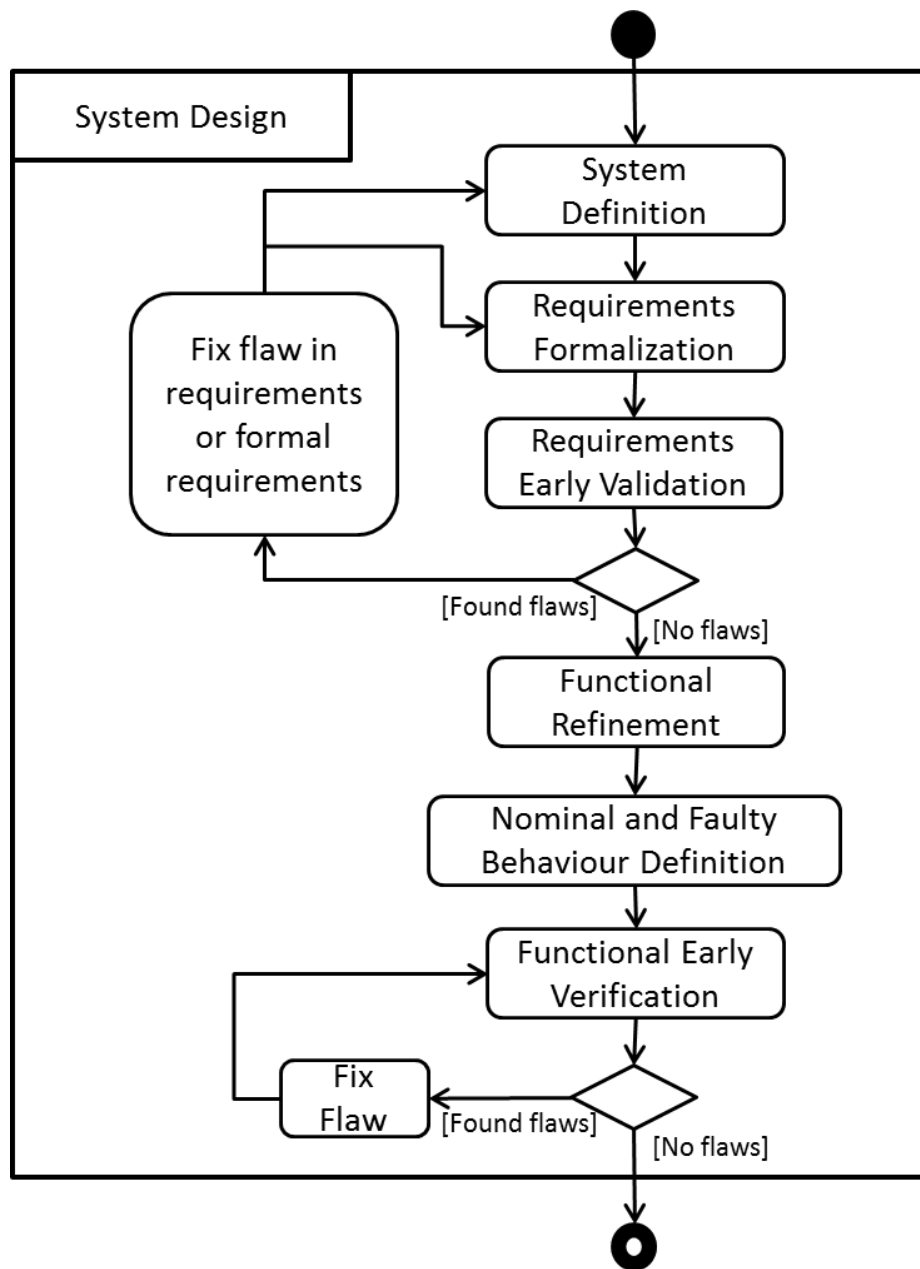
The diagram shown in Figure 13 gives an overview of the main steps of the process. It starts with a first design of the system, including both nominal and faulty behaviours. It then applies safety analysis of the system design to determine if it contains a sufficient level of safety. Otherwise it refines the system design taking into account safety mechanisms to increase the safety level. These two steps are repeated until reaching a sufficient level of safety. The system safety is then validated with traditional testing techniques. They basically correspond to the inner V-cycles with early V&V shown in Figure 5.

Finally, a safety case is produced to show how the product fulfils the safety goals. Each of these activities, apart from the safety validation which is not directly supported by the AMASS platform, is further expanded in subsequent diagrams.



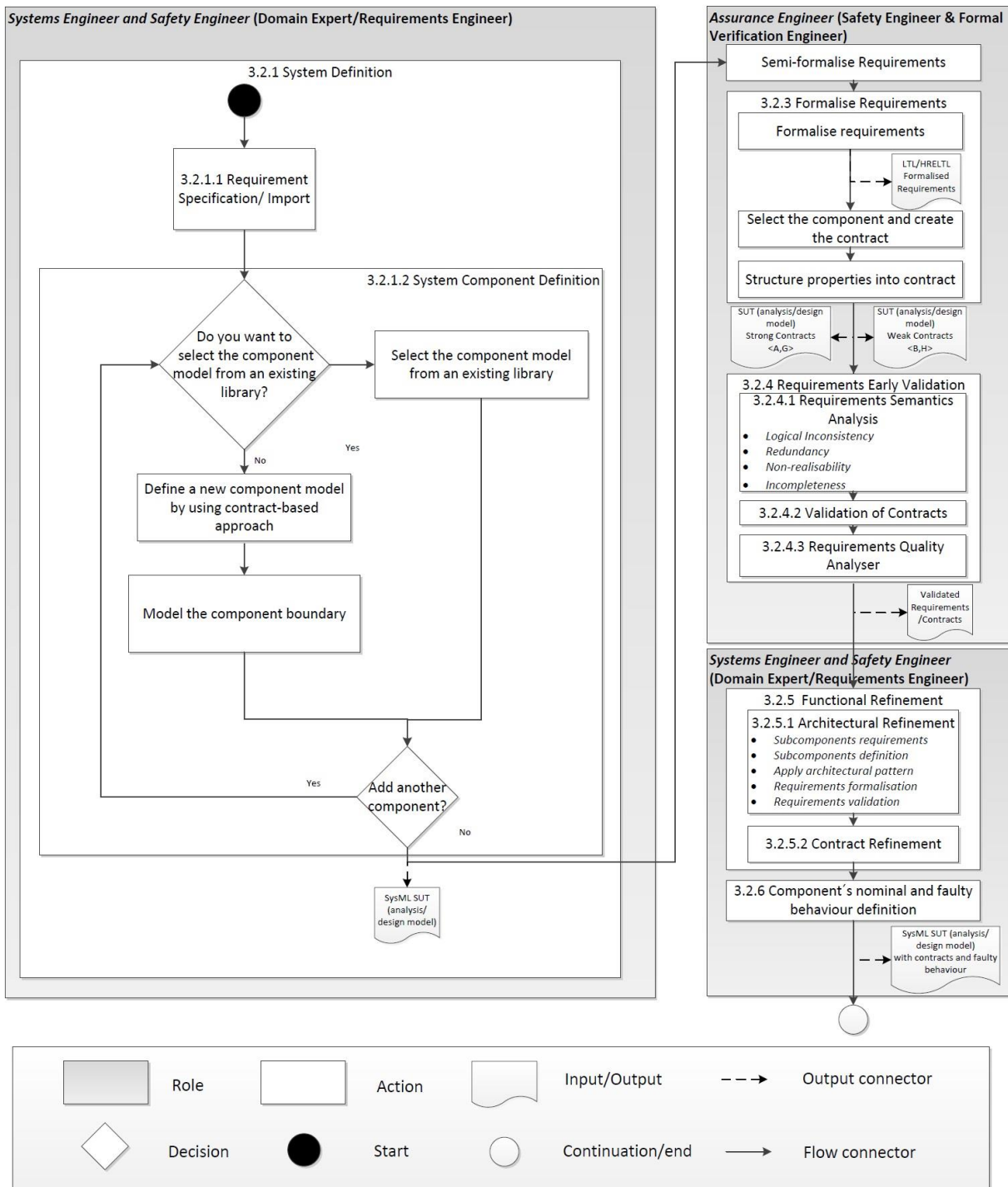
**Figure 13.** Main Steps to perform the design and development of the system with the support of the AMASS platform for Architecture-Driven Assurance

The system design shown in Figure 14 starts with the system definition, the system boundaries, the system requirements and the assumptions on the expected context, also taking into account the results of the preliminary hazards analysis. The design process then formalizes requirements into formal or semi-formal specifications. The process performs an early validation of such requirements by means of different techniques based on either formal semantics or quality metrics. The process then refines the system functions defining the architectural decomposition of components and contract-based refinement. Nominal and faulty behaviours are specified for the leaf components of the architectural decomposition. Finally, an early verification of the functional refinement is performed by means of contract-based reasoning, model checking, and simulation-based monitoring.

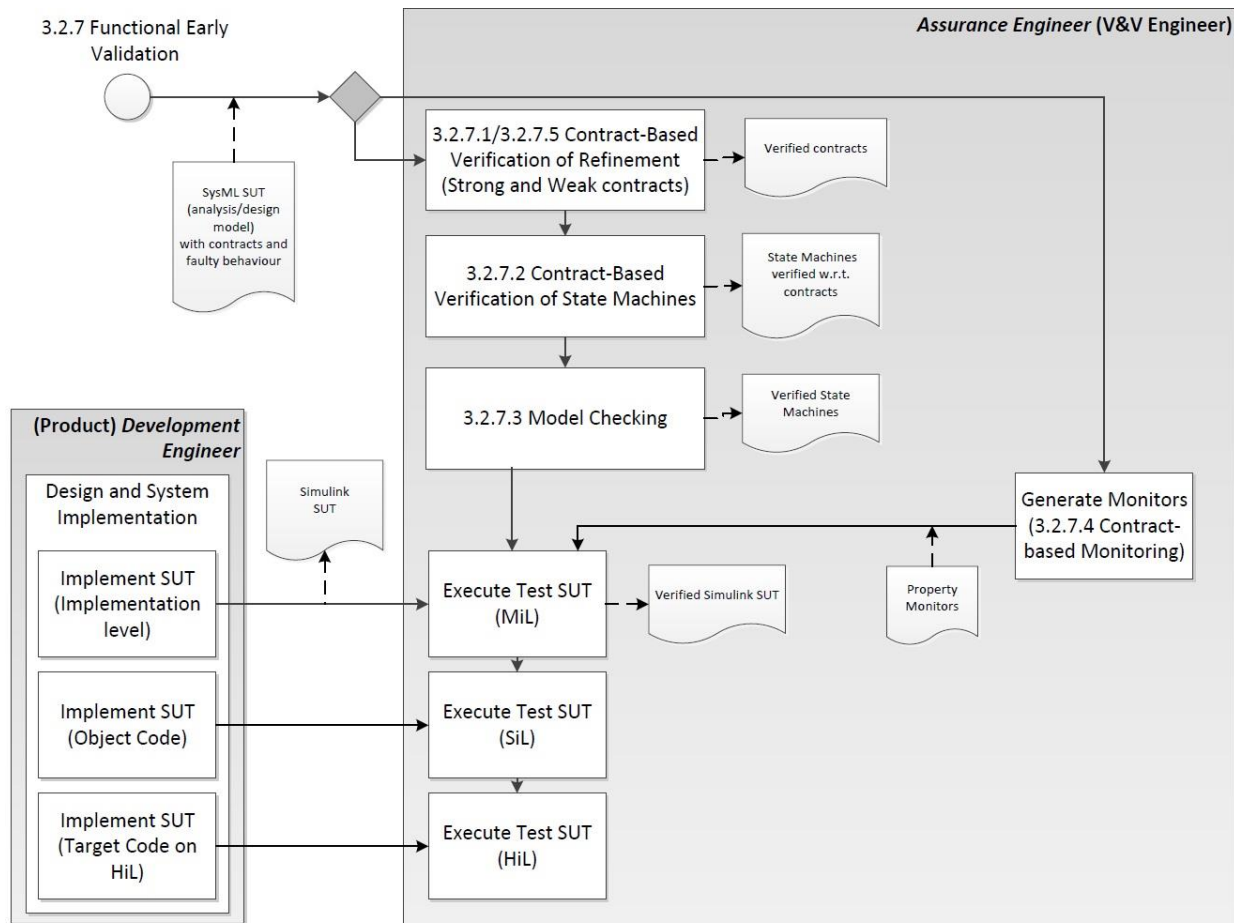


**Figure 14.** Steps for the System Design

Figure 15 and Figure 16 complete the System Design workflow depicted in Figure 14 by adding information regarding definition of roles for each activity. The roles in *italics* define the ones already described in D2.4 [32].



**Figure 15.** Detailed steps for the System Design with the definition of roles (I)



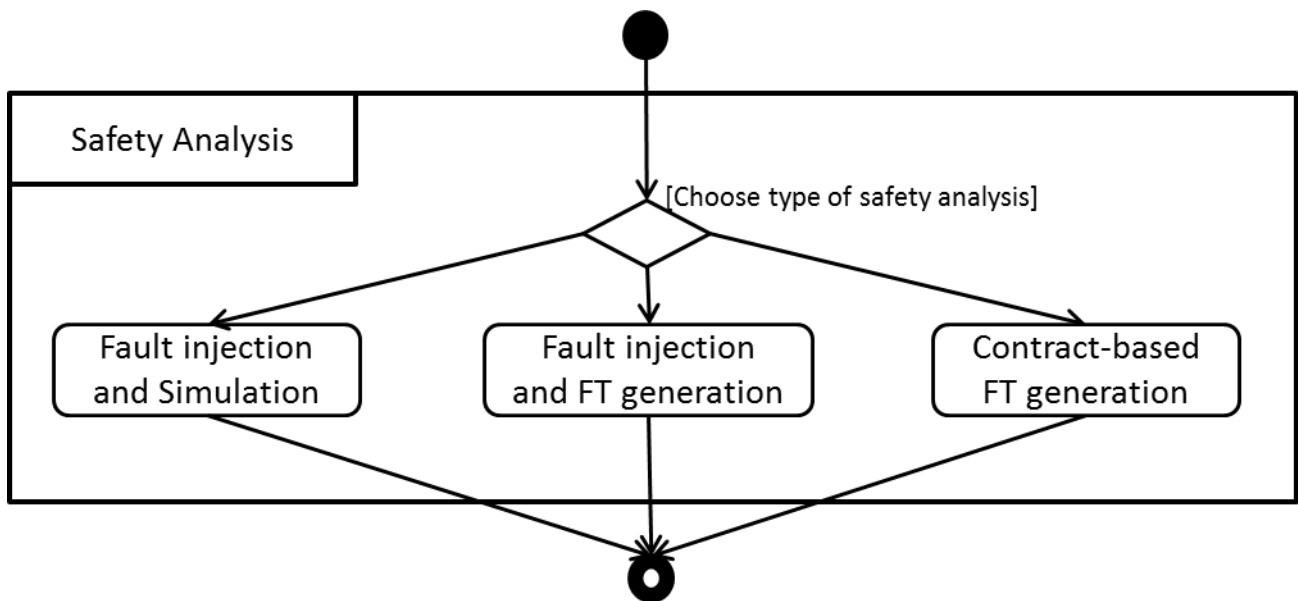
**Figure 16.** Detailed steps for the System Design with the definition of roles (II)

As depicted in Figure 8 and mentioned in Section 2.2.2, after performing the hazard analysis the top-level safety requirements are defined. Afterwards, the causes of those hazards are usually recursively investigated by Fault Tree Analysis, while FMEA technique is applied to analyse the more detailed and more technical architecture at lower levels. Thus, those high-level safety requirements are implemented by safety mechanisms which are then validated by means of the model-based safety analysis techniques tackled in the next paragraph.

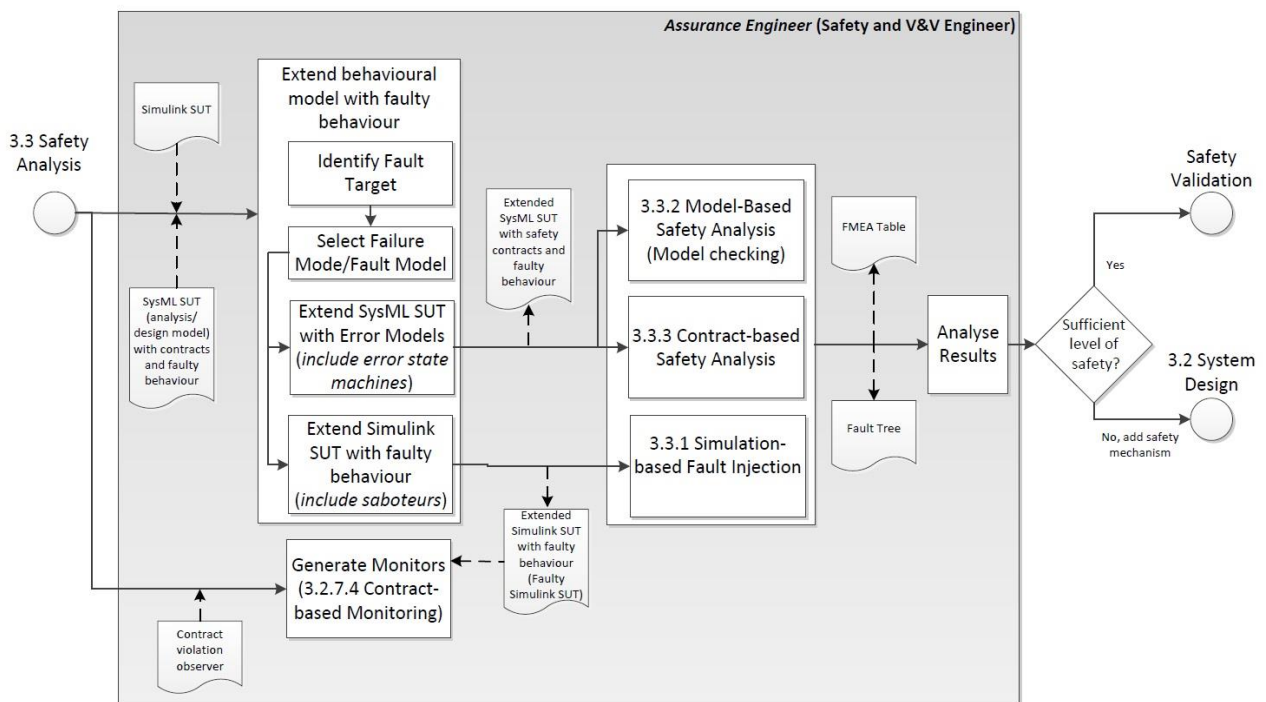
The safety analysis shown in Figure 17 and Figure 18 can apply different techniques to analyse the system when some components may fail. The current supported techniques are based either on:

- the fault injection in the simulation models and the monitoring of the simulation traces,
- the fault injection in the state machines and model-based fault-tree generation,
- the contract-based fault-tree generation.

These are alternative techniques that can complement each other to assess the safety level of the system.



**Figure 17.** Sub-activities related to the safety analysis currently supported by the AMASS platform



**Figure 18.** Detailed Safety Analysis Process

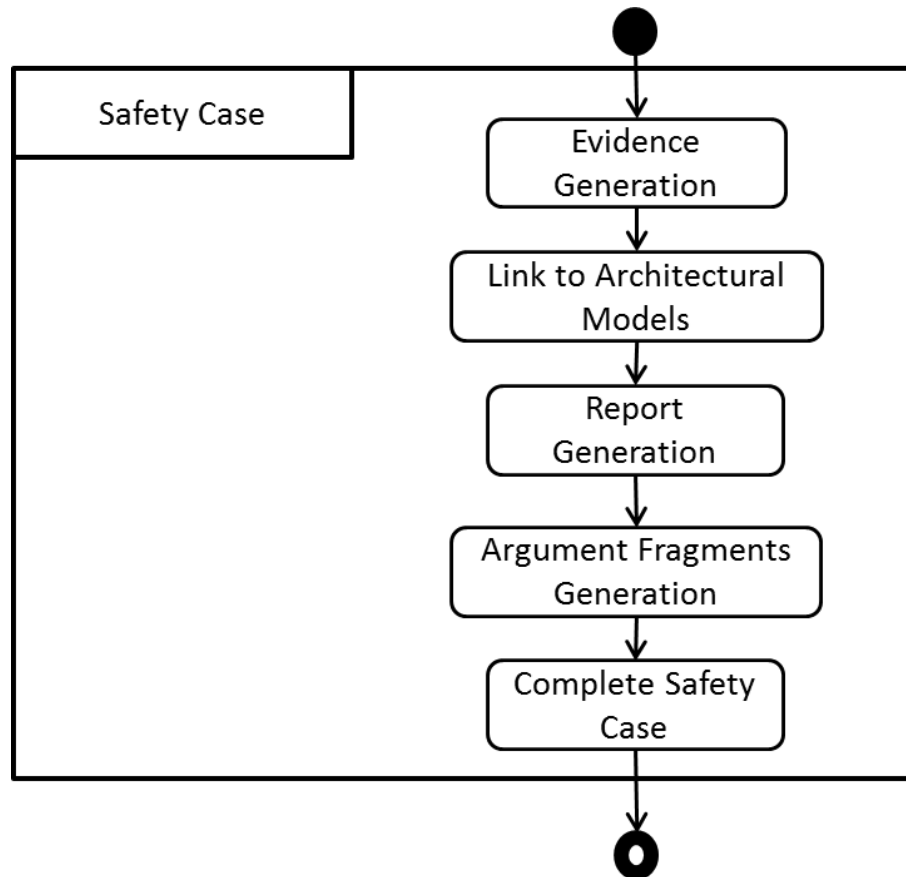
Finally, the activity of producing the safety case, shown in Figure 19, is supported with:

- the generation of evidence from the system model early V&V and safety analysis,
- the linking of the architectural models to the safety case,
- the generation of reports on the system model and V&V and safety analysis results,
- the automatic generation of safety fragments to be integrated and completed in a global safety case.

The Safety Engineer, the Safety Manager and the Internal Assessor are the roles involved in this activity. The Safety Engineer is the responsible for executing the different assurance and V&V activities to demonstrate

that the product is acceptably safe. The Safety Manager is responsible to show compliance within a particular standard. The Safety Assessor is responsible for assessing the adequacy of the evidence and assurance ‘package’, in terms of demonstrating the safety of the system under consideration.

Note that this document focuses only on the architectural-driven support to the safety case. For a more general support to the creation of the safety case, the reader is referred to D2.5 [33].



**Figure 19.** Sub-activities related to the safety case currently supported by the AMASS platform

## 3.2 System Design

In the following, the System Design activities depicted in Figure 14 are detailed, one subsection for each activity.

### 3.2.1 System Definition

#### 3.2.1.1 Requirements Specification/Import

Requirements can be represented in the model by using the SysML standard support, i.e. by using the Requirements Diagram and Requirement SysML construct; no specific extension of the SysML support for requirement modelling is currently proposed as part of the AMASS solution. Of course, specific profiles for SysML regarding the modelling of requirements can be adopted, in particular to allow the modelling of adopted requirements attributes (e.g. the author, kind, priority, risk).

CHESS comes with a dedicated view/package called “RequirementView”, where the SysML Requirement Diagram and then the requirements can be created.



In addition, Papyrus, which is the tool on which CHESS is based, allows importing requirements from external sources<sup>5</sup> like excel, csv files, or by using the ReqIF<sup>6</sup> (requirement interchange format), the latter supported by most of the commonly used requirement management tools, like DOORS<sup>7</sup>.

Requirements available in the CHESS/Papyrus model can then be linked to other architectural entities by using the standard SysML support, so in particular by allocating requirements to components (see section 3.2.1.2) using the Satisfy relationships.

As alternative to the aforementioned usage of SysML to represent and link requirements, Capra can be used to directly trace requirements available in model/repository external to CHESS/Papyrus (like DOORS or ReqIF) to the CHESS system architectural elements; in this case the import of the requirements in the CHESS/Papyrus model is not required and the information about the satisfy relationships between system components and requirements is stored in the Capra dedicated traceability model.

Moreover, requirements can be traced to the assurance case model, the latter available from the OpenCert assurance case editor; for this goal, the feature of Capra<sup>8</sup> can be used: so, for instance, a safety requirement can be linked to the assurance case reasoning about why the system satisfies the requirement itself. In this case, the requirement can be selected from the Papyrus/CHESS model, the assurance case can be selected from the OpenCert editor and a trace can be created with the Capra features (see AMASS user manual, available as annex of the AMASS D2.5 [33], about how to work with Capra in the context of the AMASS platform).

### 3.2.1.2 System Component Definition

During this activity the architecture of the system, i.e. its constituent components and their relationships, is produced. The definition of the system architecture foresees the design of components in isolation, i.e. components designed out of any particular context (to exploit their reuse), or the identification (reuse) of the components from existing libraries/models. The hierarchical modelling of the entire system architecture is supported to be able to manage the complexity of the system itself, where the hierarchy can be obtained by using a top-down or bottom-up design process, where the later in particular can be realized by following a design by reuse approach.

A component must come with typed input/output ports representing the boundary of the component itself, where ports represent interaction points through which the component can exchange information with the context. It is worth noting that the modelling of the component boundary cannot be limited to the identification of the input/output ports of the component itself; additional information about the possible (functional and extra-functional) behaviours of the component and the expected behaviours of the context must be provided, e.g. to be able to evaluate if a given component can collaborate in a given scenario. This relevant part of the specification of the component boundary can be performed by using the contract-based approach, as discussed in Section 3.2.2.

When a component has been defined, then it can be *instantiated* as part of the given system under design and connected to the other instances already identified (via the ports, see Section 3.2.5.1). The system is modelled, i.e. it is decomposed, as a collaboration of instances of components. The decomposition of the architecture can proceed at any level, i.e. a system component can be also decomposed, according to its

---

5

<https://www.eclipsecon.org/france2016/sites/default/files/slides/EclipseConf2016%20sysml%20and%20requirements.pdf>, <https://www.youtube.com/watch?v=edHAXb8-1Io>

6 [www.omg.org/spec/ReqIF/About-ReqIF/](http://www.omg.org/spec/ReqIF/About-ReqIF/)

7 <https://www.ibm.com/software/products/it/ratidoor>

8 Capra is an Eclipse project for traceability management (<https://projects.eclipse.org/proposals/capra>); it has been extended in the context of AMASS.

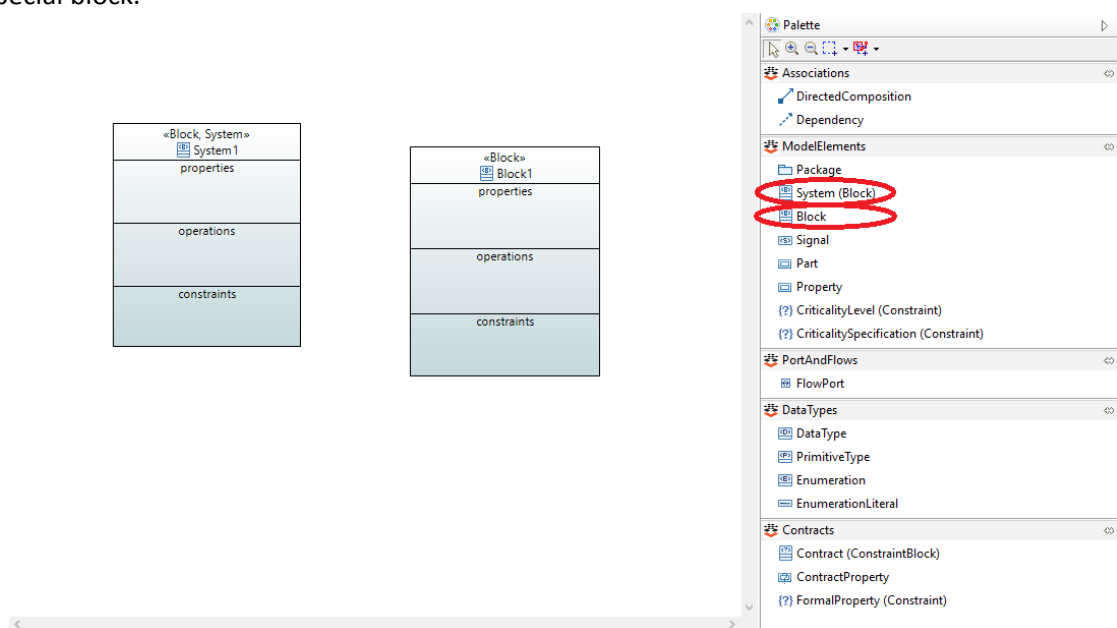
complexity. Different instances of the same component (and different ports with the same type) can be represented as a single instance with multiplicity greater than 1.

SysML [66], the OMG general-purpose modelling language for systems engineering, is proposed in AMASS (by using the Papyrus/CHESS editor) for the modelling of the system and its constituent components. In particular, SysML Block Definition Diagram is used to model the system and its components, while a SysML Internal Block Diagram can be associated to the system or component and used to model its input/output ports, its decomposition into parts and how the parts are connected.

CHESS modelling language (CHESSML)[61] is based upon SysML and UML. This guide focuses on the usage of the SysML language for what regards the system definition; however, the same approach presented here can be followed by using UML diagrams and entities corresponding to the SysML ones here addressed<sup>9</sup> (the choice of using UML instead of SysML is typically dictated by the fact that the system under design is a pure software one). The reader can refer to the CHESS user guide<sup>10</sup> to understand the UML support for software specification available with CHESS.

To define the system components, perform the following steps:

1. Prepare the graphical editor environment: In the “Model Explorer” view, go to the package “modelSystemView”, create a Block Definition Diagram and open it.
2. Define the system components: Using the palette of the diagram editor, select the “Block” element and drag it in the editing space, see Figure 20. The “Block” element becomes visible also in the “Model Explorer” view. Analogously, it is possible to create the “System” element, that is specified as a special block.



**Figure 20.** A generic component and the system component created in the Block Definition Diagram

<sup>9</sup> In UML the user can use Class/Component Diagram (in place of the SysML Block Definition Diagram) to model the software system, its components and the software interfaces (as collection of operations). Composite Structure Diagram can be associated to the system or component (in place of the SysML Internal Block Diagram) and used to model its ports (input/output or service port providing or requiring interfaces), its decomposition into parts and how the parts are connected.

<sup>10</sup> [https://www.polarsys.org/chess/publis/CHESSToolset\\_UserGuide.pdf](https://www.polarsys.org/chess/publis/CHESSToolset_UserGuide.pdf)

### 3.2.2 Hazard Identification

After the system (or “item” in ISO 26262 terminology) has been defined, the first actually safety-specific process step is to identify the hazards of the item in terms of accident risks that the system can cause to humans. This activity is common to all major safety standards, but the naming and the proceeding in detail – in particular which influence factors to be considered when ranking the hazards - are quite different: in automotive ISO 26262, it is named “Hazard Analysis and Risk Assessment (HARA)”, in industrial IEC 61508 it is called “Hazard and Risk Analysis”, in aerospace ARP4761 it is called “Functional Hazard Analysis (FHA)”.

In industrial settings, influence factors like the probability of people being in the danger zone is considered, whereas in automotive applications there is no explicit measure for that (up to now, it can be assumed, that at least one human - the driver – is in the danger zone, in most cases many more). In automotive, however, the probability of exposure to some driving situations is considered as the “E” (for exposure) factor, and the possibility for the driver or other traffic participants to prevent the accident after a failure has occurred is rated by the “C” (for controllability) factor. Also, the severity ratings are different, ranging up to “many people killed” for industrial plants, whereas for passenger cars the highest rating for the “S” (for severity) measure is S3, which translates as “severe or fatal injuries” (of at least one human – catastrophic consequences are not assumed for a single passenger car). The way of arranging, analysing and rating the hazards is slightly different among different standards (in tabular form, in most cases), and so is the way of determining the final rating for each hazard, which may be called SIL (safety integrity level), ASIL (automotive safety integrity level), DAL (design assurance level) or otherwise. These levels are usually ranked on a scale comprising 4 to 6 levels, designated by letters or numbers (e.g. ASIL A is the lowest in ISO 26262 and ASIL D is the highest – whereas in ARP 4761 it is in the opposite sense, A is the most severe and meaning “catastrophic”.) But despite all the differences, all safety standards have in common that hazards are understood as the effects of a system failure (specification violation due to design error or due to part failure occurring at runtime) and a usage or environmental situation (e.g. platoon driving on highway). Also, it is a commonality that the hazards are ranked on a scale according to their risk, which is in some sense a combination of severity and occurrence probability.

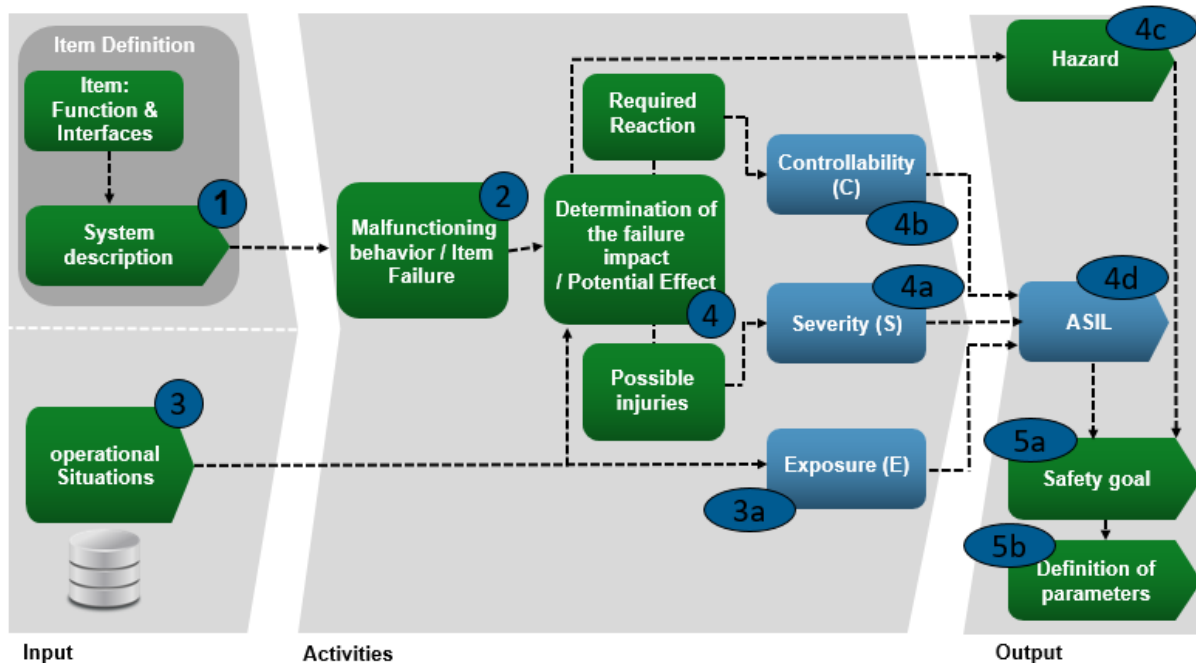
The AMASS workflow and toolchain does not define any novel method for hazard identification, but refers to the standards in place. From the selection of AMASS tools, the tool medini analyze, that comes with different profiles for different industries and standards, is the general recommendation for performing the hazard identification and has been used for some of the case studies in the AMASS project; however, other tools can be used as well. What is important is the export capability for hazards, safety goals and rankings (e.g. ASIL) into the AMASS tool chain, because the hazard identification is the most important starting point for all safety activities, as it determines as an output:

- a) The **hazards**, which are the ultimate failure effects and can therefore serve as top-level events in a Fault Tree Analysis or other type of safety analysis.
- b) The **safety goals**, which are the top-level requirements, with the meaning that a given hazard shall be prevented even in presence of failures, and therefore are the starting point for refinement into safety requirements (or safety contracts, respectively) that add to the normal functional requirements for the product.
- c) The **hazard risk levels or safety integrity levels** (like SIL, ASIL, DAL etc.), which determine the subsequent effort to be spent on (semi)formal specification, implementation according to certain rules and guidelines, and verification, such as depth of testing or application of formal verification techniques for higher hazard risk levels. They are also an important input parameter for setting up the safety plan (selection of methods to be applied, in particular) and the safety case.

In the following, the proceeding is exemplified for an automotive HARA acc. ISO 26262 with the tool medini analyze, based on an excerpt from Case Study 2 (Driver assist function with electric powertrain). To exploit synergies between the AMASS case studies, we were focusing on hazards that can be applied similarly to

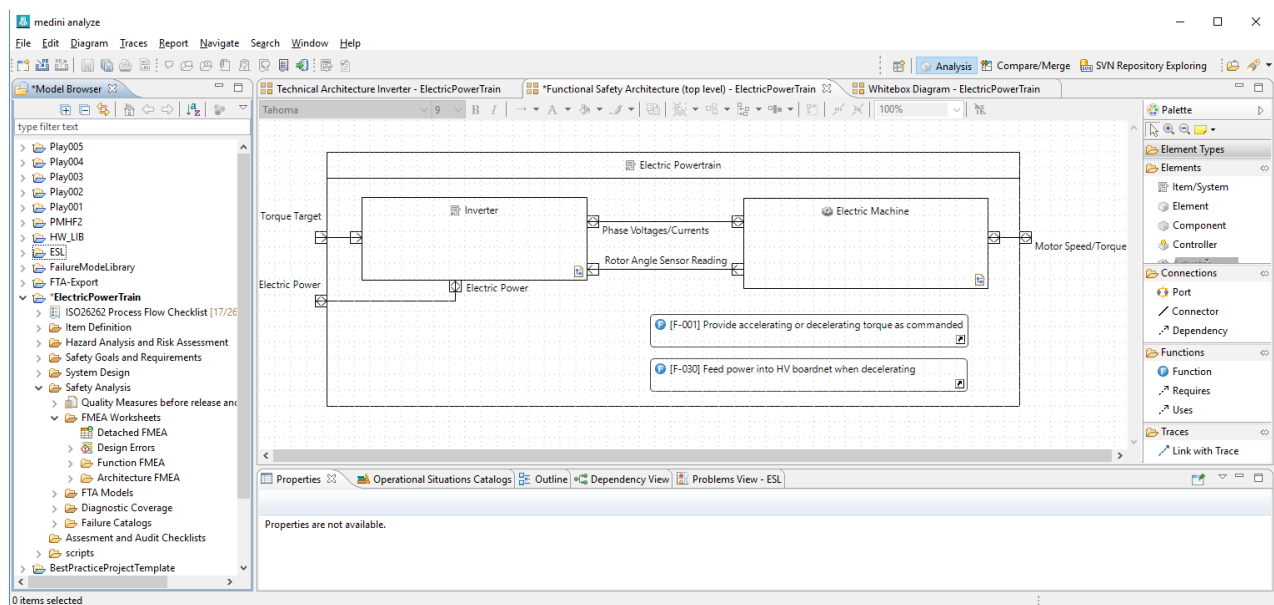
Case Study “DC Drive” and Case Study 3 (Collaborative fleet of vehicles), e.g. “self-acceleration”, “unauthorized braking”, “unauthorized start” or “acceleration in the wrong direction”.

The process of an ISO 26262 HARA can be explained as in Figure 21 (the explanations below refer to the number labels in the diagram).



**Figure 21.** ISO 26262 HARA Process Flow

Input (1) is the System Description from Item Definition Phase (for corresponding AMASS step see 3.2.1), showing the outer interfaces and main functions of the system.



**Figure 22.** Example of system description in SysML

From these, in step (2) the malfunctions are derived. This can be performed in a systematic way by applying the HAZOP method that uses guidewords like “higher than”, “lower than”, “unintendedly”, “other”, “early”, “late”, etc. to modify the intended behaviour to produce a description of potential unintended behaviour

(e.g. function: “control vehicle speed such that the correct distance to the predecessor vehicle is kept” × guideword: “higher than” → malfunction: “the system sets the vehicle speed to a value that is higher than specified”). Note that, the more formal the nominal functions are specified (e.g. using template language for semi-formal assumptions and guarantees), the higher are the chances to semi-automate malfunction derivation (see Section 4.2 of [3]).

HAZOP Analysis					
Function	higher than	lower than	unexpected	not at all	inaccurate/disturbed
[F-001] Provide accelerating or decelerating torque as commanded	[MF-001] Excessive Torque applied to wheels	[MF-002] Insufficient Propulsion	[MF-003] Unexpected Torque applied	[MF-004] No Propulsion provided	[MF-005] Torque ripples
[F-002] Provide Speed Actual Signal to other vehicle functions	[MF-007] Too high speed reported to external safety-critical function	[MF-008] Too low speed reported to external safety-critical function		[MF-032] No speed signal provided to other vehicle functions	[MF-033] Distorted or inaccurate speed signal provided to other vehicle functions

**Figure 23.** Example of HAZOP Analysis

To judge the potentially resulting accidents, it is necessary to discuss the malfunction in different environmental contexts or usage scenarios, e.g. driving in city traffic with vulnerable road users nearby, manually driving on a highway at high speed, automatically driving on a highway as part of a platoon at moderate speed, waiting at a red signal with pedestrians crossing in front of the vehicle, etc. This requires the presence of a catalogue of situations (3) as an auxiliary input. The process of creating and managing this catalogue is often underestimated, because when combining all road types with all speed ranges and weather conditions and all light conditions and all types of manoeuvres and all kinds of other traffic participants around etc., one will usually be overburdened by the combinatorial explosion, resulting in ten thousand cases to be discussed for each of the malfunctions. We can expect that this will get even worse for automated vehicles in the future, as they require to discuss not only malfunctions due to failures, but also due to limitations of the nominal functionality (termed “SOTIF” for “Safety of the intended functionality”), and to judge this, even more parameters of the environmental situations are of relevance, e.g. pose, clothing colour and movement speed of a pedestrian (perhaps one out of many that are present in the scenario), that is supposed to be detected by the automated driving function (e.g. to trigger a braking maneuver). A lot of experience and advanced tool support is required to keep the number of situations manageable, e.g. by forming equivalence classes or by exploiting subsumption relationships or mutual exclusion relationships (e.g. driving at 200 km/h should normally not occur in city scenarios).

In the next step (4), the effects of the malfunctions are determined for certain environmental and usage scenarios, which finally leads to the definition of the hazard (4c), e.g. malfunction: “the system sets the vehicle speed to a value that is higher than specified” × usage scenario “driving on a highway as part of a platoon” → potential immediate effect: “too low distance to predecessor vehicle” → potential hazard<sup>11</sup>: “rear-end collision with predecessor vehicle”. This is today a manual and usually table-based process. In the future, simulation together with fault-injection and monitoring techniques (which is one of the main goals of AMASS) could automate a part of this work.

<sup>11</sup> ISO 26262 makes a fine distinction between “hazard” and “hazardous event” – only the latter being the combination of a malfunction and an operational situation. In practice, this distinction is seldom made, and to avoid confusion we will also not make this distinction, as it has no relevance for the AMASS approach.

ID	Operational Situation	Exposure	Exposure Comment	Malfunctioning Behaviour	Hazard	Potential Effect	Severity	Severity Comment	Controllability	Controllability Comment
HE-001	Driving on Highway (dry road)	E4	every day usage situation	[MF-001] Excessive Torque applied to wheels	Self-Acceleration	Hitting another motor vehicle from behind	S2	When normal distances are kept on a highway, the crash from behind will be with moderate speed difference and a car will protect the people inside to a certain degree	C1	Acc. to EM power and expected vehicle mass, the acceleration will be moderate and can be controlled by moderate braking (for the average, but not for every driver)
HE-002	Driving on Highway (dry road)	E4	every day usage situation	[MF-002] Insufficient Propulsion						
HE-003	Driving on Highway (dry road)	E4	every day usage situation	[MF-003] Unexpected Torque applied	Self-Acceleration					
HE-004	Driving on Highway (dry road)	E4	every day usage situation	[MF-004] No Propulsion provided						
HE-005	Driving on Highway (dry road)	E4	every day usage situation	[MF-005] Torque ripples						
HE-006	Driving on Highway (dry road)	E4	every day usage situation	[MF-006] Start in wrong direction						
HE-007	Driving on Highway (slippery road)	E3	highways are well maintained and get slippery at most 10% of the time	[MF-001] Excessive Torque applied to wheels						

**Figure 24.** Example of scenario analysis

Associated with step (4) is the rating of the hazard in terms of ASIL. The ASIL is calculated out of the factors:

- E for exposure (3a), which depends only on the usage scenario and can therefore be provided along with the situation catalogue.
- S for severity (4a), which depends on the type of injuries that can be expected to any humans – inside or outside of the car – when the described type of accidents happened. These must be manually ranked by experts from S0 – no injuries – to S3 – severe injuries, survival not probable.
- C for controllability (4b), which rates the probability that the driver or other affected people (e.g. pedestrians, that could in some cases jump aside) can prevent the accident, ranking from C0 – normal control operations that every driver does all the time, like slightly adjusting the speed to C3 – almost uncontrollable. Note that for automated driving functions that allow the driver to perform side tasks, or that run vehicles in close distance to each other in a platoon, C3 must be assumed in most cases.

From these factors, the ASIL can be automatically calculated (4d) according to the Table 4 in ISO 26262-3:2018, e.g. S2, E4, C3 → ASIL C.

The remaining steps are (5a) the naming of a corresponding safety goal (= top-level safety requirement), normally by just adding “prevent” to the hazard name (hazard: “self-acceleration” → safety goal: “prevent self-acceleration”) and (5b) the specification of additional parameters, e.g.

- Safe state (e.g. in case of failure leading to a self-acceleration → switch propulsion off completely).
- Fault Tolerant Time Interval (e.g. in the range of 100 ms to 500 ms for typical powertrain-related failures).
- Defining parameters (e.g. self-acceleration is any acceleration generated by the electric powertrain which is greater than the driver’s demand by more than 4 m/s<sup>2</sup>).

The output of the HARA are the rated hazards that will serve as root events for safety analysis and the rated safety goals that will be refined into more detailed safety requirements (or safety contracts) according to the further steps of the AMASS methodology.

### 3.2.3 Requirements Formalization

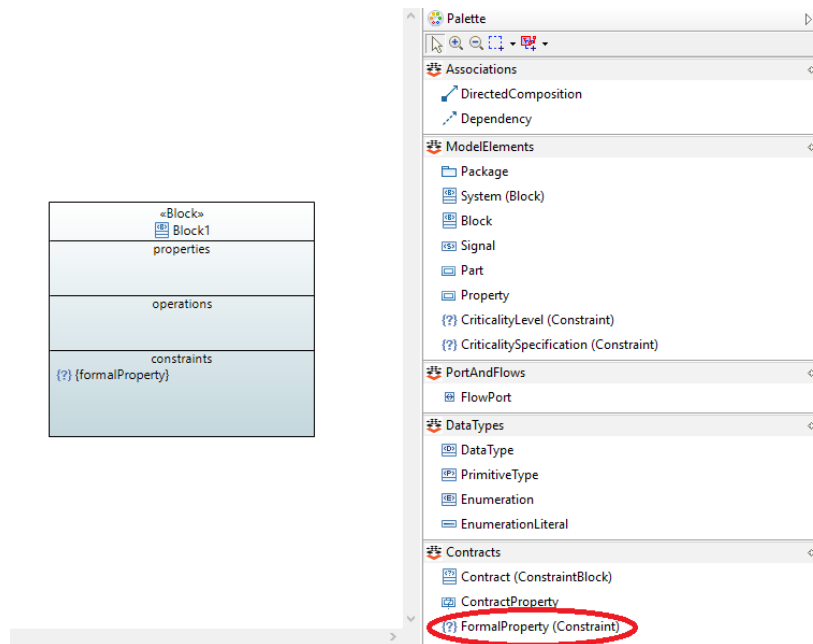
#### 3.2.3.1 Requirements Formalization in CHES

This activity has the goal of translating the informal requirements allocated to system components into formal properties. Formal properties are textual expressions specified in a restricted grammar so that their semantics is formally defined (in rigorous mathematical terms). Such expressions refer to the elements of the

system component interface defined in the previous activity. The AMASS platform supports the specification of well-formed formal properties with the help of an automated completion feature that suggests the possible continuations of a partial specification.

In order to formalize a requirement previously allocated to a component, perform the following steps:

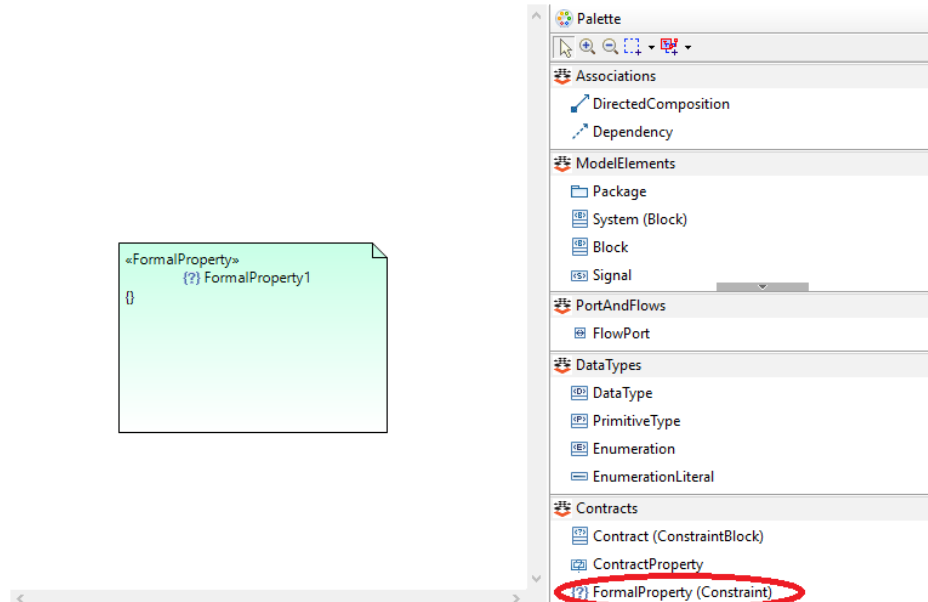
1. Create a formal property associated to a component: Navigate the elements of the model using the “Model Explorer” view, and create a new diagram (class diagram or block definition diagram) or open an existing one. Using the palette of the diagram editor, select the formal property element and drag it inside a component already created, see Figure 25. The formal property element becomes visible also in the “Model Explorer” view. The same result can be achieved following Step 2 and 3.



**Figure 25.** Formal property created in the diagram editor. The property is associated to one component

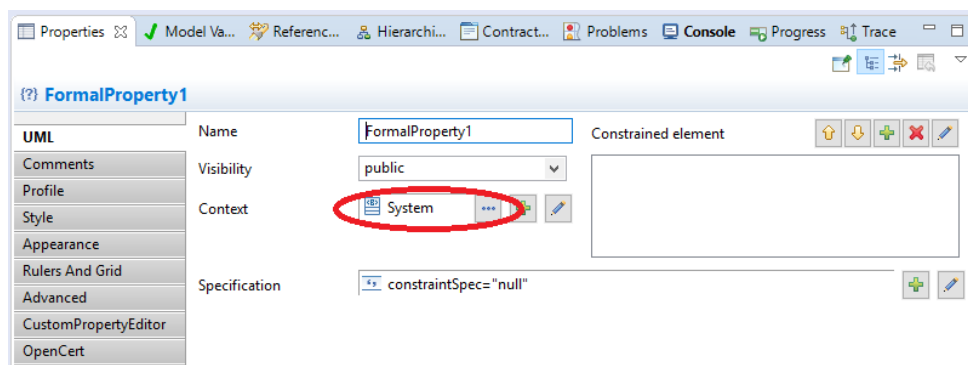


2. Create a formal property: Navigate the elements of the model using the “Model Explorer” view, and create a new diagram (class diagram or block definition diagram) or open an existing one. Using the palette of the diagram editor, select the formal property element and drag it in the editing space, see Figure 26. The formal property element becomes visible also in the “Model Explorer” view.



**Figure 26.** Formal property created in the diagram editor. The property is not associated to any component

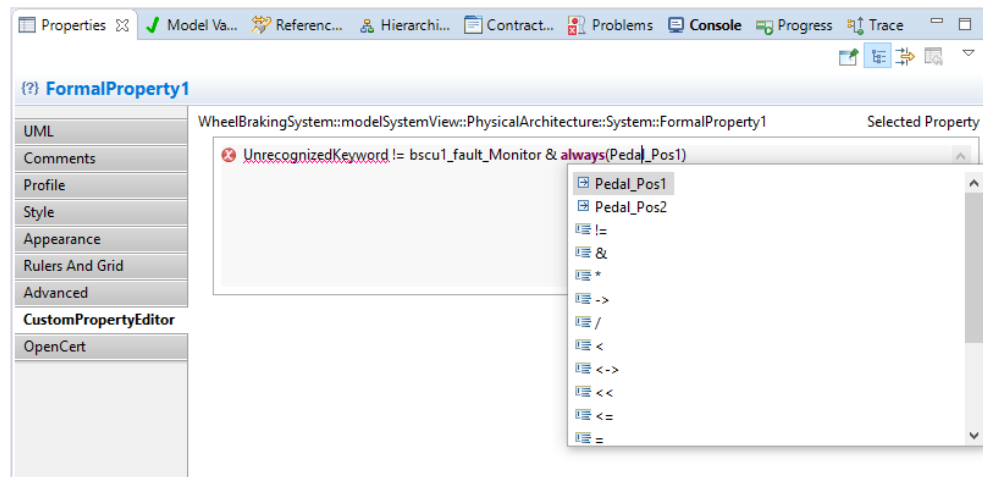
3. Set the owner component of the formal property: Select the formal property (in the “Model Explorer” view) or the corresponding graphical representation (in the diagram editor) and open the tab “UML”. In the “context” area, browse the model and set the owner component of the formal property, see Figure 27. This step is not mandatory but is required to enable the “completion assistance”, see step 4.



**Figure 27.** Owner component set in the UML tab



4. Edit the formal property: Open the tab “CustomPropertyEditor” in the “Properties” view tab. Write the formal property in LTL. By pressing CTRL + SPACE it is possible to have a list of supported keywords of the LTL to define the formal property. If the owner component is set correctly (see step 2), the “completion assistance” can suggest also the input/output ports and the attributes of the owner component to type. Moreover, it notifies whether a typed word does not belong to a keyword/port/attribute, see Figure 28.

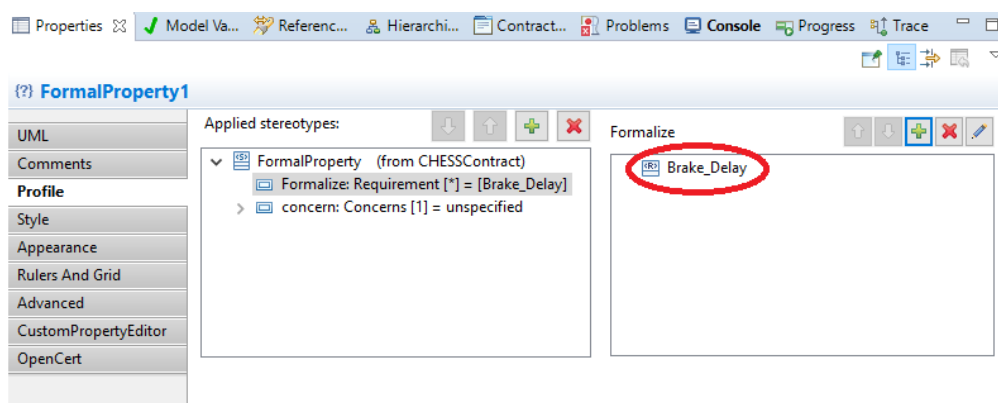


**Figure 28.** Property Editor with “completion assistance”

Note that in case the formalized requirements are not available in the CHESS model, Capra traceability tool can be used to create a trace between the FormalProperty and the requirements available in the external source.

Note that, according to the previous steps, the formalized requirements should have been allocated to the current component (see section 3.2.1.1).

5. Link the requirement to the formal property: Open the tab “Profile” in the “Properties” view tab. In the area “Applied stereotypes” select “Formal Property” – “Formalize”. Then, browse the model and set one or more requirements that are formalized by the current property, see Figure 29.

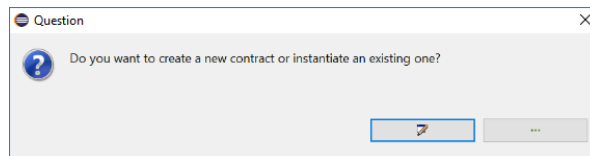


**Figure 29.** Requirements set in the Profile Tab

Formal properties can be further structured into contracts, performing the following steps:

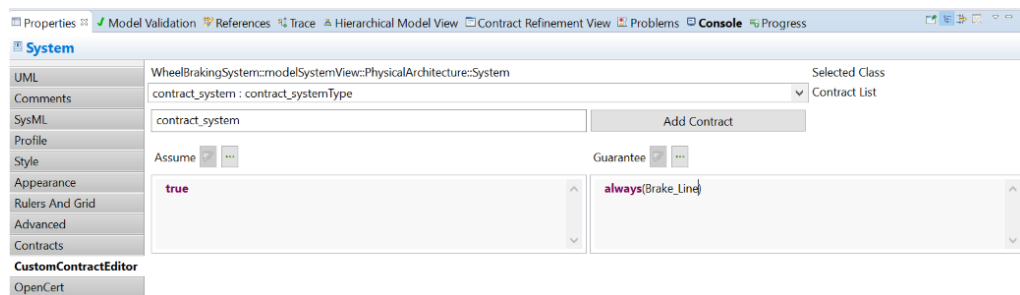
1. Select the component to assign the contract. The element can be selected in the “Model Explorer” view or in the graphical editor.
2. Create the contract. Open the tab “custom property editor” in the “Properties” view, type the name of the contract in the “new contract's name” area and click the “Add Contract” button. A popup

appears to choose if a new contract must be created or if an existing one must be instantiated, see Figure 30.



**Figure 30.** Popup related to the contract definition process

3. Edit the contract. In the tab “CustomPropertyEditor”, it is possible to assign existing formal properties to the assumption and guarantee fields of the contract or create new ones. Then it is possible to edit the properties with “completion assistance”, see Figure 31.

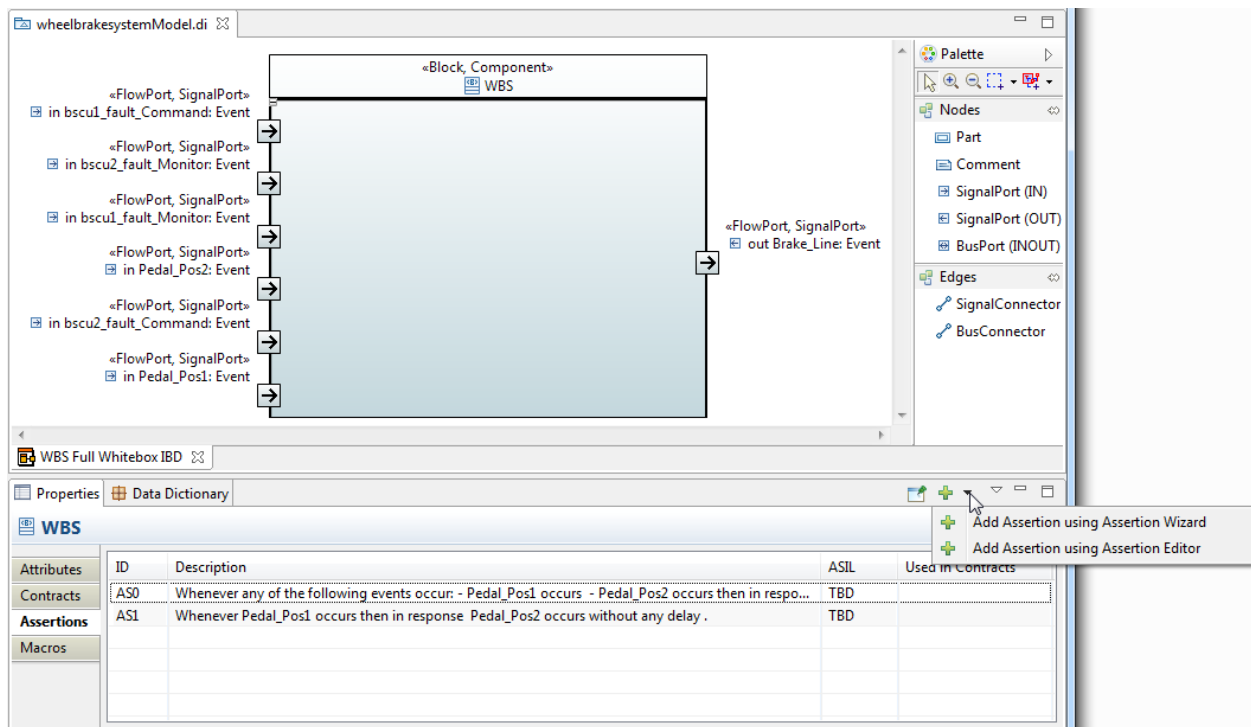


**Figure 31.** Contract editor with “completion assistance”

### 3.2.3.2 Requirements Formalization Using SAVONA

As users might not be familiar with formal expressions to define contracts, SAVONA supports a custom text-based editor with syntax checks and auto-completion, and a wizard to set up assertions with pre-defined templates for the most common assertion patterns.

In SAVONA *Assertions* are the equivalent to CHESSE’ *Formal Properties* which can later be used as *assumptions* or *guarantees* within a contract. Assertions are bound to a model element, so the system component which shall be specified needs to be selected either in the Model Explorer or the Diagram Editor. The “Assertions tab” of the Properties View shows all defined assertions of the selected model element. To create a new assertion either use the right click context menu of the Assertion Table or press the green “Add”-button on the upper right-hand corner of the Properties View (see Figure 32).



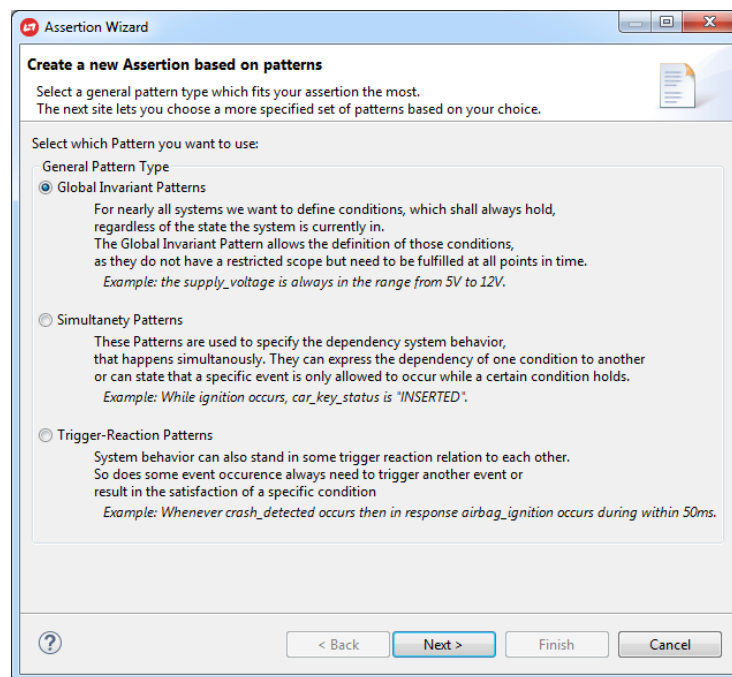
**Figure 32.** Assertion section of the properties view in SAVONA

Assertions are defined in a constrained natural language called System Specification Pattern Language (SSPL)[71]. Using this semi-formal language guarantees certain qualities of the built expressions like high readability, unambiguity and the option for automated verification.

As it cannot be expected that the user either knows SSPL or can write such expressions free-hand SAVONA offers two options to define assertions with tool guidance which will be described in the following paragraphs.

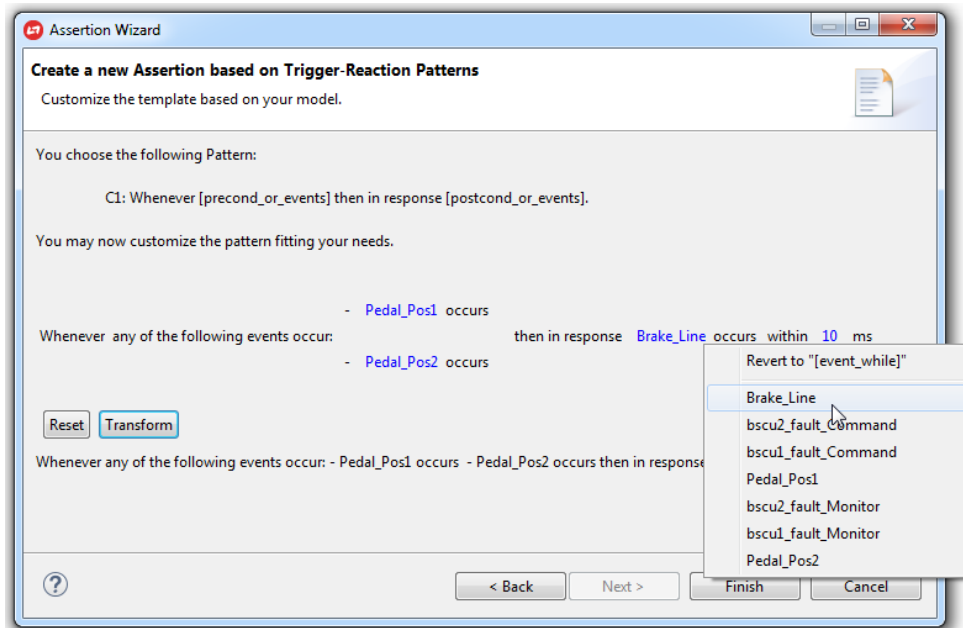
### Assertion Wizard

As applying a template language can be quite difficult without any guidelines, SAVONA features a wizard that guides the user through the process of choosing and filling out an appropriate pattern structure for their system's specification. The first page of the wizard shows the three main pattern types of SSPL: Global Invariant Pattern, Simultaneity Pattern, and Trigger-Reaction Pattern (see Figure 33). A short description and a usage example for each one is provided.



**Figure 33.** Assertion-Wizard: Selection of a General Pattern Type to formulate an assertion

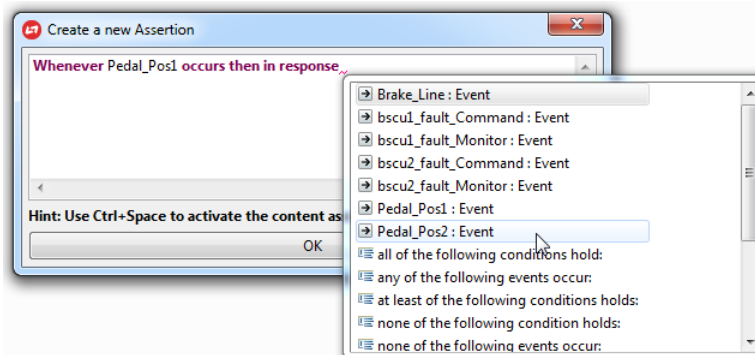
After selecting the main pattern type, several possible pattern instances of the type are presented to the user. Each of them features an example to demonstrate a possible application. If an appropriate pattern instance is chosen, the user will be directed to the last page of the wizard, where the patterns construct needs to be customized. The user can now replace non-terminals by simply clicking on them. A drop-down menu shows possible substitutions and the option to use a macro. If a terminal that must be replaced by an event name is selected, a list containing all event interface names of the currently selected component appears (see Figure 34). That way the user can only choose and use model elements that are in scope. The same holds for terminals that must be replaced by variable names except that the suggested names come from all available ports except the event ports. A set of time units is provided to the user can choose from when specifying timed behaviour. Only if no non-terminals remain in the pattern instance and all terminals are replaced by actual interface names, values, units, etc., can the assertion be assigned to a selected component. Otherwise, the wizard will give a hint to the user about the remaining non-terminals or terminals.



**Figure 34.** Assertion-Wizard: Refine the pattern instance with names of available model elements

### Assertion Editor

If the user has already gathered some experience with our template language, the use of the Assertion Wizard might include too many unnecessary steps to formulate a valid assertion. The right pattern structure is already known by the user, so going through the wizard seems inefficient. The Assertion Editor allows the user to directly type in the desired assertion. As writing valid assertions free-hand can be difficult and error-prone, the tool supports an online syntax check and suggestions for auto-completion of the statement, as one might expect from various programming IDEs. Figure 35 shows the Assertion Editor suggesting valid possibilities to continue the current statement.

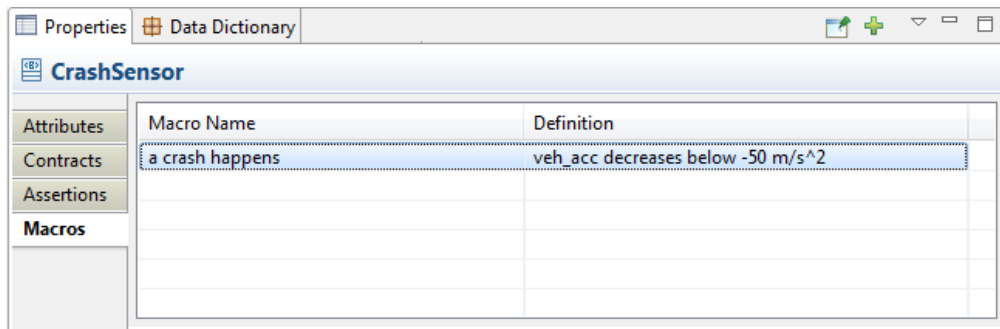


**Figure 35.** Pattern-suggestion feature of the Assertion Editor

#### 3.2.3.2.1 Macro Definition

Sometimes it is unavoidable to use complex expressions within a pattern language, where a natural language expression would be much shorter or easier to read and understand. That is why we introduce the concept of Macros, which allows the use of natural language expressions within our pattern language. The user defines a meaning for each natural language phrase by specifying a corresponding pattern language expression. This way we ensure that even with natural language elements, all built expressions within our pattern language have unambiguous semantics.

As Macros are used to create assertions, they can be created on the same types of model elements. To add a new macro, click the Add-Button on the upper right-hand corner of the Macros Section (see Figure 36).



**Figure 36.** Macros Section of the Properties View in SAVONA

In the wizard, a keyword of the template language is selected for which a macro shall be defined. Additionally, macro name must be defined.

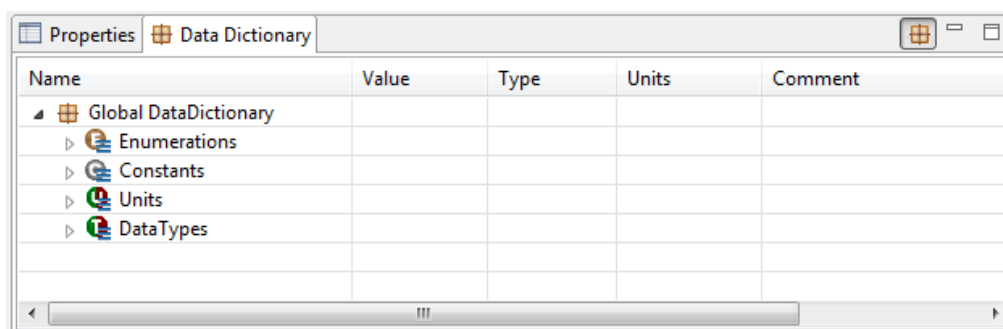
Macros can only replace a non-terminal from the (semi-) formal syntax, as the semantics are only guaranteed to be specified on that level. Terminals (such as port names) can have different meanings due to their context and can therefore not be used as a macro definition.

The subsequent macro wizard pages allow the customization of the selected keyword. Therefore, the same page layout is used as the Assertion Wizard.

### 3.2.3.2.2 Data Dictionaries

When specifying (semi-)formal assertions, there needs to be a way to define custom variables such as constants or units. SAVONA offers a Data Dictionary View (see Figure 37), where several model elements can be defined that can later be used within the definition of assertions:

- **Enumerations** and **Enumeration Members** can be defined
- **Constants** of a certain type (Integer, String, etc.) with or without a Unit
- **Units** (a predefined set of Units is available from start-up)
- **Datatypes** to use as types on ports (a predefined set of Units is available from start-up)



**Figure 37.** Data Dictionary View in SAVONA

Each SAVONA project/model features one **Global Data Dictionary** whose entries are available through the entire project. Additionally, each component type (block) has its own **Local Data Dictionary** whose entries are only available to this exact same component and its owned ports.

### 3.2.4 Requirements Early Validation

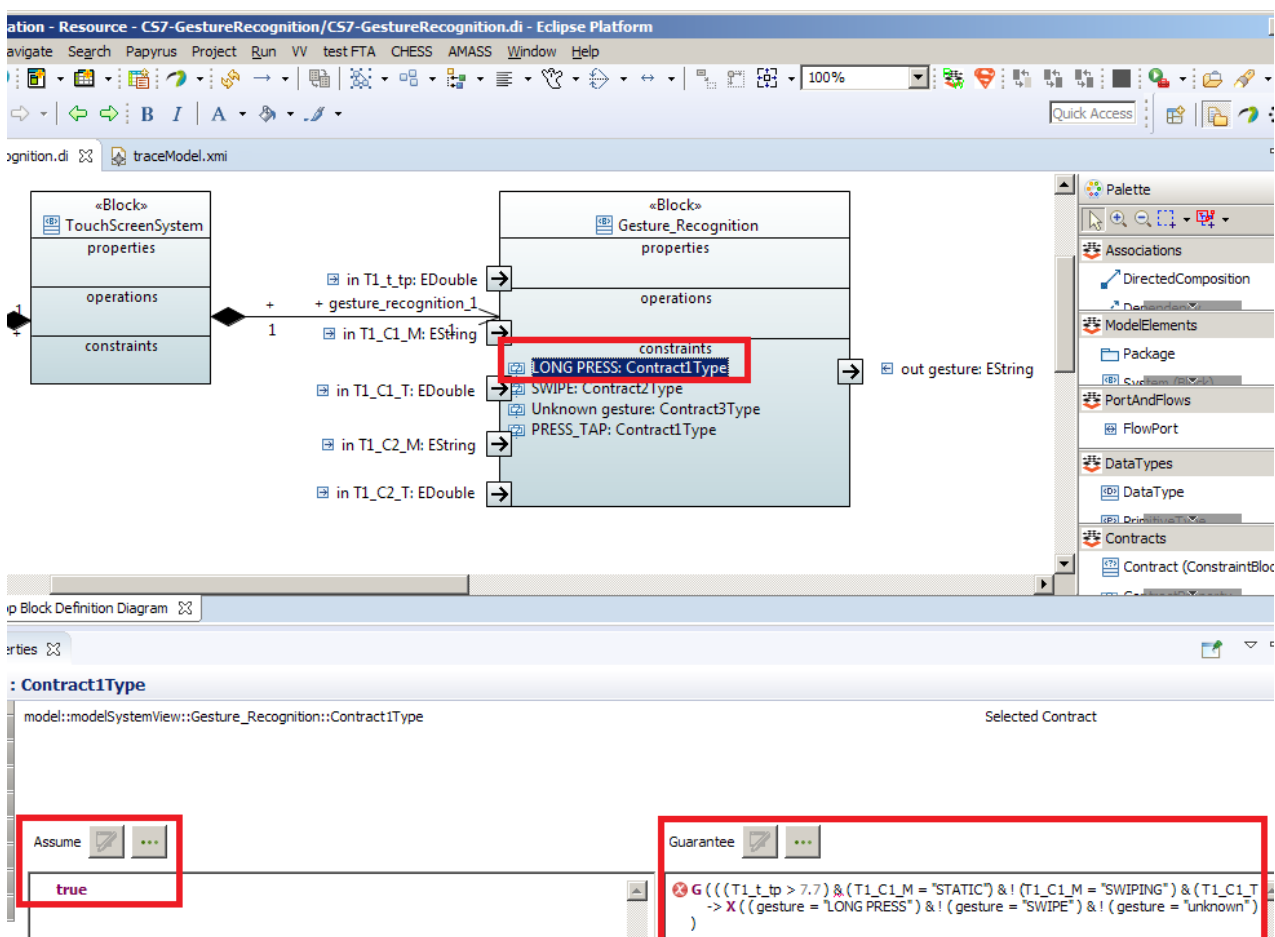
#### 3.2.4.1 Requirements Semantic Analysis

The goal of this activity is to increase requirement quality by detecting and removing requirement defects. Currently detected requirement defects are logical inconsistency, redundancy, and non-realizability.

A set of requirements is considered to be **logically consistent** when no subset of requirements is contradictory. If the set is inconsistent, then the tool **Remus** is used to identify the minimal inconsistent subsets, i.e. the sources of the inconsistency. A requirement is **redundant** if it is implied by another requirement. In such a case, the redundant requirement is reported to the user. The redundancy is currently checked using the **SPOT** tool. Requirements are **realizable** by a non-trivial system and relatively complete when a system can be created that satisfies all requirements, does not restrict any input on top of the restrictions already introduced by the requirements, and no output could remain constant forever from the very beginning. The realizability checking is implemented using the tool **Acacia+**. All these background tools are integrated on verification servers and are controlled by V&V Manager using OSLC.

This activity assumes that the requirements are formalized already as described in the Section 3.2.2.

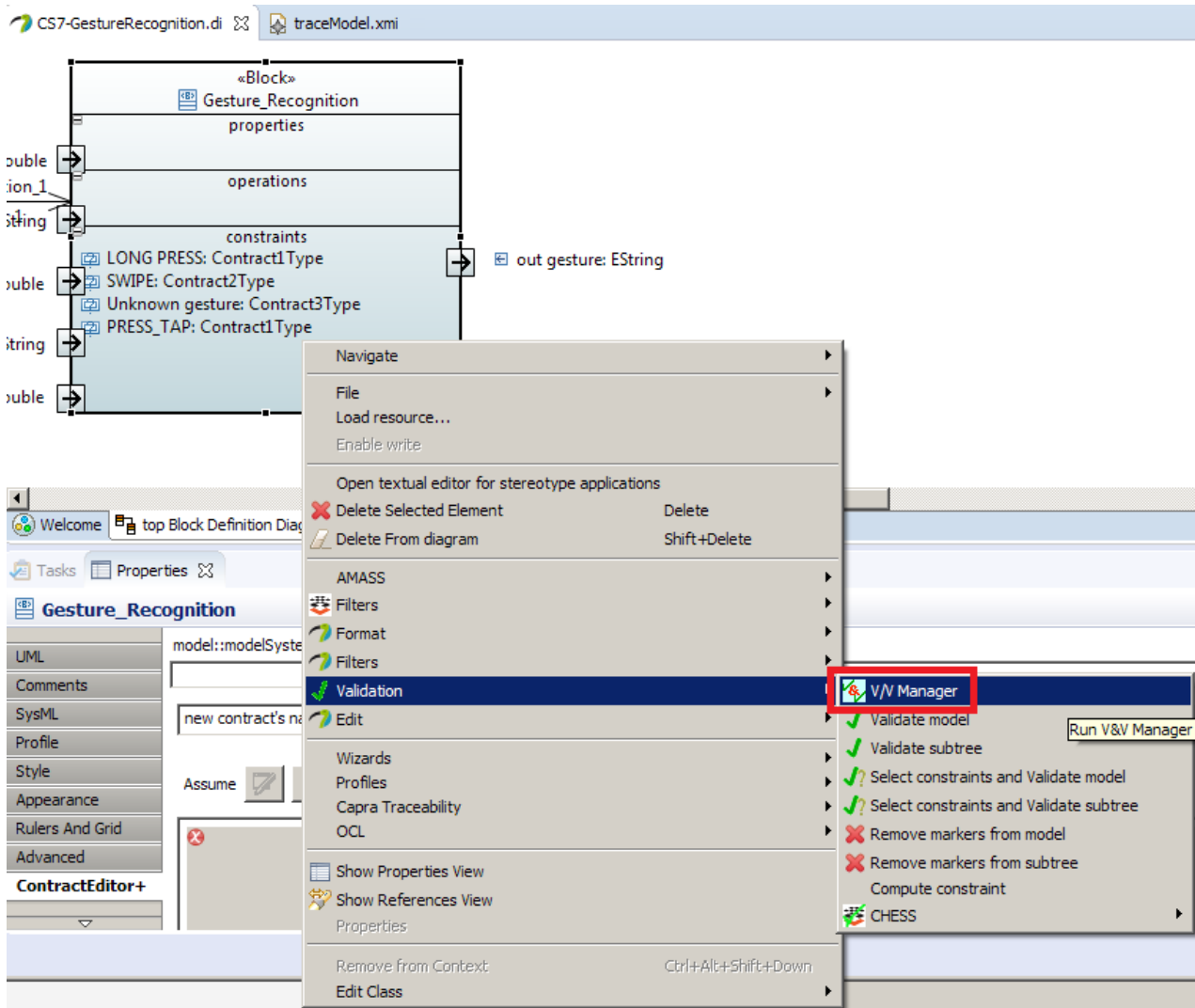
The formalized requirements are stored as the formal assumptions and guarantees in the contracts, see Figure 38.



**Figure 38.** Formal assumptions/guarantees in the contract of a component

In order to submit these formal properties to the appropriate V&V tool(s), the user selects a set of the formal properties and sends them through the V&V Manager to the connected tools. The selection of the formal

properties is performed manually e.g. in the Block Definition Diagram, either at the level of individual properties, or at the level of contracts, where all formal properties related to the contract are included in the set, or similarly at the level of components, via the related contracts. The validation is invoked from a contextual menu as depicted in the Figure 39.



**Figure 39.** Invocation of the V&V Manager

After the validation tools return the results of their semantic analysis to the V&V Manager, these results will be presented in a “V&V Result” view to the user, see Figure 40. The user can examine the results and remove any detected defects of the formal properties.

V&V Manager supports contracts with formal properties only. Formal properties are supported in Linear Temporal Logic (or some expressively equivalent Bounded First-Order extension). While some V&V tools have restrictions and support for example only subset of LTL, the V&V Manager in general supports it completely.



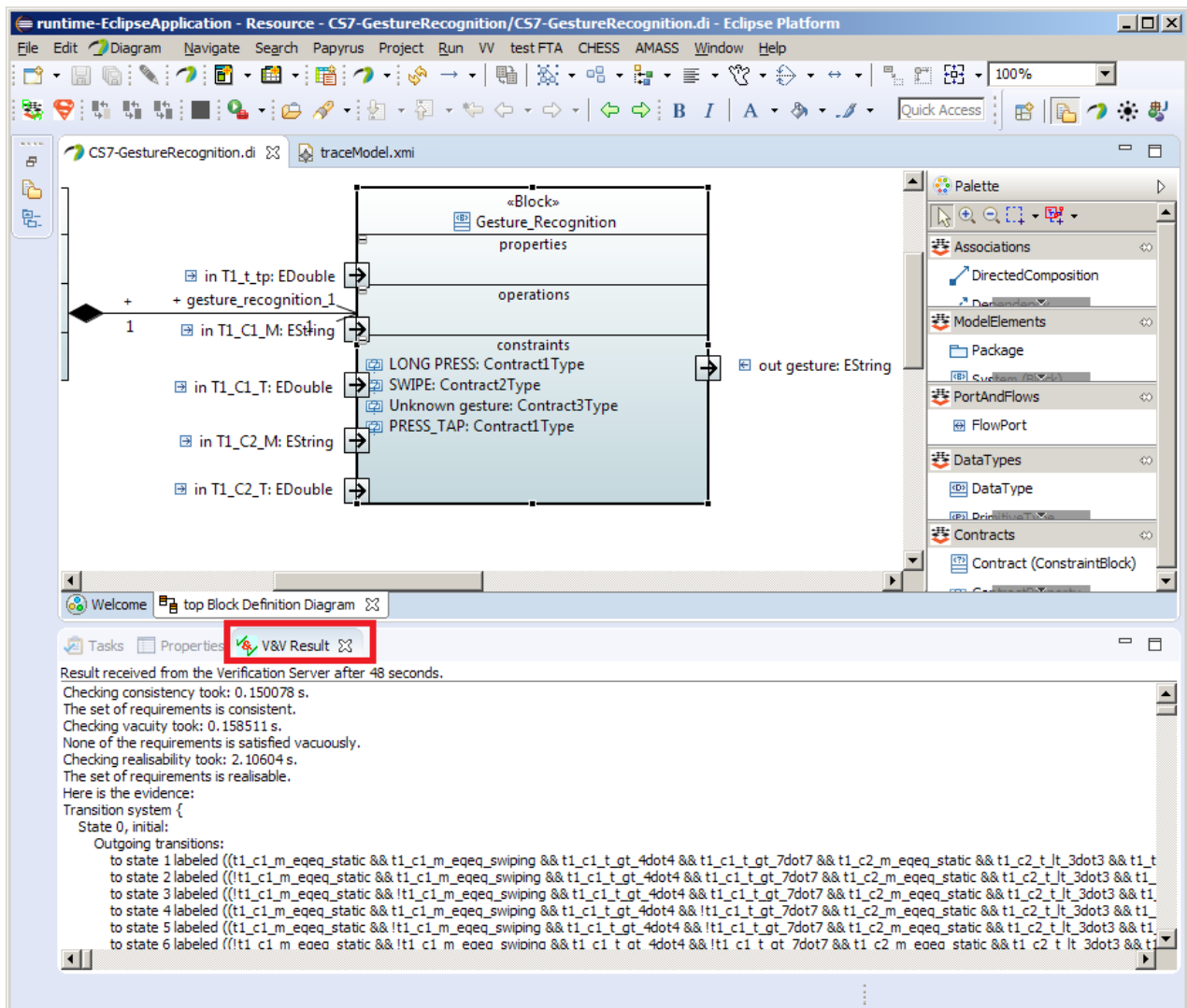


Figure 40. V&V Result view

### 3.2.4.2 Validation of contracts

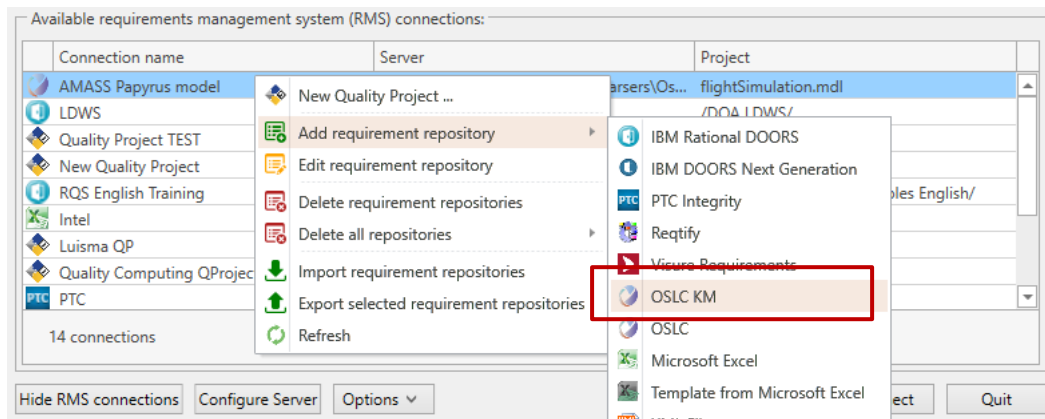
This validation is done by checking if a specific guarantee of the contract satisfies the assumption of another contract. To verify the contract property, perform the following steps:

1. Choose the component that owns the property to check: Select a component (in the “Model Explorer” view) or the corresponding graphical representation (in the diagram editor). The properties available to check will be the assumptions and guarantees of contracts owning to the selected component and to its sub components. This operation includes recursively all the properties from the root to the leaves of the selected component.
2. Perform the validation: right click on the selected component, then go to “AMASS” – “Validation” – “Check Validation Property” on the Selected Component. A popup appears to set the parameters of the command (see [16] for more details).
3. Receive the result of the validation: when the check is completed, the status of the check is shown in the “Trace” view.

### 3.2.4.3 Requirement Quality Analyser in the AMASS platform

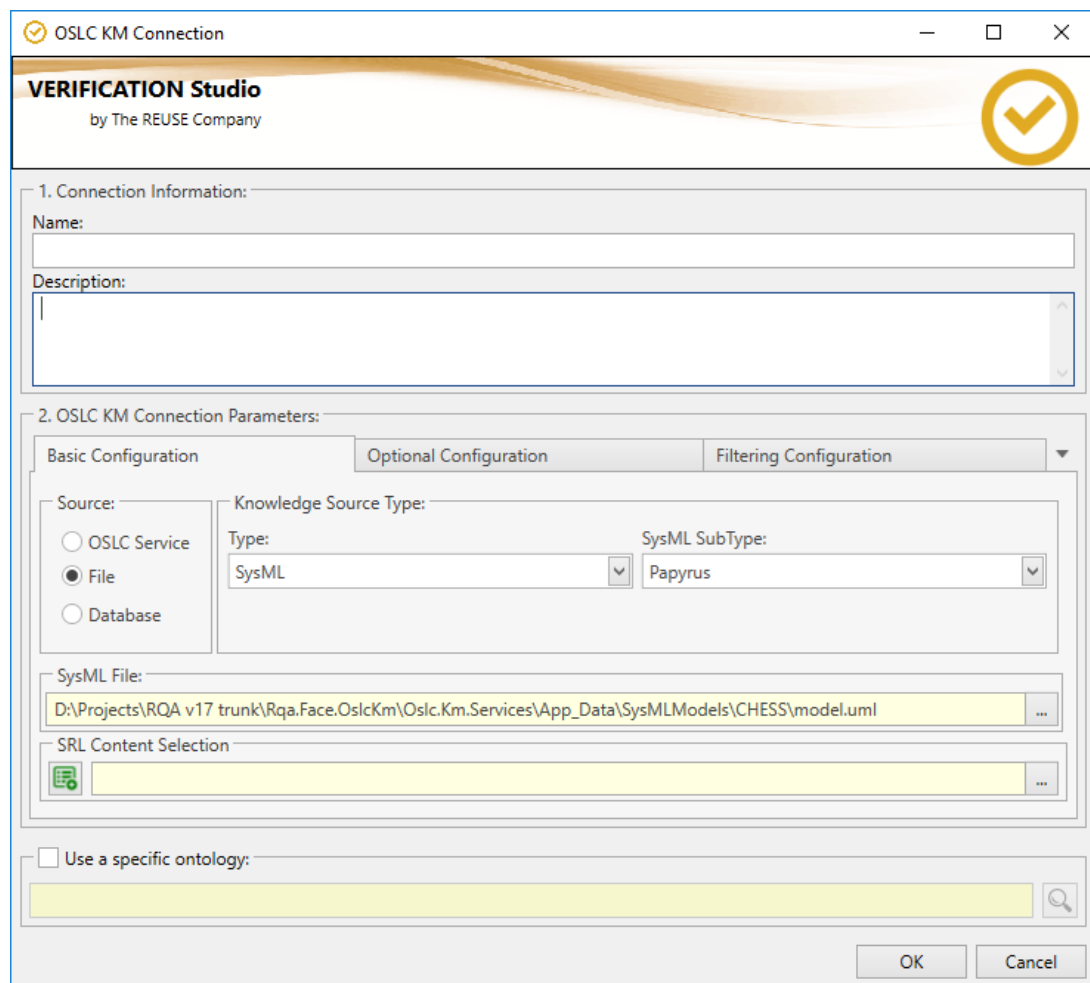
The tool Requirement Quality Analyser (RQA) implements connectors to different environments providing to the AMASS platform evidences of the quality of the projects. These evidences are the quality assessed by the quality metrics included in RQA. Following, the different steps to connect to a CHES project, assess the quality of the model and store the evidence in AMASS are detailed:

1. Create a new OSLC-KM connection in RQA to reference the CHES project. Figure 41.



**Figure 41.** RQA Connection Window

2. In the OSLC-KM Connection window select the project and complete the rest of configuration values. Figure 42 (to detail information of the different parameters see the deliverable D5.6 “Prototype-for-seamless-interoperability (c)” [31] section 2.2.3.5).



**OSLC KM Connection**

**VERIFICATION Studio**  
by The REUSE Company

1. Connection Information:

Name:

Description:

2. OSLC KM Connection Parameters:

Basic Configuration    Optional Configuration    Filtering Configuration

Source:

☐ OSLC Service

☒ File

☐ Database

Knowledge Source Type:

Type: SysML

SysML SubType: Papyrus

SysML File:

D:\Projects\RQA v17 trunk\Rqa.Face.OslcKm\Oslc.Km.Services\App\_Data\SysMLModels\CHES\model.uml

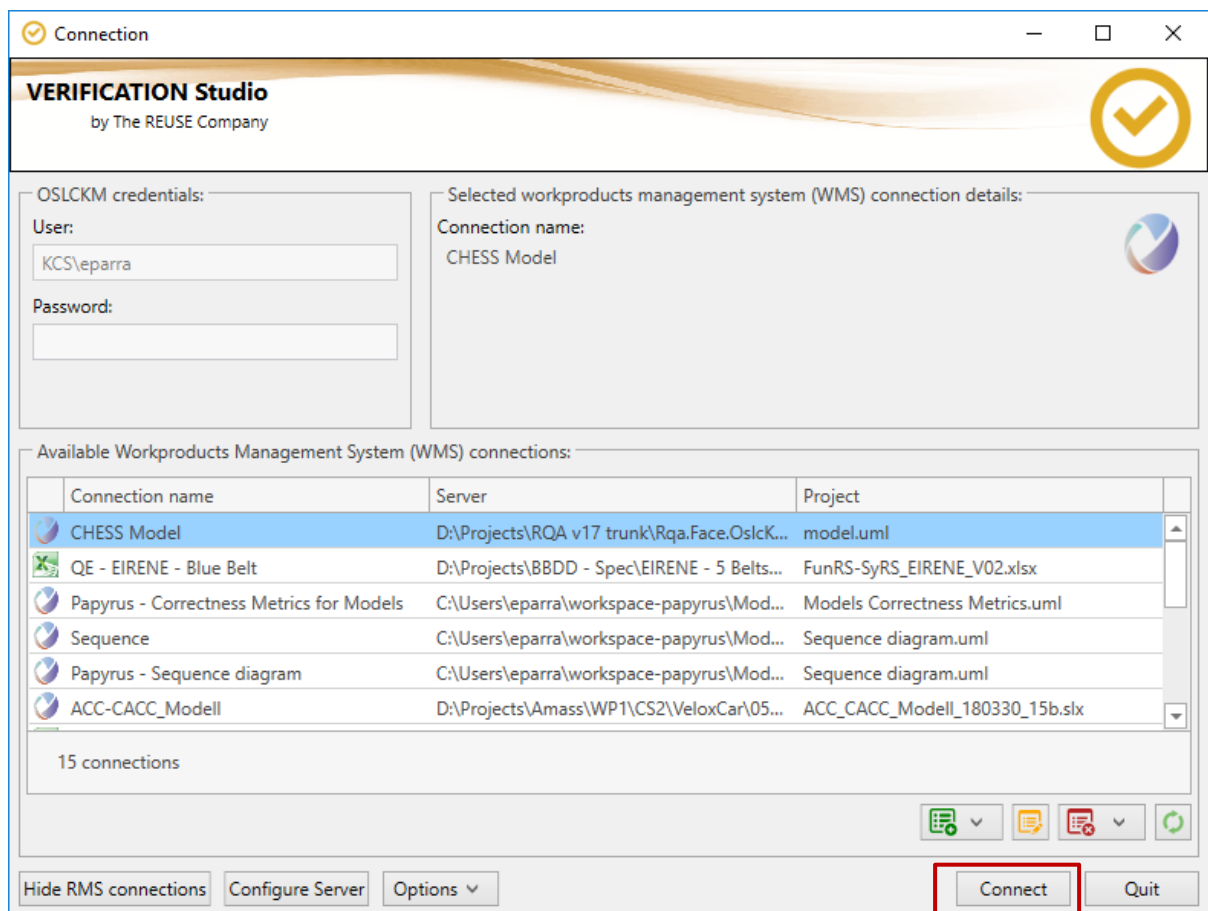
SRL Content Selection

☐ Use a specific ontology:

OK Cancel

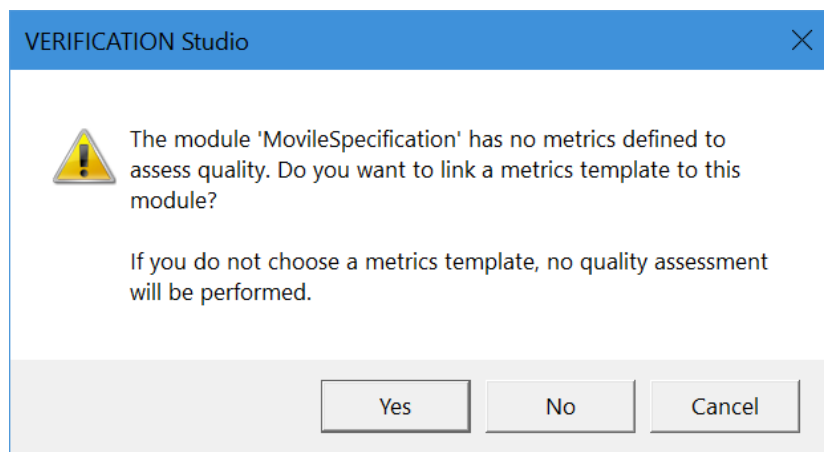
**Figure 42.** OSLC-KM Connection (SysML CHES sub-type)

- When the configuration is created it is possible to connect to the project and the artefacts of the model are imported to RQA tool (see Figure 43).

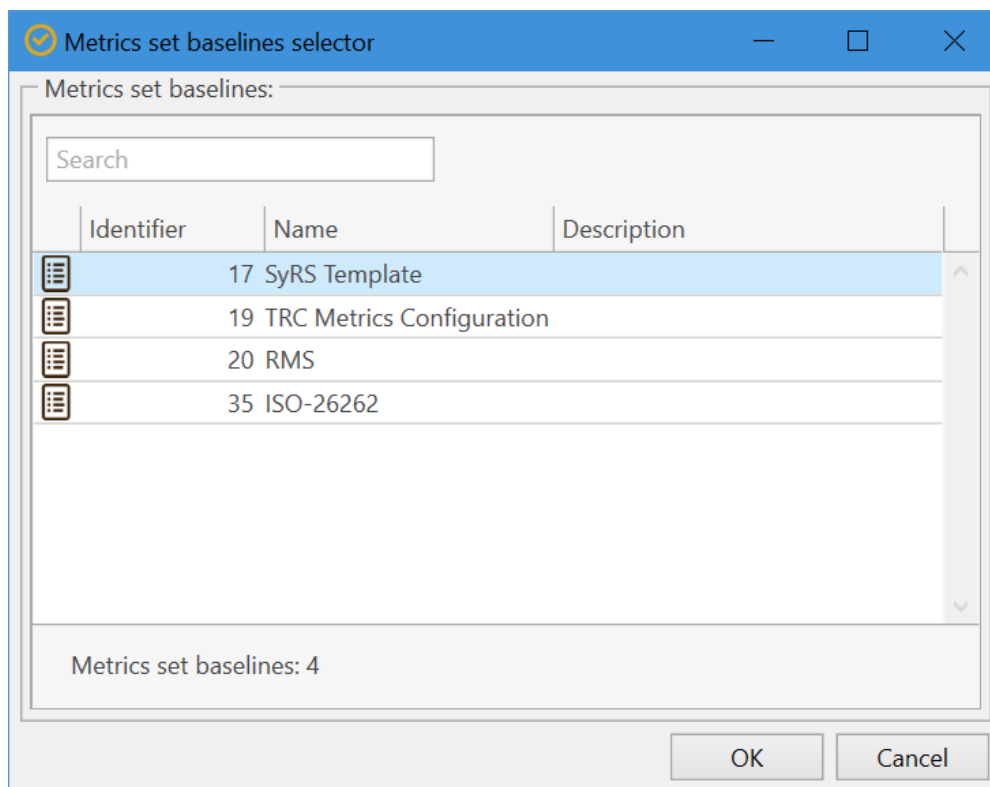


**Figure 43.** RQA Connection Window

- In order to assess the quality of the model, it is necessary to select a template that stores the set of metrics, For further information of the metrics [63] (Figure 44 and Figure 45).

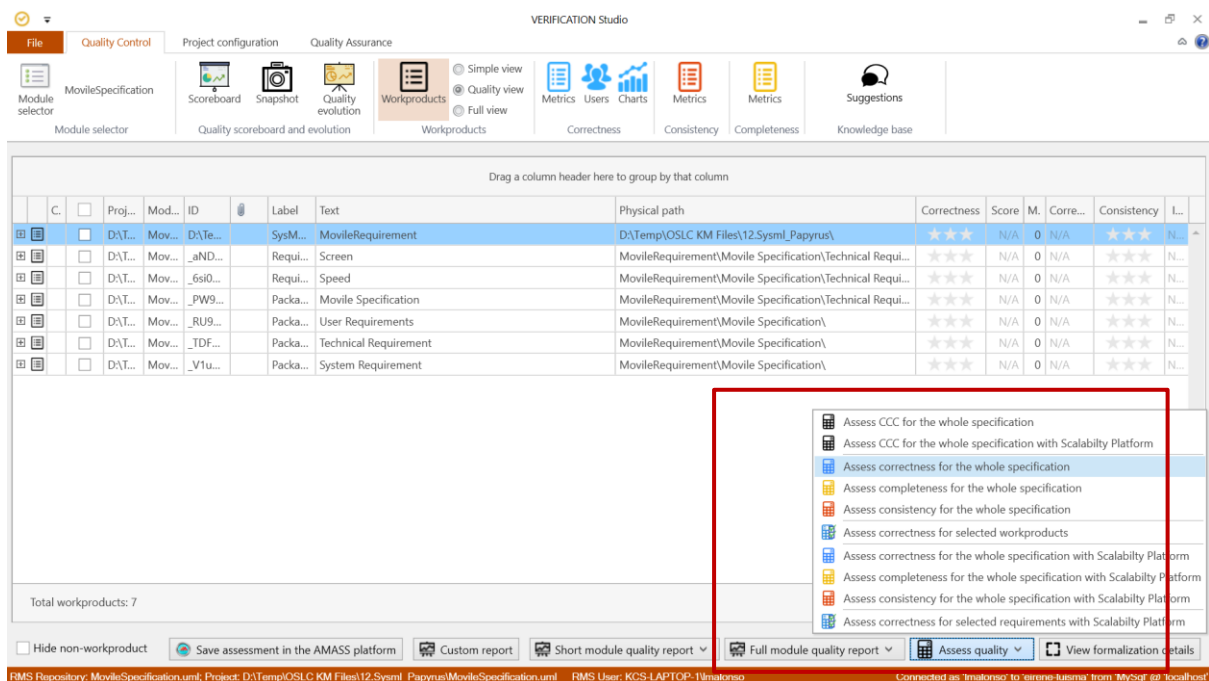


**Figure 44.** Information message to select a set of metrics



**Figure 45.** Window with the templates store in RQA that contains the set of metrics

5. When the template is selected, RQA is ready to assess the quality of the model. It is possible assess the quality for correctness, completeness and consistency metrics (see Figure 46 and Figure 47).



**Figure 46.** RQA ready to assess the quality

MobileRequirement\Mobile Specification\Technical Requi...	★★★★	N/A	0	N/A	★★★★	N...
MobileRequirement\Mobile Specification\Technical Requi...	★★★★	N/A	0	N/A	★★★★	N...
MobileRequirement\Mobile Specification\Technical Requi...	★★★★	N/A	0	N/A	★★★★	N...
MobileRequirement\Mobile Specification\	★★★★	N/A	0	N/A	★★★★	N...
MobileRequirement\Mobile Specification\	★★★★	N/A	0	N/A	★★★★	N...
MobileRequirement\Mobile Specification\	★★★★	N/A	0	N/A	★★★★	N...

- Assess CCC for the whole specification
- Assess CCC for the whole specification with Scalability Platform
- Assess correctness for the whole specification
- Assess completeness for the whole specification
- Assess consistency for the whole specification
- Assess correctness for selected workproducts
- Assess correctness for the whole specification with Scalability Platform
- Assess completeness for the whole specification with Scalability Platform
- Assess consistency for the whole specification with Scalability Platform
- Assess correctness for selected requirements with Scalability Platform

module quality report ▼
Full module quality report ▼
Assess quality ▼
View formalization details

MS User: KCS-LAPTOP-1\lmalonso
Connected as 'lmalonso' to 'eirene-luisma' from 'MySQL' @ 'localhost'

**Figure 47.** Detail of the assessment options

- Once the model has been assessed, the quality of the different artefacts of the model is shown in RQA tool (see Figure 48). This quality measure can be exported as evidence to an AMASS repository. For this it is necessary to include the location of the server and select the project (see Figure 49 and Figure 50).

VERIFICATION Studio

view view v

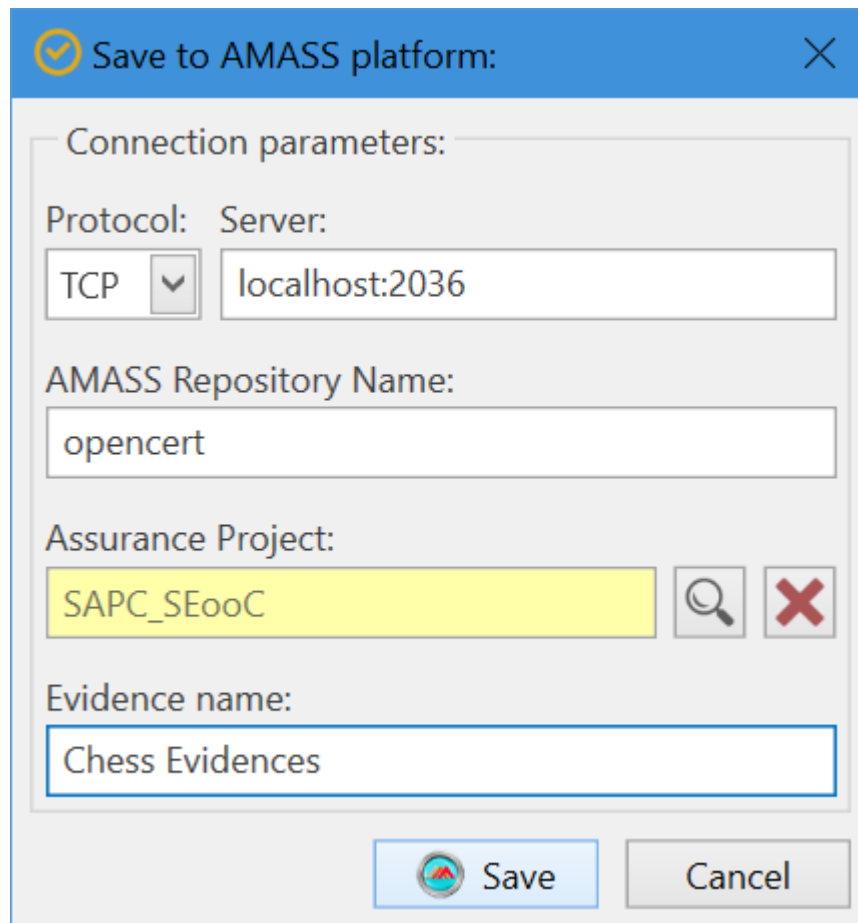
- Metrics Users Charts
- Metrics
- Metrics
- Suggestions

Correctness Consistency Completeness Knowledge base

drag a column header here to group by that column

Physical path	Correctness	Score	M.	Corre...	Consistency	I...
D:\Temp\OSLC KM Files\12.Sysml_Papyrus\	★★★★	1.00	0	19/09...	★★★★	N...
MobileRequirement\Mobile Specification\Technical Requi...	★★★★	1.00	0	19/09...	★★★★	N...
MobileRequirement\Mobile Specification\Technical Requi...	★★★★	1.00	0	19/09...	★★★★	N...
MobileRequirement\Mobile Specification\Technical Requi...	★★★★	1.00	0	19/09...	★★★★	N...
MobileRequirement\Mobile Specification\	★★★★	1.00	0	19/09...	★★★★	N...
MobileRequirement\Mobile Specification\	★★★★	1.00	0	19/09...	★★★★	N...
MobileRequirement\Mobile Specification\	★★★★	1.00	0	19/09...	★★★★	N...

**Figure 48.** Quality information of the model in RQA.



**Save to AMASS platform:**

Connection parameters:

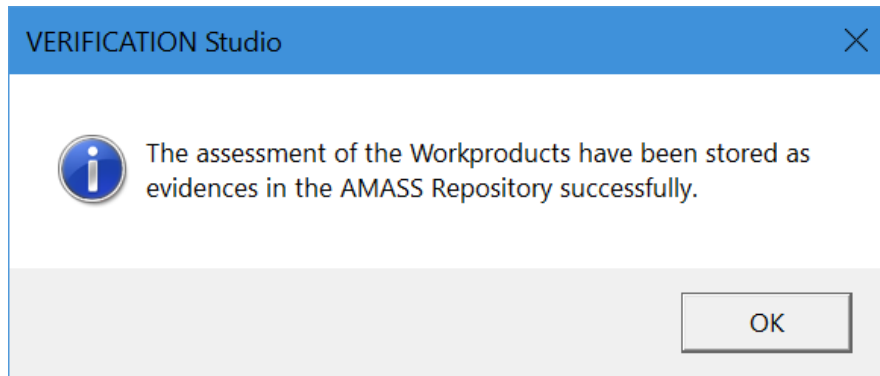
Protocol:  Server:

AMASS Repository Name:


Assurance Project:

Evidence name:

**Figure 49.** Connection window to export the evidence in an AMASS repository

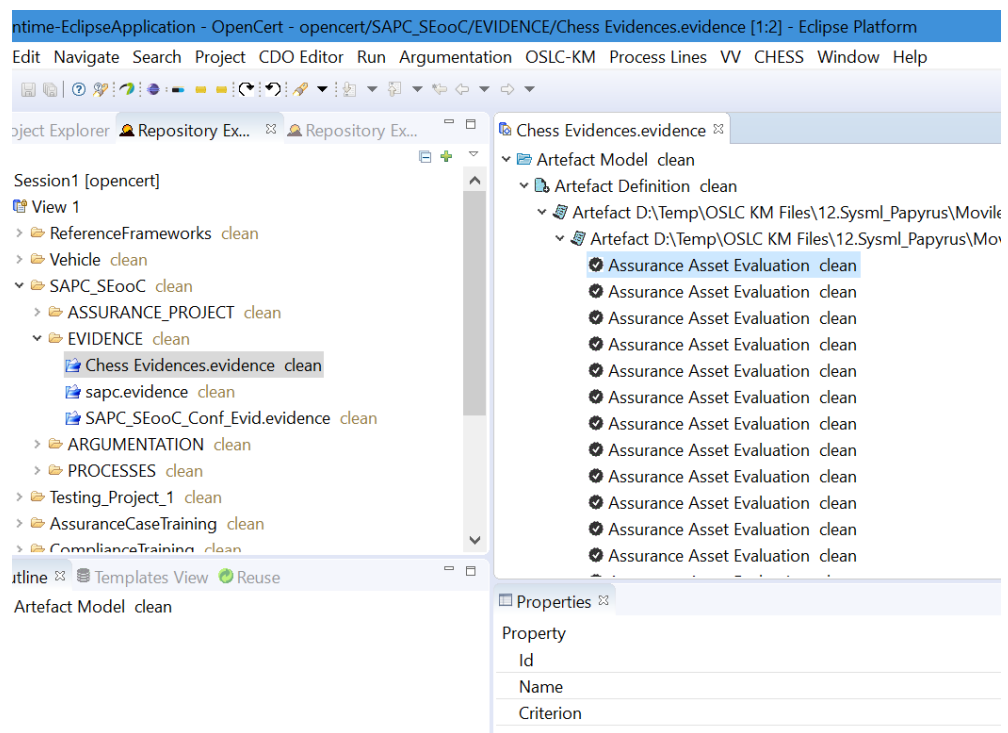


**VERIFICATION Studio**

 The assessment of the Workproducts have been stored as evidences in the AMASS Repository successfully.

**Figure 50.** Information message of the stored process

7. Following the detail with the evidence stored in the assurance project is shown (see Figure 51).



**Figure 51.** Evidence stored in the assurance project

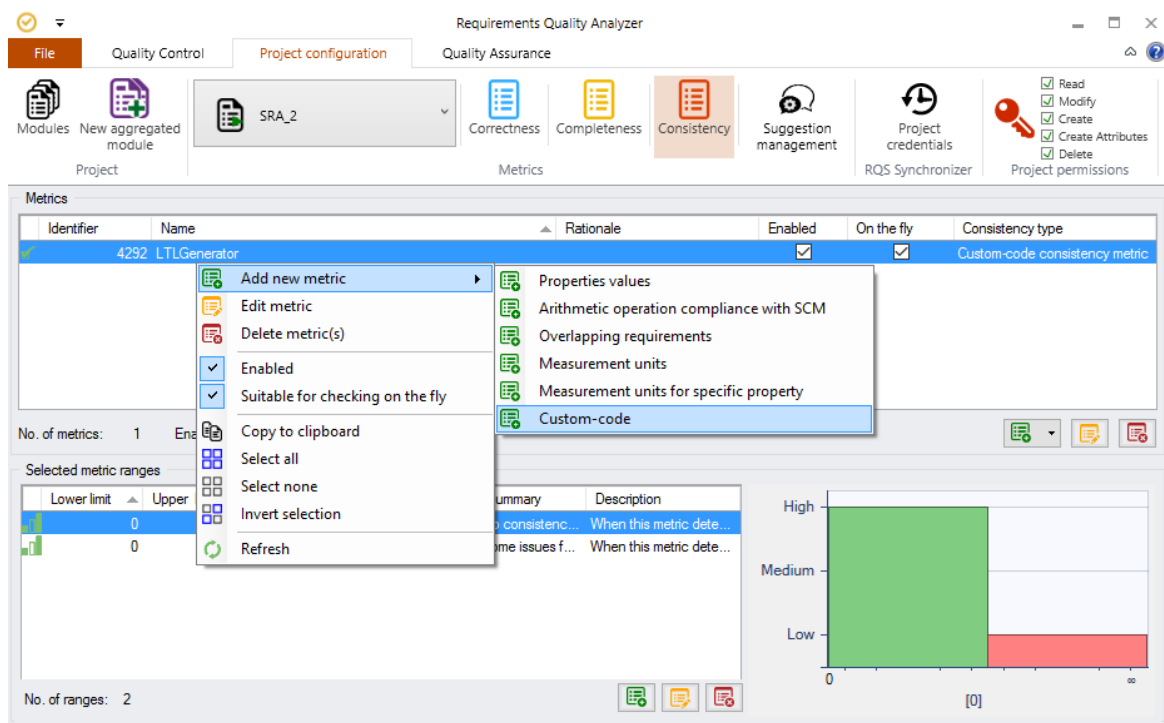
### 3.2.4.3.1 Requirements Formalization for analysis of Temporal Realizability – Requirement Quality Analyser approach

The goal of this activity is to perform a consistency quality assessment in requirements specifications, by detecting temporal elements and analysing its realizability.

As explained in D3.6 “Prototype-for-architecture-driven-assurance (c)” [29] (Section 2.2.3.1), the implementation consists of a NLP software mechanism applied to textual requirements in order to make a quality assessment, in terms of temporal consistency. To summarize, the metric 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, these patterns are a structure used to detect sentence based on the position of the words with the goal to recover relevant information. The resultant LTL is being processed by **Acacia+**, an open source tool with algorithms to check the reliability, syntehisis and optimization of LTL specifications. Thus, Acacia+ will check the temporal consistency and return if the resultant LTL is reliable (high quality) or not (low quality). The deployment and configuration of the temporal consistency metric in RQA will perform these steps:

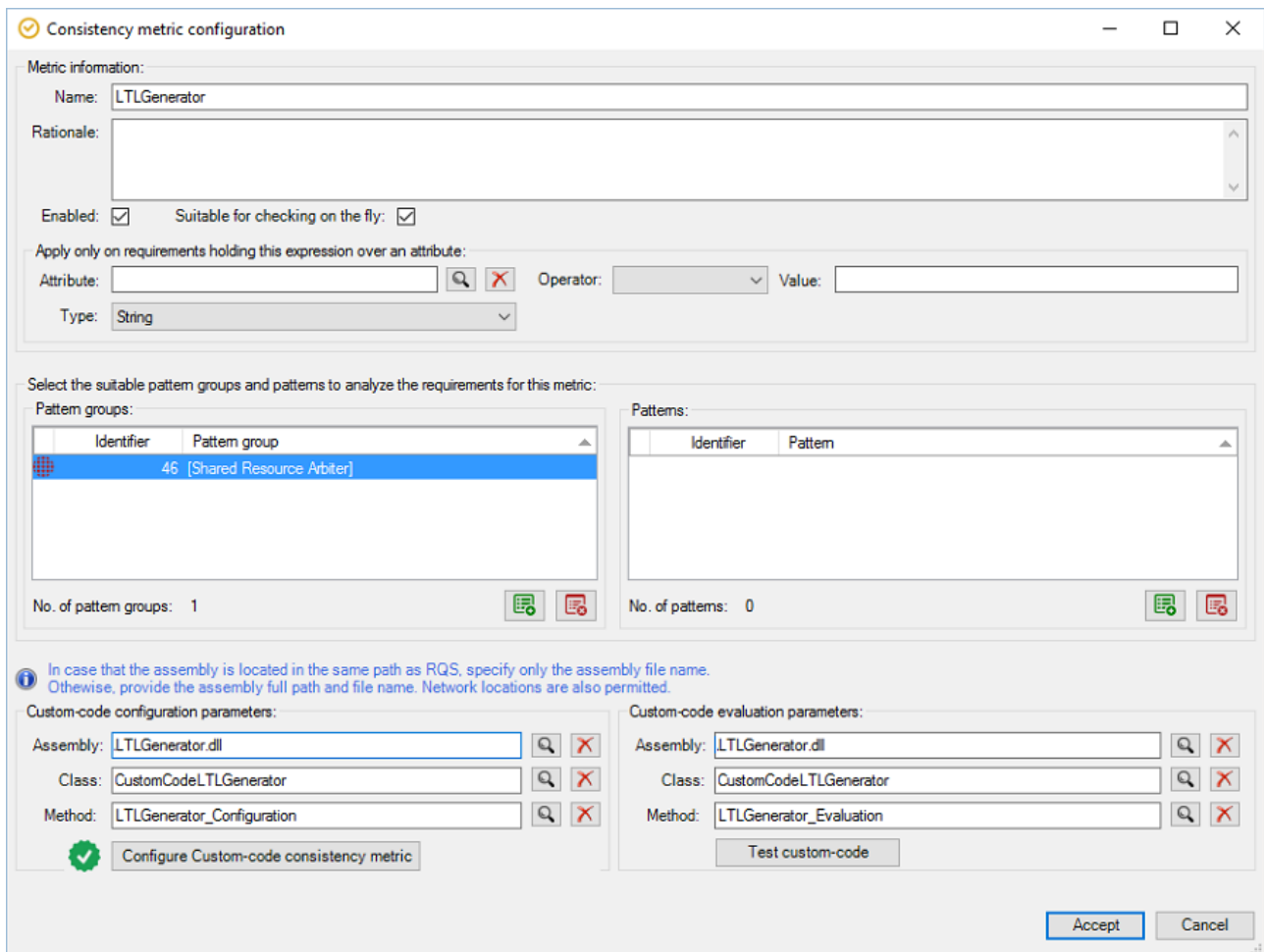
In the first place, for the deployment of the new metric, it is necessary to get into the “*Consistency*” section in Requirement Quality Analyser (RQA), and select the addition of a new Custom-coded quality metric, see Figure 52.





**Figure 52.** Creation of a new custom-coded consistency metric

The following window will pop-up, Figure 53. It requests a metric name and description (optional) of the new metric, a requirements filter (optional), the suitable patterns or patterns group (set of patterns) from the System Knowledge Base, and the implemented library information. The System Knowledge Base (SKB) is the Ontology of the system managed by the Knowledge Manager (KM) tool.



**Consistency metric configuration**

Metric information:

Name: LTLGenerator

Rationale:

Enabled: ☒ Suitable for checking on the fly: ☒

Apply only on requirements holding this expression over an attribute:

Attribute:  Operator:  Value:

Type: String

Select the suitable pattern groups and patterns to analyze the requirements for this metric:

Pattern groups:

Identifier	Pattern group
46	[Shared Resource Arbiter]

No. of pattern groups: 1

Patterns:

Identifier	Pattern
------------	---------

No. of patterns: 0

**Custom-code configuration parameters:**

Assembly: LTLGenerator.dll

Class: CustomCodeLTLGenerator

Method: LTLGenerator\_Configuration

☒ Configure Custom-code consistency metric

**Custom-code evaluation parameters:**

Assembly: LTLGenerator.dll

Class: CustomCodeLTLGenerator

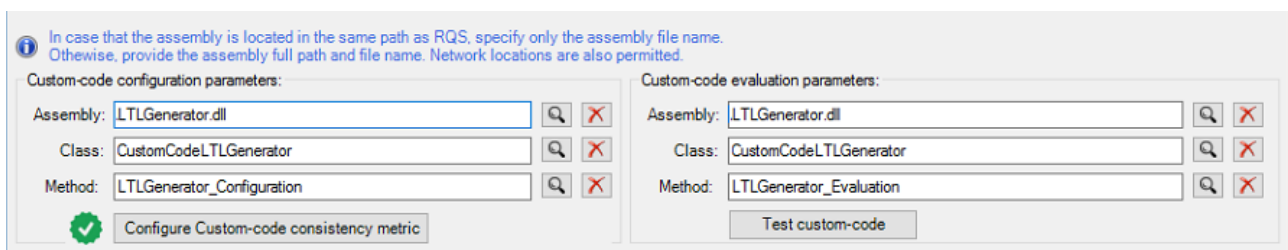
Method: LTLGenerator\_Evaluation

Test custom-code

Accept Cancel

**Figure 53.** Custom-coded configuration step

For more details, according to the RQA's available API, it requests the library (assembly name and location, class name and method name) where the Configuration and Evaluation function are implemented (Figure 54).



**Custom-code configuration parameters:**

Assembly: LTLGenerator.dll

Class: CustomCodeLTLGenerator

Method: LTLGenerator\_Configuration

☒ Configure Custom-code consistency metric

**Custom-code evaluation parameters:**

Assembly: LTLGenerator.dll

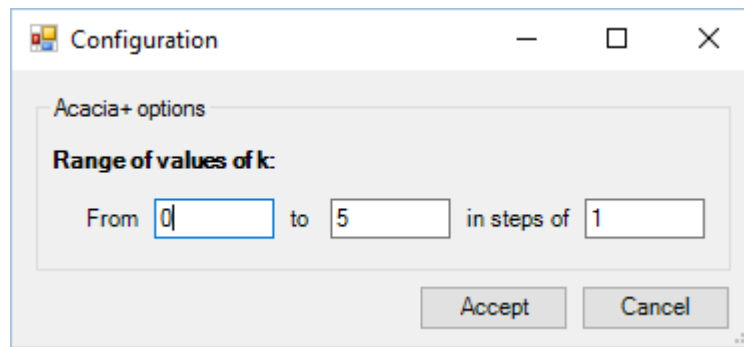
Class: CustomCodeLTLGenerator

Method: LTLGenerator\_Evaluation

Test custom-code

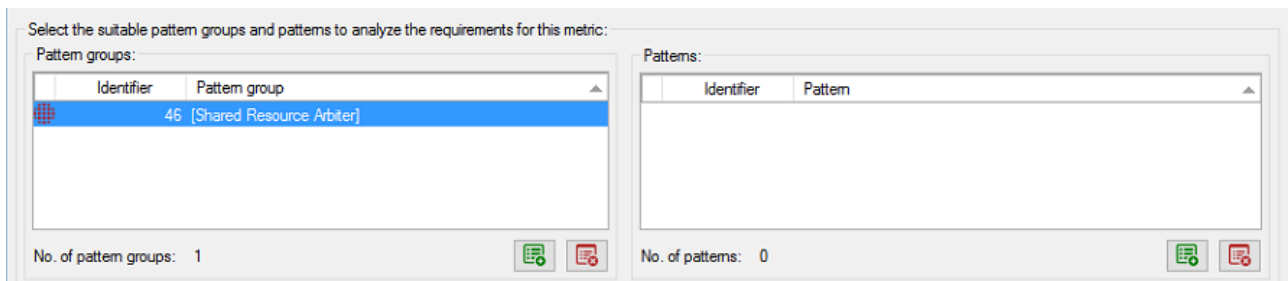
**Figure 54.** Assembly, class and method selection

Second, it is time for the configuration step, Figure 55. In one hand, it is necessary to establish the only Acacia+ configurable parameter. Precisely, it is the K number of accepted states, setting a range from 0 to 5, with an incremental coefficient of 1, which restricts the number of iterations in case the problem is hard to be computed.



**Figure 55.** Range of values of K

On the other hand, it is also necessary to setup the suitable patterns or pattern groups that will match the requirements for the LTL translation, see Figure 56.



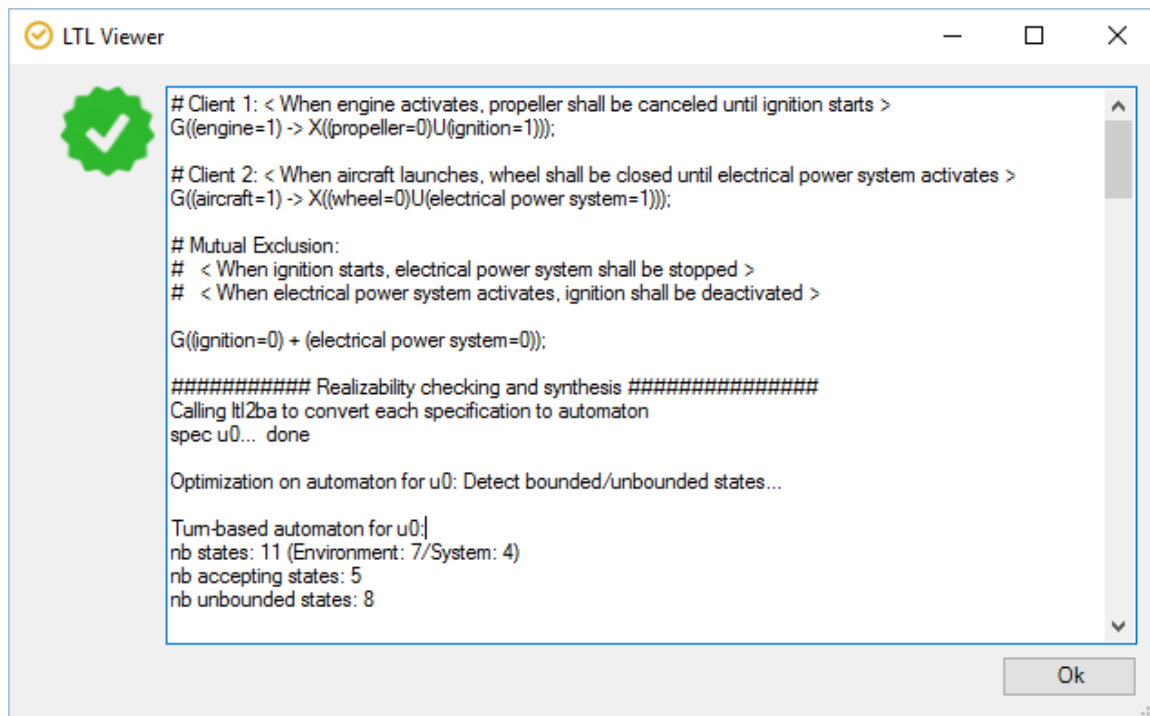
**Figure 56.** Pattern groups selection

The deployment and configuration steps are done and the metric is ready to be applied against the requirements specification, see Figure 57. Of course, the objective is to either tune the metric configuration or modify the requirements specification according to the metric results.

Metrics						
Identifier	Name	Quality	Score	Quality date	Summary	
4525	Acacia+ Temporal Verification 1	★☆☆	1.00	9/4/2016 11:55:23 AM	Neither realizability nor unrealizability has been proved.	
4526	Acacia+ Temporal Verification 2	★★★	0.00	9/4/2016 11:55:23 AM	Formula is realizable.	

**Figure 57.** Metric ready to assess quality

By double-clicking on the metric, the results will show up. Notice that there are only two possible final states: temporally reliable specification or not (with the selected configuration). Thus, for the reliable case, the result is high quality (3 stars), Figure 57, or low quality (1 star) for the case it is not possible to demonstrate the reliable. Additionally, there is a result window showing the final translation and Acacia+ result, see Figure 58.



**Figure 58.** Metric results

### 3.2.4.3.2 Requirements Quality Metrics

#### Correctness metric:

With the activity related to the correctness metrics, the requirements in the specification are analysed with the System Knowledge Base (SKB) detecting correctness issues, for further information: about SKB [64], and about correctness metric [63].

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.

Following, it is shown the different metrics implemented, see Figure 59 and Figure 60. To create one of these metrics it is only necessary to select the proper metric.



Correctness    Completeness    Consistency

Correctness metrics:

Identifier	Metric I...	Name	Rationale	Weight
✓ 17,119	1175	Out-of-SCC Nouns	Nouns that are not part of the SCC must be avoided.	
✓ 16,067	1145	Out-of-SCC Nouns	Nouns that are not part of the SCC must be avoided.	
✓ 17,047	1171	Out-of-SCC Verbs	Verbs that are not part of the SCC must be avoided.	

Context menu for 'Out-of-SCC Verbs':

- Add new metric
  - Based on RMS
  - Based on Simple Text content
  - Based on SKB
  - Based on Textual structure
  - Based on Special Sentences
  - Custom-code metric
- Edit metric
- Delete metric(s)
- Enable Selected
- Enable all
- Disable all
- Search...
- Copy to
  - Indefinite articles (Avoid)
  - Passive voice (Avoid)
  - TRC - Conditional mode (Avoid)
- Select all
- Select none
- Invert selection
- Copy selection to clipboard
- Export...
- Refresh

Quality metrics table:

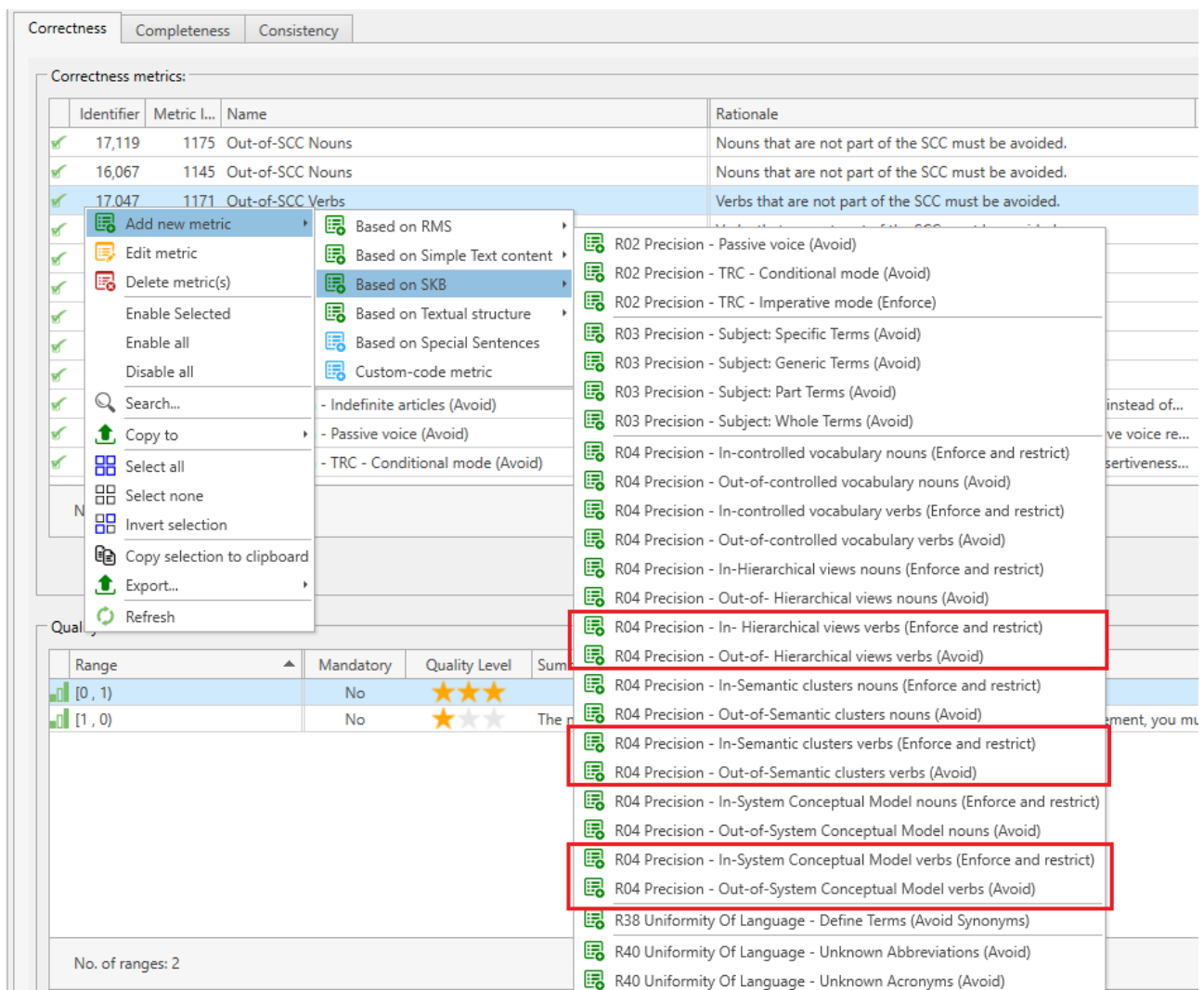
Range	Mandatory	Quality Level	Summary
[0, 1)	No	★★★	
[1, 0)	No	★☆☆	The n...

No. of ranges: 2

Additional metrics (R04 Precision - In-Hierarchical views nouns (Enforce and restrict) and R04 Precision - Out-of- Hierarchical views nouns (Avoid) are highlighted in red):

- R02 Precision - Passive voice (Avoid)
- R02 Precision - TRC - Conditional mode (Avoid)
- R02 Precision - TRC - Imperative mode (Enforce)
- R03 Precision - Subject: Specific Terms (Avoid)
- R03 Precision - Subject: Generic Terms (Avoid)
- R03 Precision - Subject: Part Terms (Avoid)
- R03 Precision - Subject: Whole Terms (Avoid)
- R04 Precision - In-controlled vocabulary nouns (Enforce and restrict)
- R04 Precision - Out-of-controlled vocabulary nouns (Avoid)
- R04 Precision - In-controlled vocabulary verbs (Enforce and restrict)
- R04 Precision - Out-of-controlled vocabulary verbs (Avoid)
- R04 Precision - In-Hierarchical views nouns (Enforce and restrict)
- R04 Precision - Out-of- Hierarchical views nouns (Avoid)
- R04 Precision - In- Hierarchical views verbs (Enforce and restrict)
- R04 Precision - Out-of- Hierarchical views verbs (Avoid)
- R04 Precision - In-Semantic clusters nouns (Enforce and restrict)
- R04 Precision - Out-of-Semantic clusters nouns (Avoid)
- R04 Precision - In-Semantic clusters verbs (Enforce and restrict)
- R04 Precision - Out-of-Semantic clusters verbs (Avoid)
- R04 Precision - In-System Conceptual Model nouns (Enforce and restrict)
- R04 Precision - Out-of-System Conceptual Model nouns (Avoid)
- R04 Precision - In-System Conceptual Model verbs (Enforce and restrict)
- R04 Precision - Out-of-System Conceptual Model verbs (Avoid)
- R38 Uniformity Of Language - Define Terms (Avoid Synonyms)
- R40 Uniformity Of Language - Unknown Abbreviations (Avoid)
- R40 Uniformity Of Language - Unknown Acronyms (Avoid)

Figure 59. Correctness metrics related to nouns



Correctness metrics:

Identifier	Metric I...	Name	Rationale
17,119	1175	Out-of-SCC Nouns	Nouns that are not part of the SCC must be avoided.
16,067	1145	Out-of-SCC Nouns	Nouns that are not part of the SCC must be avoided.
17,047	1171	Out-of-SCC Verbs	Verbs that are not part of the SCC must be avoided.

Context Menu:

- Add new metric
  - Based on RMS
  - Based on Simple Text content
  - Based on SKB
  - Based on Textual structure
  - Based on Special Sentences
  - Custom-code metric
- Edit metric
- Delete metric(s)
- Enable Selected
- Enable all
- Disable all
- Search...
- Copy to
- Select all
- Select none
- Invert selection
- Copy selection to clipboard
- Export...
- Refresh

Quality metrics table:

Range	Mandatory	Quality Level	Sum
[0, 1]	No	★★★	
[1, 0]	No	★☆☆	The p

No. of ranges: 2

Metrics list (highlighted items in red boxes):

- R02 Precision - Passive voice (Avoid)
- R02 Precision - TRC - Conditional mode (Avoid)
- R02 Precision - TRC - Imperative mode (Enforce)
- R03 Precision - Subject: Specific Terms (Avoid)
- R03 Precision - Subject: Generic Terms (Avoid)
- R03 Precision - Subject: Part Terms (Avoid)
- R03 Precision - Subject: Whole Terms (Avoid)
- R04 Precision - In-controlled vocabulary nouns (Enforce and restrict)
- R04 Precision - Out-of-controlled vocabulary nouns (Avoid)
- R04 Precision - In-controlled vocabulary verbs (Enforce and restrict)
- R04 Precision - Out-of-controlled vocabulary verbs (Avoid)
- R04 Precision - In-Hierarchical views nouns (Enforce and restrict)
- R04 Precision - Out-of- Hierarchical views nouns (Avoid)
- R04 Precision - In- Hierarchical views verbs (Enforce and restrict)
- R04 Precision - Out-of- Hierarchical views verbs (Avoid)
- R04 Precision - In-Semantic clusters nouns (Enforce and restrict)
- R04 Precision - Out-of-Semantic clusters nouns (Avoid)
- R04 Precision - In-Semantic clusters verbs (Enforce and restrict)
- R04 Precision - Out-of-Semantic clusters verbs (Avoid)
- R04 Precision - In-System Conceptual Model nouns (Enforce and restrict)
- R04 Precision - Out-of-System Conceptual Model nouns (Avoid)
- R04 Precision - In-System Conceptual Model verbs (Enforce and restrict)
- R04 Precision - Out-of-System Conceptual Model verbs (Avoid)
- R38 Uniformity Of Language - Define Terms (Avoid Synonyms)
- R40 Uniformity Of Language - Unknown Abbreviations (Avoid)
- R40 Uniformity Of Language - Unknown Acronyms (Avoid)

**Figure 60.** Correctness metrics related to verbs

Following, an explanation of the correctness metrics is shown:

### In-System Conceptual Model Nouns

The RQA tool allows to create this metric to check if each term in the requirement belongs to the SCM view or any semantic. The SCM describes a (usually) partial model of the system universe, exclusively at conceptual level. Everything regarding the System universe can/should/must be represented in the SCM and stored in the SKB.

### Out-of-System Conceptual Model Nouns

This metric checks if each term does not belong to any SCM view or any semantic cluster.

### In-Semantic Clusters Nouns

The metric checks if each term of the requirement belongs to one or more semantic clusters.

### Out-of-Semantic Clusters Nouns

The metric checks if each term does not belong to any semantic cluster.

### In-Hierarchical Views Nouns

It is allowed to create this metric to check that each term in the requirement belongs to one or more SCM view.

### Out-of-Hierarchical Views Nouns

The RQA tool allows to create this metric to check that each term in the requirement does not belong to one or more SCM view.

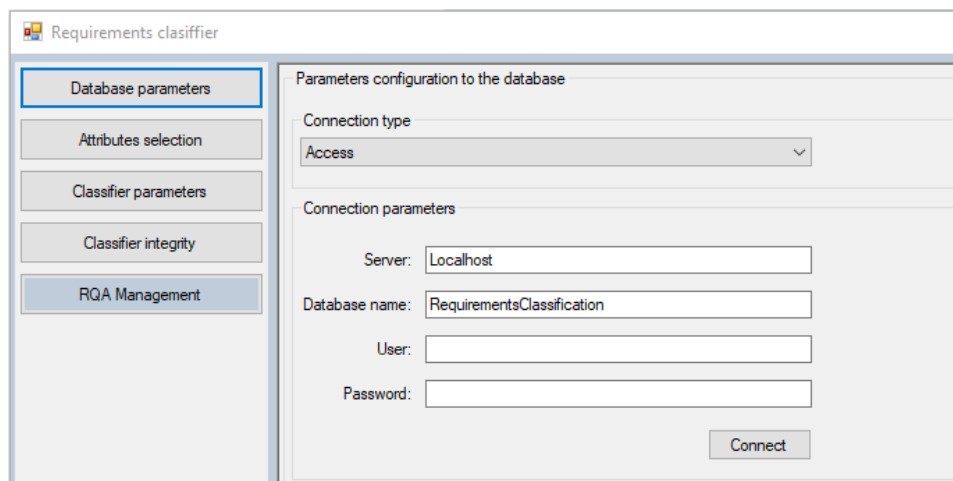
The metrics related to the verbs are similar that those about nouns but focused on verbs.

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

The objective of this activity is to generate a classifier to analyse the correctness metrics of a requirement and to predict their quality in function of a set of requirements previously classified in function of their quality. For further information, see D3.3 “Design of the AMASS tools and methods for architecture-driven assurance (b)” [28]. The process uses machine learning techniques to analyse the correctness metric of the set of requirements and to generate the classifier. For further information, use the reference [62].

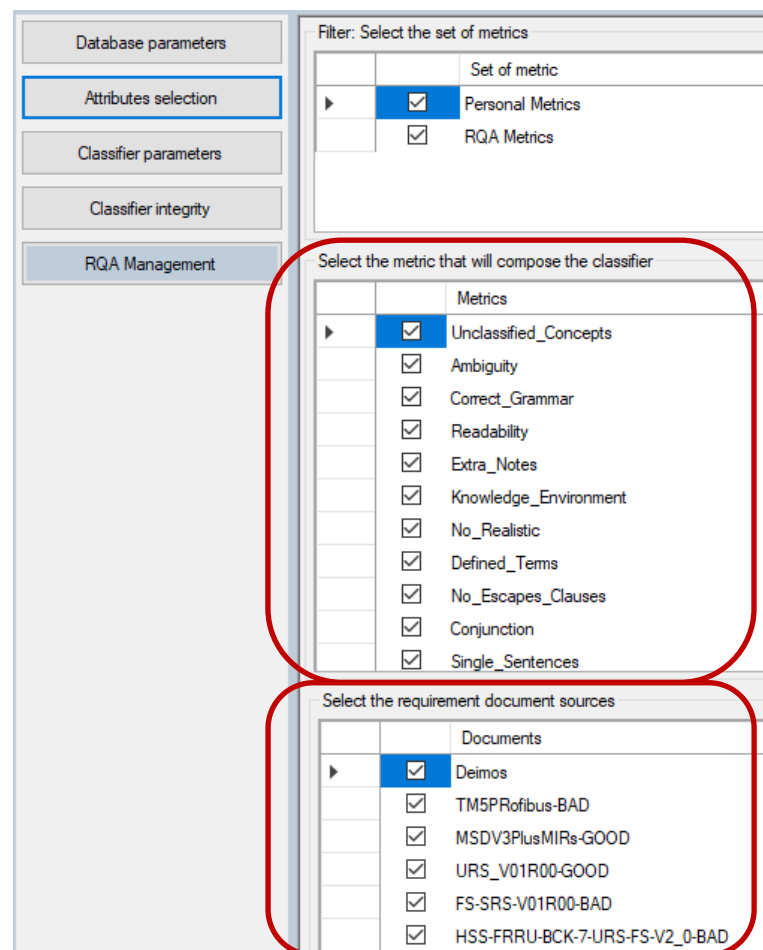
Following the steps to create the classifier are shown:

1. Connect to database, Figure 61.



**Figure 61.** Database connection

2. Select the specifications and the correctness metrics that the Requirements Quality Analyser (RQA) tool recovers for each requirement, Figure 62.



Database parameters

Attributes selection

Classifier parameters

Classifier integrity

RQA Management

Filter: Select the set of metrics

	Set of metric
<input checked="" type="checkbox"/>	Personal Metrics
<input checked="" type="checkbox"/>	RQA Metrics

Select the metric that will compose the classifier

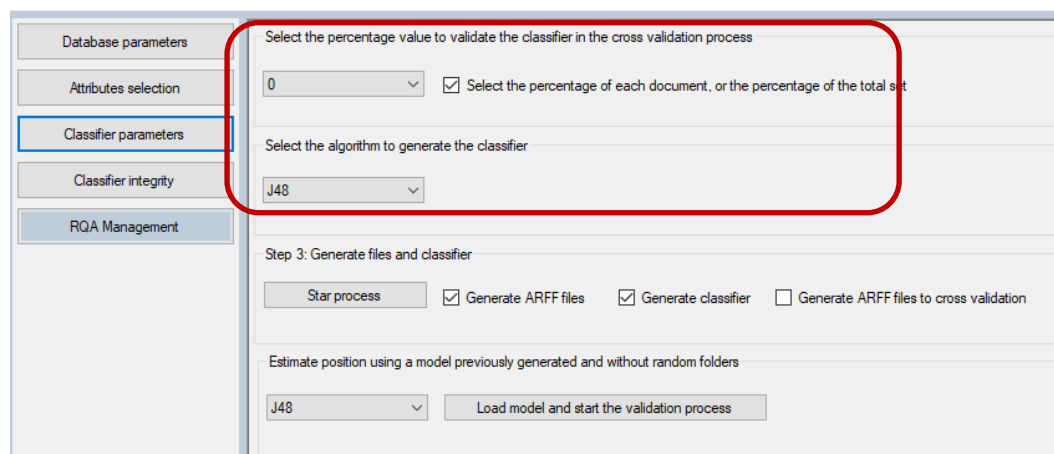
	Metrics
<input checked="" type="checkbox"/>	Unclassified_Concepts
<input checked="" type="checkbox"/>	Ambiguity
<input checked="" type="checkbox"/>	Correct_Grammar
<input checked="" type="checkbox"/>	Readability
<input checked="" type="checkbox"/>	Extra_Notes
<input checked="" type="checkbox"/>	Knowledge_Environment
<input checked="" type="checkbox"/>	No_Realistic
<input checked="" type="checkbox"/>	Defined_Terms
<input checked="" type="checkbox"/>	No_Escapes_Clauses
<input checked="" type="checkbox"/>	Conjunction
<input checked="" type="checkbox"/>	Single_Sentences

Select the requirement document sources

	Documents
<input checked="" type="checkbox"/>	Deimos
<input checked="" type="checkbox"/>	TM5PProfibus-BAD
<input checked="" type="checkbox"/>	MSDV3PlusMIRs-GOOD
<input checked="" type="checkbox"/>	URS_V01R00-GOOD
<input checked="" type="checkbox"/>	FS-SRS-V01R00-BAD
<input checked="" type="checkbox"/>	HSS-FRRU-BCK-7-URS-FS-V2_0-BAD

**Figure 62.** Correctness metrics selection

3. Configure parameters of machine learning algorithms, Figure 63.



Database parameters

Attributes selection

Classifier parameters

Classifier integrity

RQA Management

Select the percentage value to validate the classifier in the cross validation process

0 ☐ Select the percentage of each document, or the percentage of the total set

Select the algorithm to generate the classifier

J48

Step 3: Generate files and classifier

Star process ☒ Generate ARFF files ☒ Generate classifier ☐ Generate ARFF files to cross validation

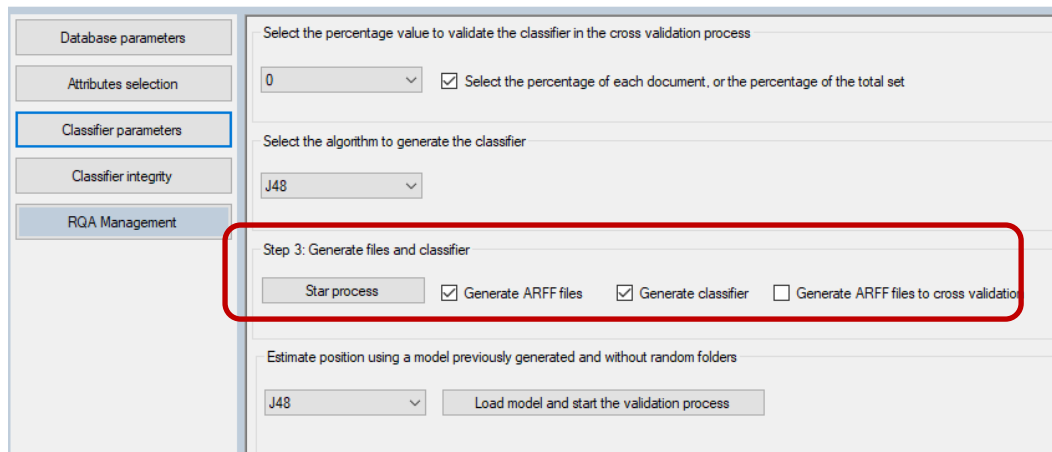
Estimate position using a model previously generated and without random folders

J48

**Figure 63.** Parameters configuration

4. Generate classifier, Figure 64.





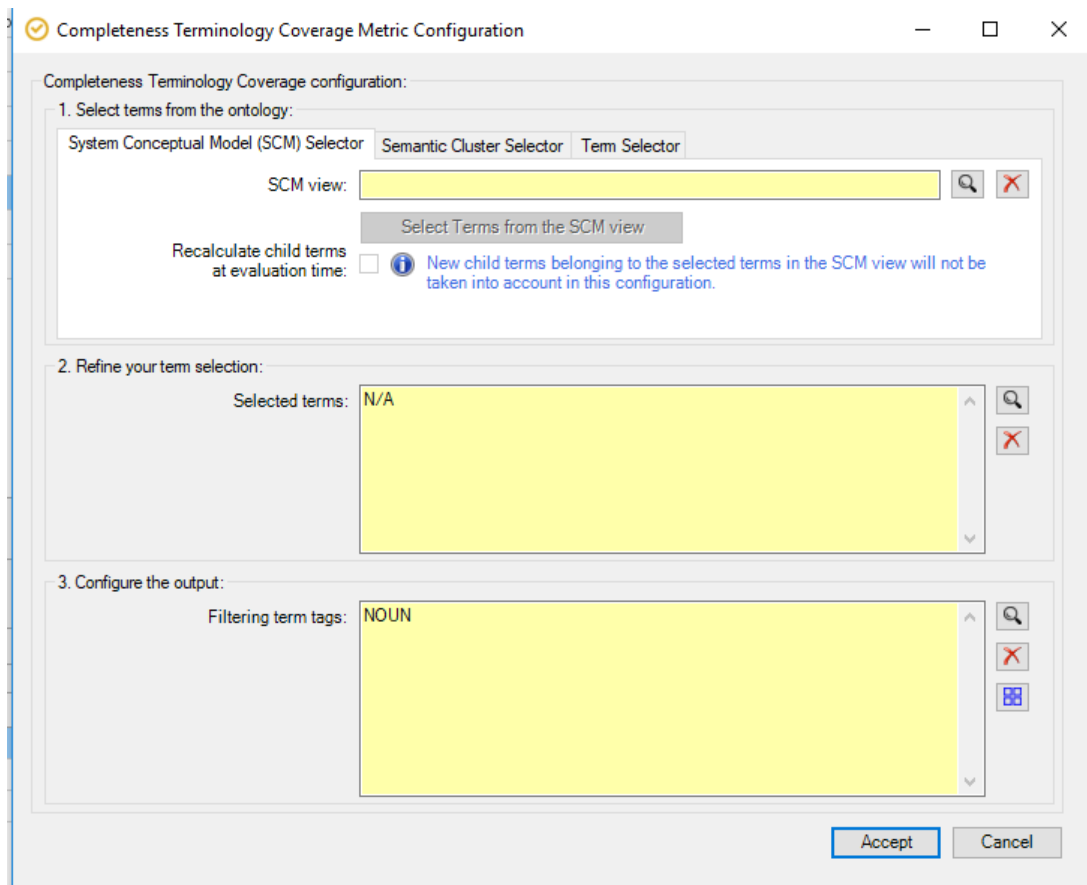
**Figure 64.** Generate classifier

#### 3.2.4.3.4 Metrics for models

The objective of the activity is to create metrics for models to evaluate completeness and consistency. To extend the knowledge of this kind of metrics use the document D3.3 “Design of the AMASS tools and methods for architecture-driven assurance (b)” [28]. A configuration process is necessary in order to define the elements that will be analysed along with the information of the models. Once the configuration is defined it is possible to create the metric and to assess the quality.

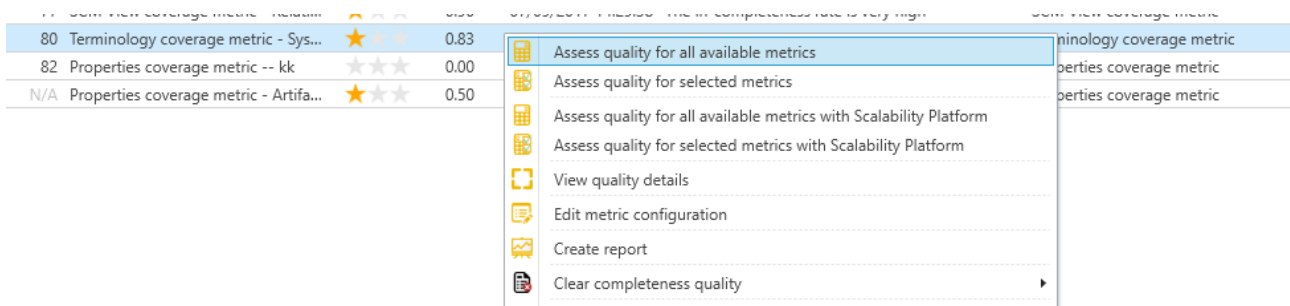
Following, as an example, the steps for creating and evaluating a terminology coverage metric are shown:

1. Select the systems and subsystems from the (SKB), Figure 65.



**Figure 65.** Selecting systems and subsystems from SCM

- The metric is executed to assess the quality of the model, Figure 66. The information of the models is extracted in the evaluation time.

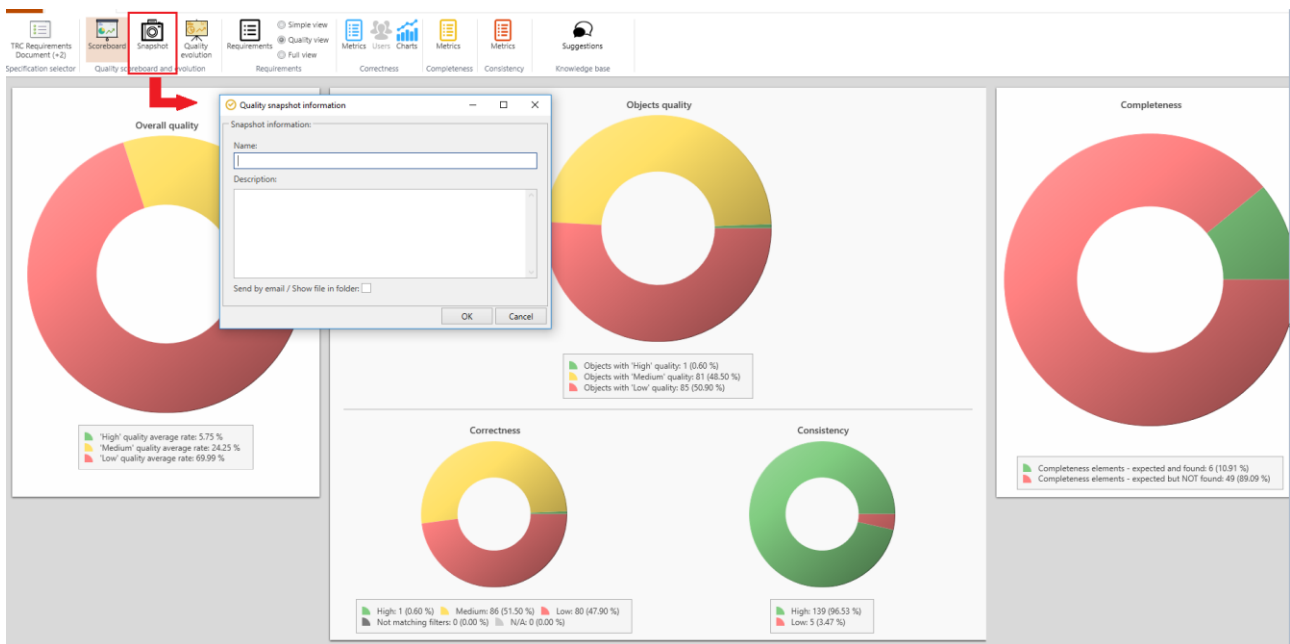


**Figure 66.** Assessing the quality metrics

### 3.2.4.3.5 Quality evolution

This activity has the goal to save a snapshot with the quality of the project and to represent, in a graphical way, the quality of the project over the time. Following the steps to save and to recover the information are shown:

- Save a snapshot with the information of the quality of the project, Figure 67.



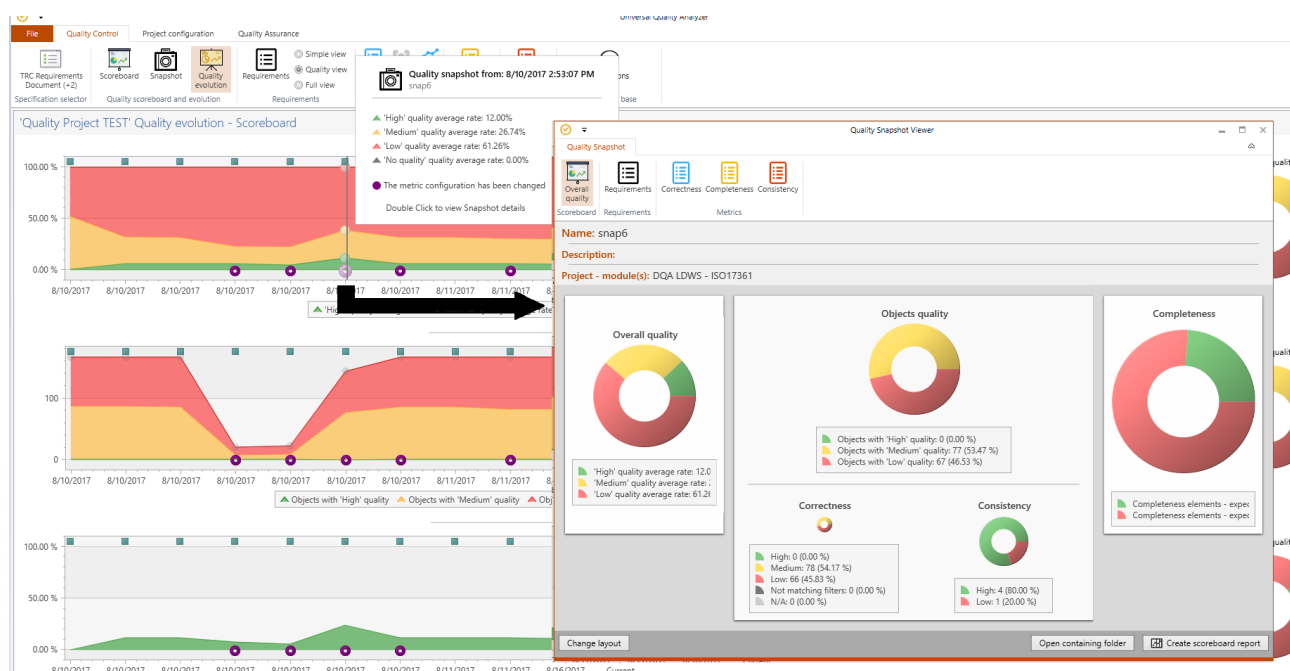
**Figure 67.** Saving snapshot with the quality of the project

The qualities of the different snapshots saved are graphically represented, Figure 68.



**Figure 68.** Graphical representation of the quality evolution

2. Select the snapshot to show the quality information of the project saved at some point, Figure 69. The snapshot contains all the information of the quality of the metrics and requirements that were included when the snapshot was saved.



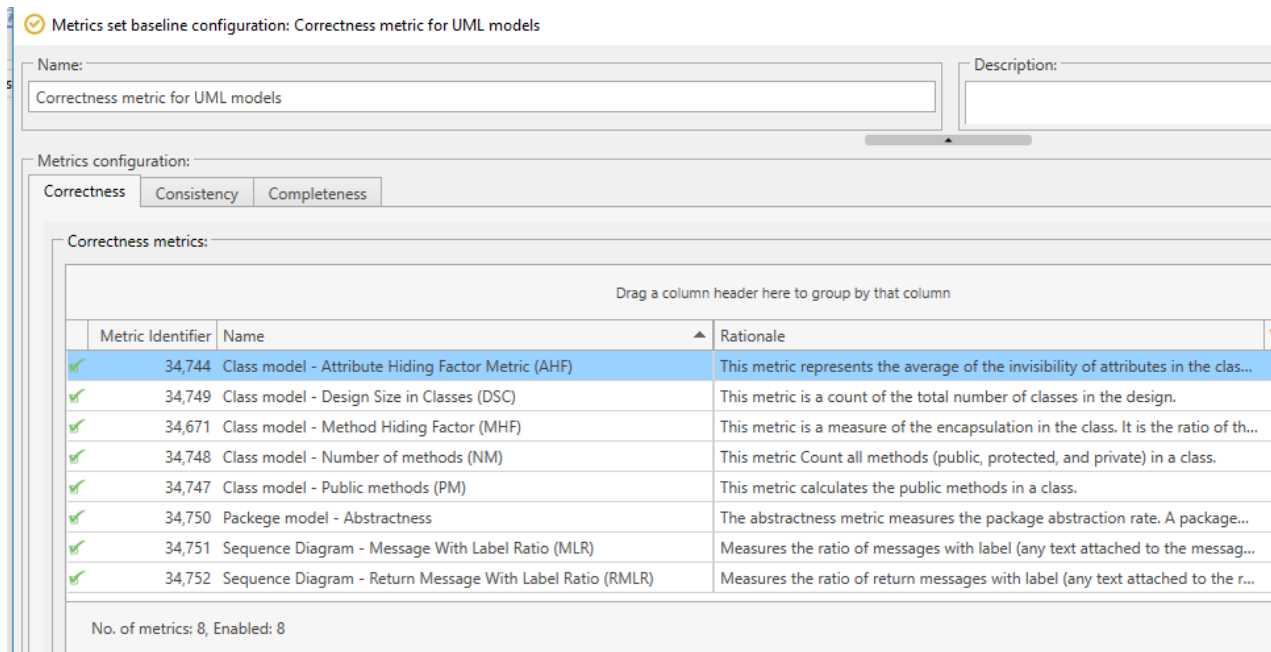
**Figure 69.** Information of the snapshot

### 3.2.4.3.6 Correctness metric for models

The objective of this activity is to create correctness metrics in order to assess the quality of models. The tool RQA includes 8 correctness metrics for 3 kinds of models: class models, package models and sequence models. For further information see the deliverable D3.3 “Design of the AMASS tools and methods for architecture driven assurance (b)” section “2.4.4.3 Metrics for models”.

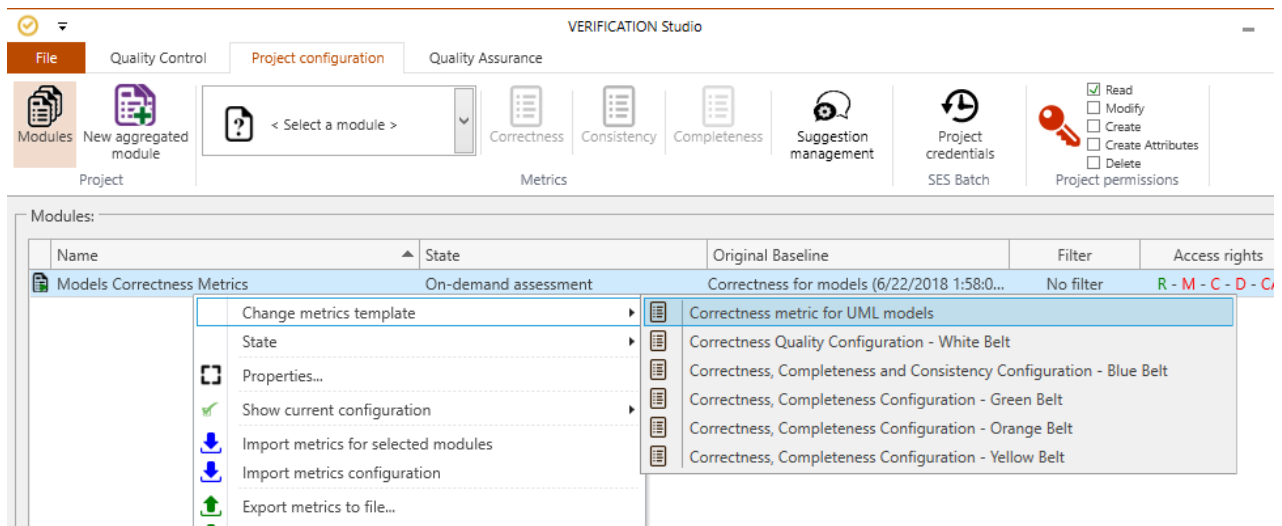
Following the steps for assigning and evaluating one of these metrics are shown:

These metrics are grouped in a template in RQA, Figure 70.



**Figure 70.** Correctness metrics for model template

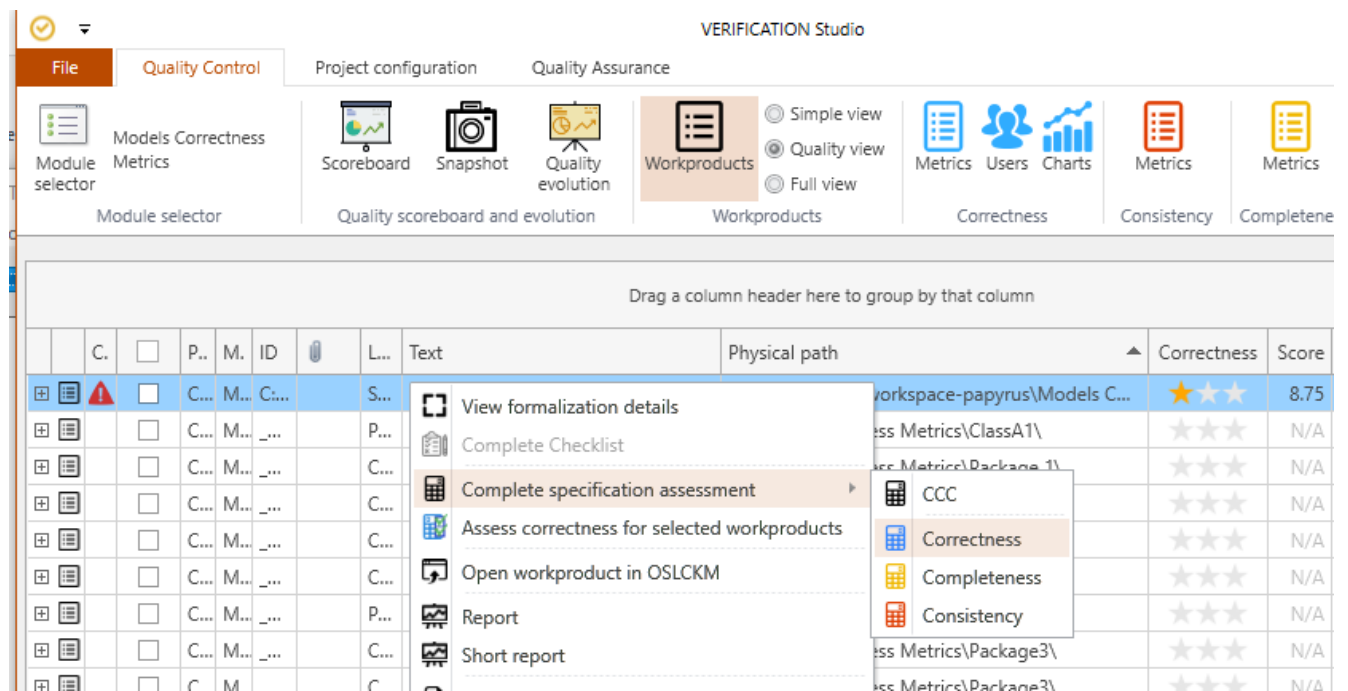
1. This template is assigned to the model to be assessed, Figure 71.



**Figure 71.** Assign template to the model

2. Assess the quality of the model:

Once the metrics are assigned, it is possible to assess the quality of the model based on these correctness metrics, Figure 72.



**Figure 72.** Assess quality of the model.

### 3.2.4.3.7 Checklist metrics

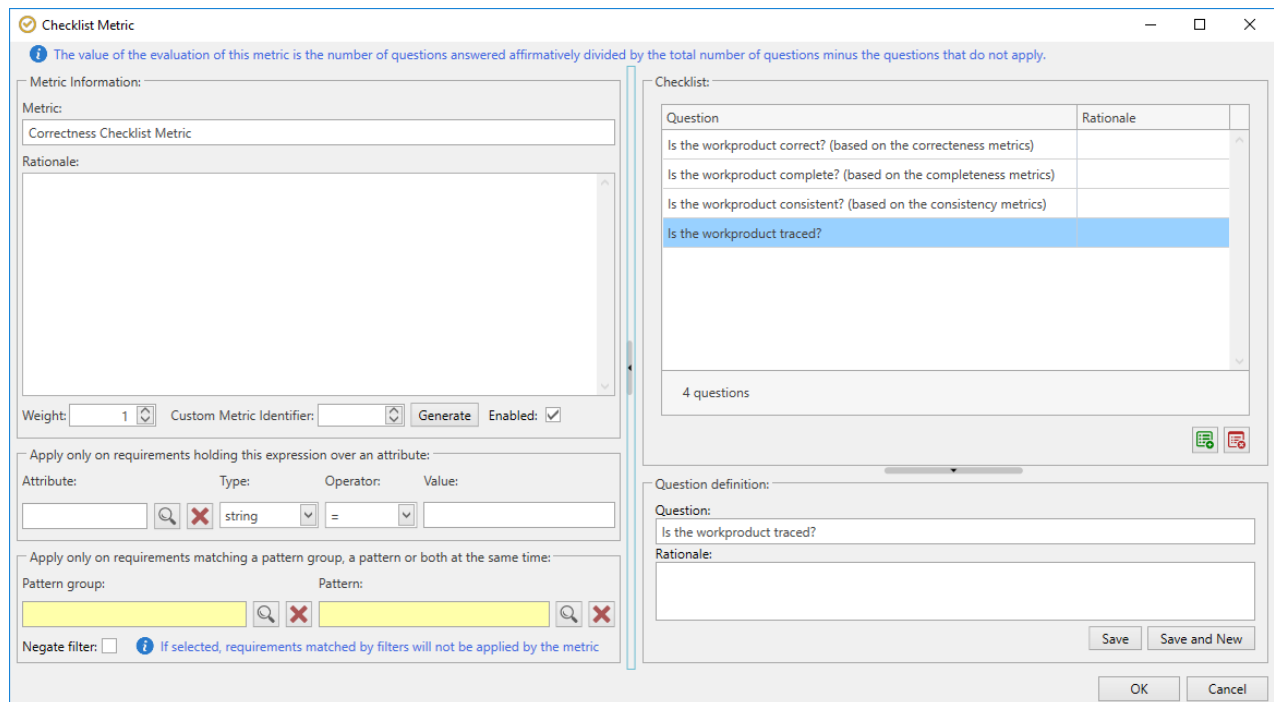
The goal of this activity is to create a new kind of metrics based on checklists. There are two types of checklist metric:

- Correctness checklist metric: the question included in this metric must be answered by each work product of the specification.
- Completeness checklist metric: the questions included in this metric must be answered at the specification level.

For further information see the deliverable D3.3 “Design of the AMASS tools and methods for architecture driven assurance”.

Following, as an example, the steps for creating and evaluating a correctness checklist metric is shown:

1. Create a checklist metric and define the questions, Figure 73:



**Checklist Metric**

The value of the evaluation of this metric is the number of questions answered affirmatively divided by the total number of questions minus the questions that do not apply.

**Metric Information:**

Metric: Correctness Checklist Metric

Rationale:

Weight: 1 Custom Metric Identifier: Generate Enabled: ☒

Apply only on requirements holding this expression over an attribute:

Attribute: Type: Operator: Value:

Apply only on requirements matching a pattern group, a pattern or both at the same time:

Pattern group: Pattern:

Negate filter: ☐ If selected, requirements matched by filters will not be applied by the metric

**Checklist:**

Question	Rationale
Is the workproduct correct? (based on the correctness metrics)	
Is the workproduct complete? (based on the completeness metrics)	
Is the workproduct consistent? (based on the consistency metrics)	
Is the workproduct traced?	

4 questions

**Question definition:**

Question: Is the workproduct traced?

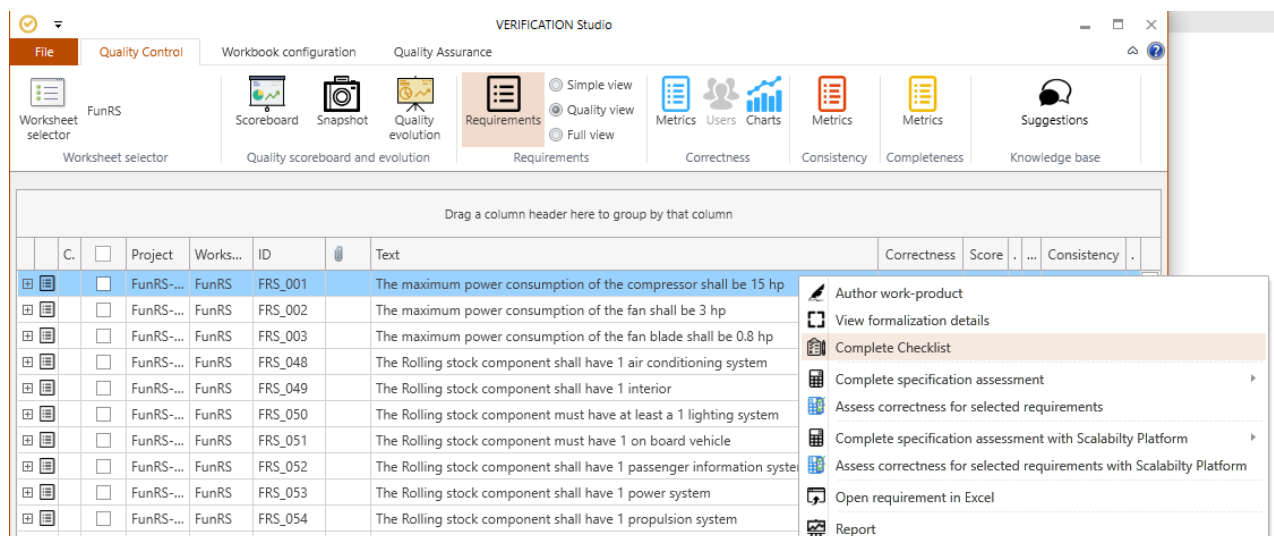
Rationale:

Save Save and New

OK Cancel

**Figure 73.** Creation the checklist metric

- Once a checklist metric is created, it is possible to complete the checklist for each requirement, Figure 74, Figure 75:



**VERIFICATION Studio**

File Quality Control Workbook configuration Quality Assurance

Worksheet selector FunRS Scoreboard Snapshot Quality evolution Requirements Simple view Quality view Full view Metrics Users Charts Metrics Metrics Suggestions

Worksheet selector Quality scoreboard and evolution Requirements Correctness Consistency Completeness Knowledge base


Drag a column header here to group by that column

	C.	Project	Works...	ID	Text	Correctness	Score	...	Consistency
<input checked="" type="checkbox"/>		FunRS-...	FunRS	FRS_001	The maximum power consumption of the compressor shall be 15 hp				
<input checked="" type="checkbox"/>		FunRS-...	FunRS	FRS_002	The maximum power consumption of the fan shall be 3 hp				
<input checked="" type="checkbox"/>		FunRS-...	FunRS	FRS_003	The maximum power consumption of the fan blade shall be 0.8 hp				
<input checked="" type="checkbox"/>		FunRS-...	FunRS	FRS_048	The Rolling stock component shall have 1 air conditioning system				
<input checked="" type="checkbox"/>		FunRS-...	FunRS	FRS_049	The Rolling stock component shall have 1 interior				
<input checked="" type="checkbox"/>		FunRS-...	FunRS	FRS_050	The Rolling stock component must have at least a 1 lighting system				
<input checked="" type="checkbox"/>		FunRS-...	FunRS	FRS_051	The Rolling stock component must have 1 on board vehicle				
<input checked="" type="checkbox"/>		FunRS-...	FunRS	FRS_052	The Rolling stock component shall have 1 passenger information system				
<input checked="" type="checkbox"/>		FunRS-...	FunRS	FRS_053	The Rolling stock component shall have 1 power system				
<input checked="" type="checkbox"/>		FunRS-...	FunRS	FRS_054	The Rolling stock component shall have 1 propulsion system				

Context Menu:

- Author work-product
- View formalization details
- Complete Checklist
- Complete specification assessment
- Assess correctness for selected requirements
- Complete specification assessment with Scalability Platform
- Assess correctness for selected requirements with Scalability Platform
- Open requirement in Excel
- Report

**Figure 74.** Complete checklist metric

 Checklist

Requirement description:

The maximum power consumption of the compressor shall be 15 hp

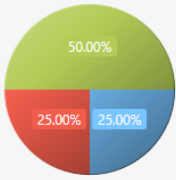
Navigation:

←

→

Edit

Answer distribution:







Yes: 2 (50.00%)

No: 1 (25.00%)

Not Applicable: 1 (25.00%)

Empty: 0 (0.00%)

Checklist:

Modified	Question	Answer	Comment	User	Date
	Is the workproduct correct? (based on the correctness metrics)	<input checked="" type="radio"/> Yes <input type="radio"/> No <input type="radio"/> Not Applicable <input type="radio"/> Empty		KCS\eparra	30/08/2018 12:...
	Is the workproduct complete? (based on the completeness metrics)	<input checked="" type="radio"/> Yes <input type="radio"/> No <input type="radio"/> Not Applicable <input type="radio"/> Empty		KCS\eparra	30/08/2018 12:...
	Is the workproduct consistent? (based on the consistency metrics)	<input type="radio"/> Yes <input checked="" type="radio"/> No <input type="radio"/> Not Applicable <input type="radio"/> Empty		KCS\eparra	30/08/2018 12:...
	Is the workproduct traced?	<input type="radio"/> Yes <input type="radio"/> No <input checked="" type="radio"/> Not Applicable <input type="radio"/> Empty		KCS\eparra	30/08/2018 12:...

4 questions

Save and Continue

Save and Exit

Cancel

**Figure 75.** Window to answer the questions

- Once the checklists metrics have been completed for each requirement, the quality of the specification must be assessed giving the possibility to analyse the quality report, Figure 76, Figure 77.

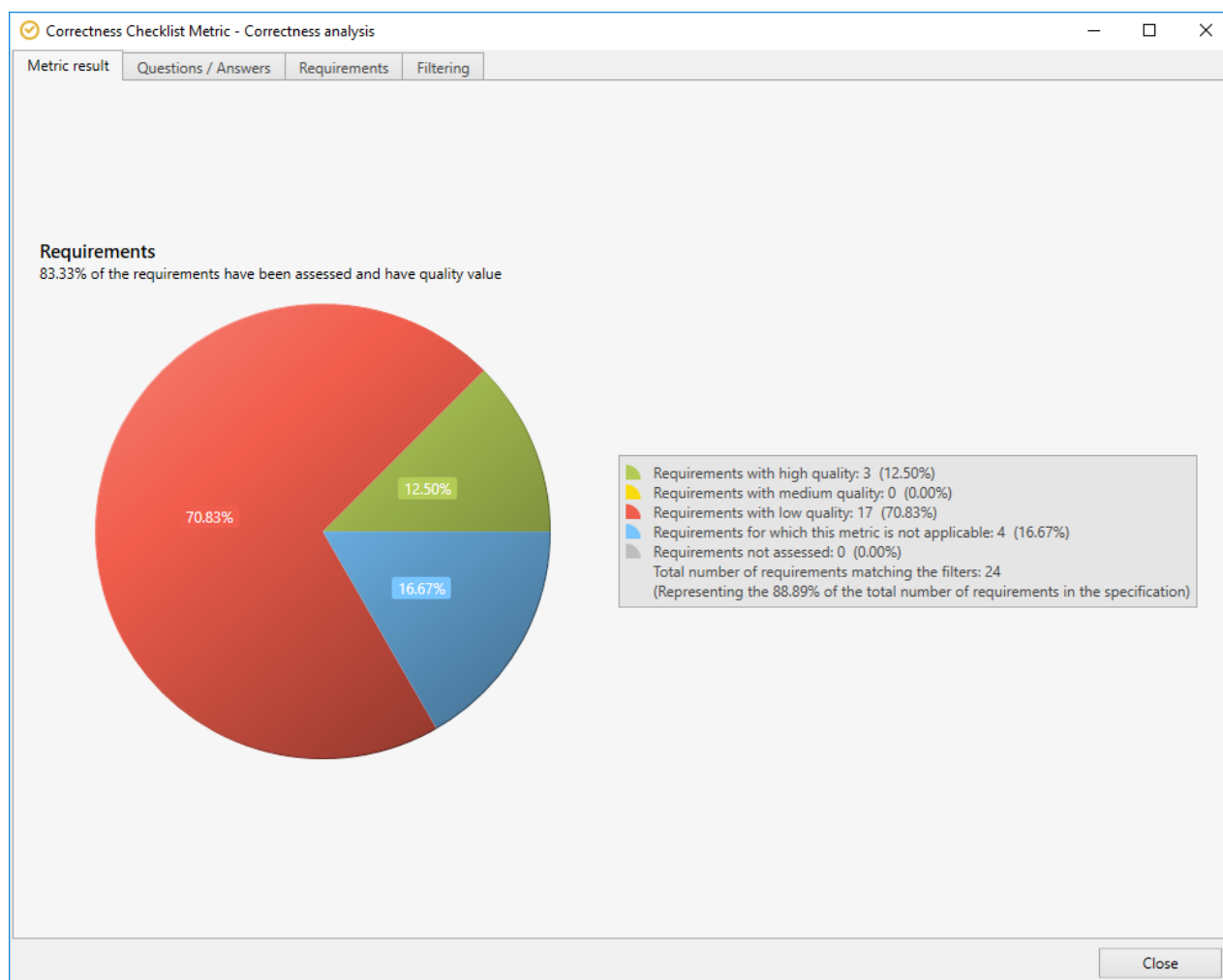


Figure 76. Checklist results statistics



Correctness Checklist Metric - Correctness analysis

Metric result Questions / Answers Requirements Filtering

Requirements:

Drag a column header here to group by that column

Code	Description	Quality value
FRS_060	Every 2 seconds, the power control system shall send a demand battery load level message to the battery	★ ★ ★
72ca031d-b79e-42b4-a579-7e96b4051...	The air conditioning system shall have 1 compressor	★ ★ ★
d4b5378f-99ae-4a66-a3a8-551636dc3e...	The air conditioning system shall have 1 fan	★ ★ ★
b36d052b-67ee-41c3-9a1b-73ba66c7b...	The air conditioning system shall have 3 heater	★ ★ ★
2d466226-86da-415d-a824-8943cef3a0...	The fan shall have 3 fan blade	★ ★ ★
40eabb1e-b14e-4138-a055-84f40ca274f5	The maximum power consumption of the air conditioning system shall be 20 hpw	★ ★ ★
FRS_001	The maximum power consumption of the compressor shall be 15 hp	★ ★ ★
FRS_003	The maximum power consumption of the fan blade shall be 0.8 hp	★ ★ ★
FRS_002	The maximum power consumption of the fan shall be 3 hp	★ ★ ★
ae4a295d-3df5-43fb-abf5-f6654132c534	The maximum power consumption of the heater shall be 7 hp	★ ★ ★
ce1e9d09-810b-407a-ab6e-0d528804f7...	the number of error of the system shall be 0	★ ★ ★
FRS_056	The propulsion shall have 1 gear box	★ ★ ★
FRS_057	The propulsion shall have 1 mechanical transmission	★ ★ ★
FRS_058	The propulsion shall have 1 power converter	★ ★ ★
FRS_059	The propulsion shall have 1 traction control unit	★ ★ ★
FRS_051	The Rolling stock component must have 1 on board vehicle	★ ★ ★
FRS_050	The Rolling stock component must have at least a 1 lighting system	★ ★ ★
FRS_048	The Rolling stock component shall have 1 air conditioning system	★ ★ ★
FRS_049	The Rolling stock component shall have 1 air conditioning system	★ ★ ★

Total: 24

Close

Figure 77. Checklist results for each requirement

## 3.2.5 Functional Refinement

### 3.2.5.1 Architectural Refinement

#### 3.2.5.1.1 Architectural Refinement in CHES

The architectural refinement defines in detail how the different parts of the system are connected and interact in order to fulfil the system requirements.

To refine the architecture, the sub requirements and subcomponents of the system must be identified.

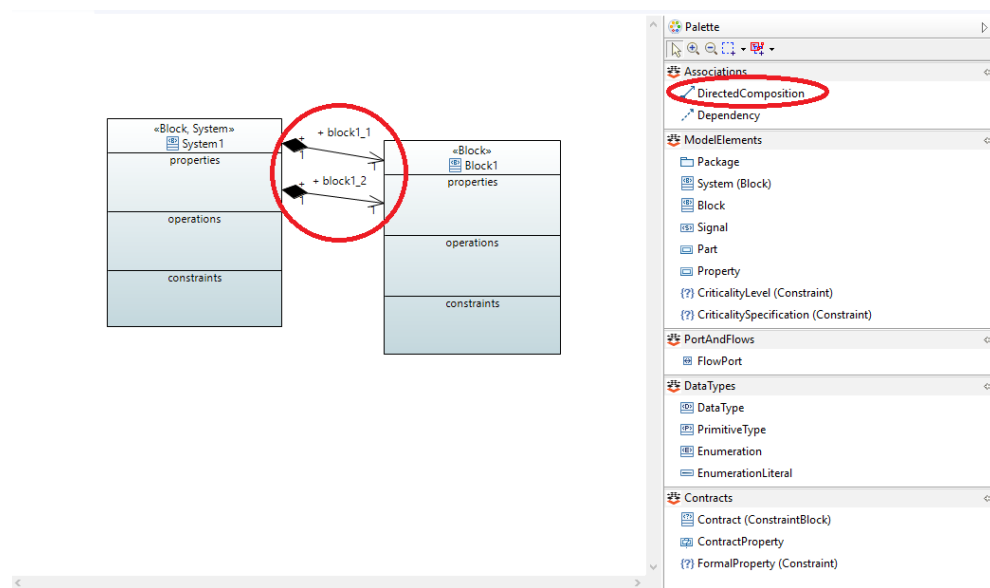
Allocation of requirements/sub-requirements to subcomponents must be managed (see section 3.2.1.1), as well as formalization of sub-requirements (see section 3.2.2).

In case of reuse of components (as subcomponents), associated satisfied requirements and implemented contracts are reused.

Information about requirements refinement can be managed in Papyrus/CHES by using the SysML standard support, in particular by using the DeriveReq relationships from the child to the parent requirement.

Then, subcomponents must be connected through their interfaces. To refine the system, perform the following steps:

1. Prepare the graphical editor environment: In the “Model Explorer” view, go to the package “modelSystemView” and open the previously created Block Definition Diagram, see Section 3.2.1.
2. Define the instances of the components: Using the palette of the diagram editor, select the “DirectedComposition” element and respectively select first the component to refine (e.g. C1) and then the chosen subcomponent (e.g. C2) in the editing space, see Figure 78. As result, in the “Model Explorer” view a property of type C2, that is an instance of C2, is created in C1. It is possible to create more than one instance of C2 in C1.

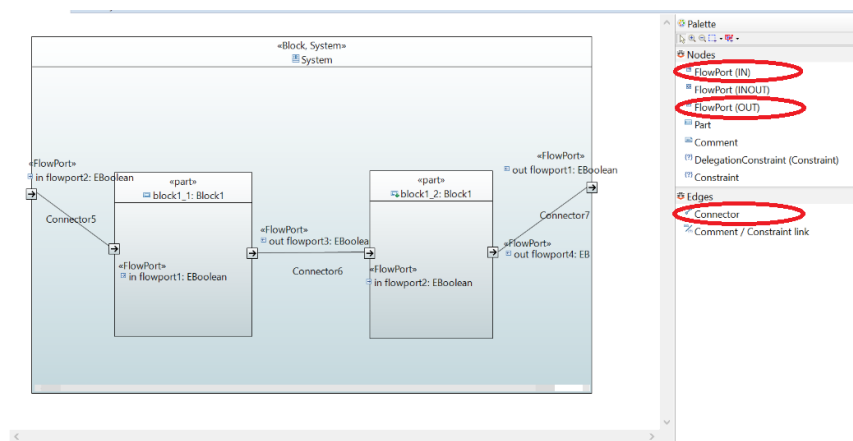


**Figure 78.** Two instances of the component Block1 compose the System component

To connect the input/output ports among components, perform the following steps:

1. Prepare the graphical editor environment: In the “Model Explorer” view, go to the package “modelSystemView” and create and open an Internal Block Diagram belonging to the component to refine. If the component to refine is hardware, then create a Composite Diagram instead. As result, a representation of the component to refine is shown in the graphical editor.
2. Select subcomponents: In the “Model Explorer” view, navigate through the elements that belong to the main component and drag the component instances (also called component parts, created in step 2 of the previous process), into the graphical editor. Ports already defined for the subcomponents can be shown in the Internal Block/Composite Structure Diagram by drag and drop them from the “Model Explorer” to the given component part available in the diagram. New ports can also be created through the Internal Block/Composite Structure Diagram for a given part: it is worth noting that in the model, ports are actually attached to the definition of the component, not just to the component part/instance itself. So, creating a port for a part actually changes the definition of the component represented by the part; this also means that other parts of the same component, if defined somewhere in the hierarchical refinement, will “inherit” the new created port.
3. Connect the components: Using the palette of the diagram editor, select the “Connector” element and connect the ports, see Figure 79. Note that:
  - a. input ports of the main component must connect with input ports of subcomponents,
  - b. output ports of subcomponent must connect with input ports of subcomponents or with output ports of the main component.

Alternatively, it is possible to connect the ports using the “DelegationConstraint” element. With this element, the user can define an assignment in the OSS language, enabling the possibility to connect more ports with one port.

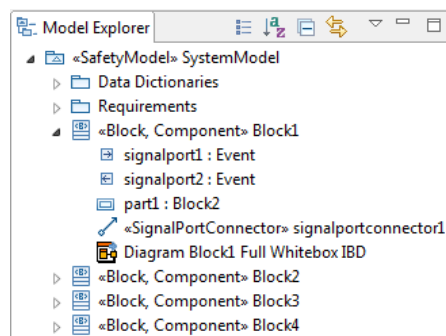


**Figure 79.** Internal Block Diagram used for architectural refinement

### 3.2.5.1.2 Architectural Refinement in SAVONA

Defining the system's architecture in SAVONA completely relies on the creation of the system model using SysML's Internal Block Diagrams. A *Block* is automatically refined by its sub-components, called *Parts* which are connected through *Ports* and *Connectors*. SAVONA thereby extends the default Papyrus diagram editor by the following functions:

- **Check for cycles:** Before a new part is being created, SAVONA checks whether a cycle in the architecture will arise. Since a cycle in the architecture definition results in an invalid system model, SAVONA will notify the user and forbid the creation of such refinement.
- **System model consistency:** A valid system model hierarchy is essential to the model's validity. SAVONA therefore restricts the operations on the model hierarchy that its validity is always preserved. Figure 80 shows a valid model hierarchy: System *Blocks* or *Types* can only be defined at the top level of the system model. *Ports*, *Parts* and *Connectors* are only associated to a Block. Any other hierarchical order of elements is forbidden per default; although reuse of model elements is allowed (see *Reuse of elements*).



**Figure 80.** Model Explorer showing a valid system architecture hierarchy in SAVONA

- **Reuse of elements:** The copy&paste functionality in the Model Explorer and Diagram Editor have been enhanced to support the idea of a valid system model hierarchy. Model elements such as ports, parts etc. can easily be copied over to other blocks. SAVONA automatically inserts the copied elements at the correct hierarchy level.
- **Full-Whitebox Diagrams:** Since the hierarchical view of a block in the Model Explorer is often not enough to give a sufficient overview of a system component, SAVONA maintains a full whitebox diagram for each component. The diagram is automatically updated on any related model change and allows a quick view on the current architecture of a component, regardless of any specific view.

### 3.2.5.1.3 Architectural Pattern-based support

A design pattern is a reusable solution to a commonly occurring problem within a given context in the system design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system<sup>12</sup>.

A useful aspect of a design pattern is that a solution comes with advantages and inconveniences. These could for instance relate to non-functional properties likely reliability, performance and resource consumption and are therefore interesting for embedded and cyber-physical systems. In some cases, design patterns can also describe undesired solutions, for instance ad-hoc solution with known disadvantages that are also known as anti-patterns.

### **Pattern integration**


CHESS-Papyrus supports the design pattern integration, i.e. the use of design patterns in an application or system model according to the logical approach described in D3.3 [28]. The process of integrating a pattern is split into several parts.

1. Identify which pattern should be applied

Select a suitable pattern for a problem that needs to be solved in a certain context. Figure 81 shows the pattern selection dialog once the pattern library coming with the CHESS editor is imported in the user model (see "Pattern Libraries" section).

---

<sup>12</sup> [https://en.wikipedia.org/wiki/Software\\_design\\_pattern](https://en.wikipedia.org/wiki/Software_design_pattern)



Select a Design Pattern

Select a design pattern from the list and click "apply" to apply it to the model

Available Patterns

Triple Modular Redundancy Pattern (TMR) (2-oo-3 Redundancy Pattern, Homogeneous Triplex Pattern)

Monitor-Actuator Pattern

Intent/Context

Developing an embedded system with no fail-safe-state in a situation that includes high random failure rate and no limitation on redundancy, with the purpose of improving safety and reliability of the

Problem

How to deal with random faults and single-point of failure in order to increase the safety and reliability of the system without losing the input data in the

Solution/Pattern Structure

This pattern contains three identical modules or channels operate in parallel. This structure is used to prevent the failure of a single component, which may lead to a complete system failure. If a single fault occurs in one channel then the other two channels will continue to work correctly and produce the correct actuation control signals.

Consequences

The main drawback for this pattern is that it is not appropriate for systematic faults handling. In this case the three channels are identical and have the same possible fault, and the system will continue to work

Implementation

To implement this pattern, the designer should replicate the channel which includes the replication of the hardware as well as software. With respect to the sensor, there are two options either to use a

Pattern Assumptions

The voter is a simple component that is carefull designed with reliability ~ 1

Pattern Guarantees

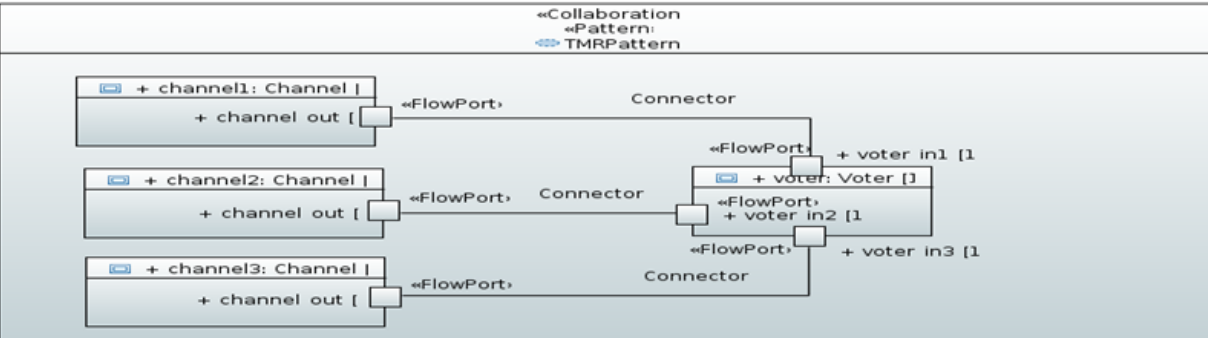
$R = 3R_{channel}^2 - 2R_{channel}^3$

Pattern Preview

«Collaboration

«Pattern:

«TMRPattern



Apply

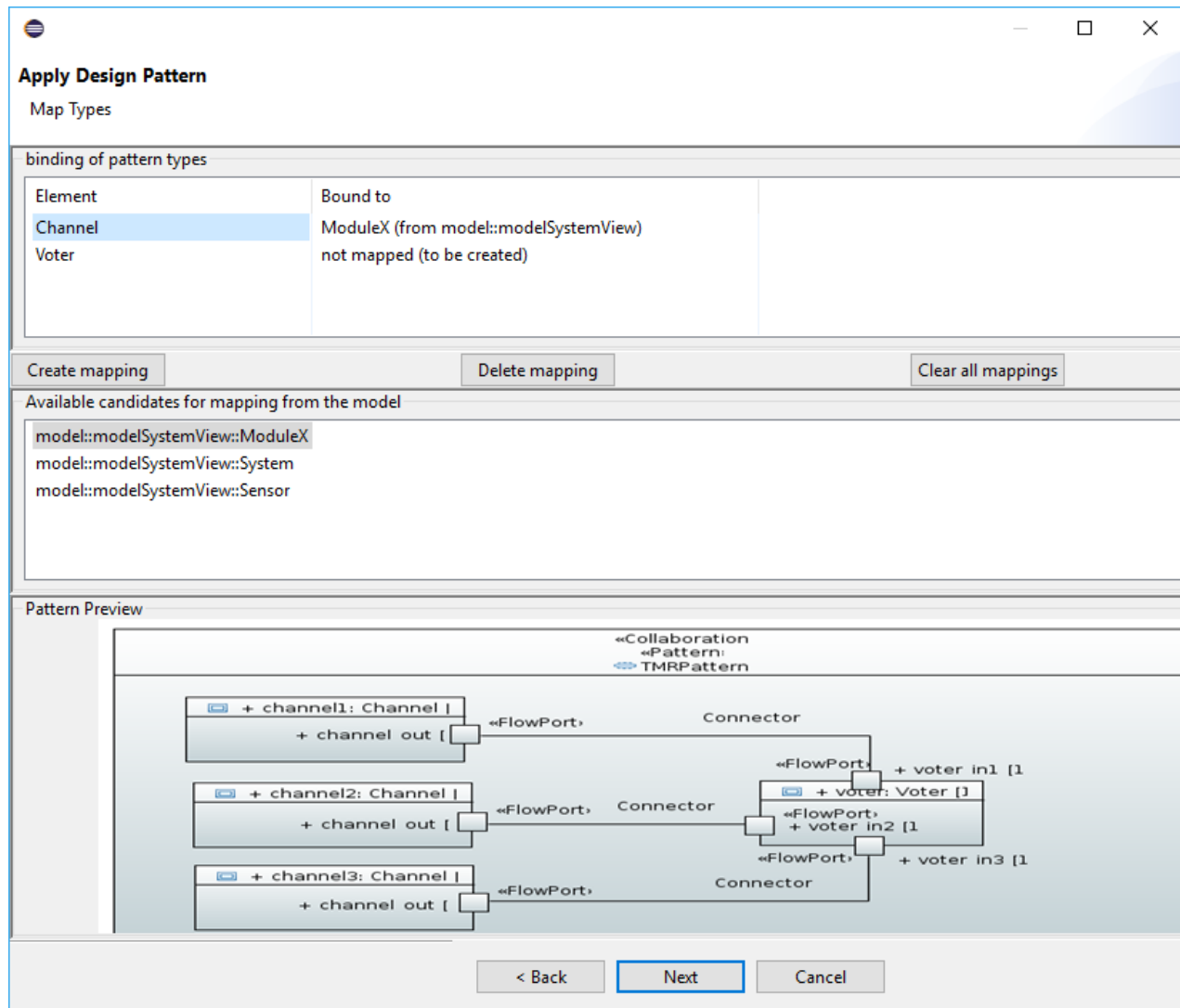
Cancel

**Figure 81.** Design pattern selection dialog

## 2. Role binding

A design pattern describes a set of roles that elements play in a pattern. In many cases, elements that play a certain role do already exist in the application model. Therefore, it is important to identify these and declare a binding to role in the pattern (i.e. a role binding). This information is used to determine which elements of a pattern need to be copied into the application and which don't.

The following Figure 82 shows one of the role binding dialogs available for the Triple Modular Redundancy pattern. The upper part shows the roles defined by the pattern, while the lower part shows the available candidates for binding in the system under design. Candidate matching simply relies on the meta-model kind, i.e. components match components, components parts to components parts, ports to ports, connections to connections; dedicated binding dialogs are available for each kind of meta-model entity by using the "Next" button of the main dialog window. In order to declare a binding, select a pair in the upper and the lower part of the dialog, respectively and then click on the "Create mapping" button.



**Figure 82.** Role binding interface

### 3. Integrate the pattern

This automatic step implies that elements that are in the pattern but not already in the model are copied.

### Pattern libraries

CHESSE is released with a pattern library including some safety patterns.

Each pattern comes with a set of information like the description of the problem, the intention behind possible solutions, roles within the pattern and a solution description; moreover assumption-guarantee properties are captured, in the contracts style; this can serve as the basis for assuring that the application of the architectural pattern adequately addresses the problem that is trying to solve (see D3.3 [28] section 2.5.1).

The library can be imported in application models and will then be available in the design pattern catalog that is presented when you choose to apply a design pattern via the designer context menu. Also, the library can be enriched with additional patterns; please refer to the AMASS user manual [33] for more information about importing pattern libraries and new patterns definition.

### 3.2.5.1.4 Parameters and Configurations

The parameterization of an architecture is the procedure to define and use a (possibly infinite) set of parameters to set the number of components, the number of ports, the connections, and the static attributes of components.

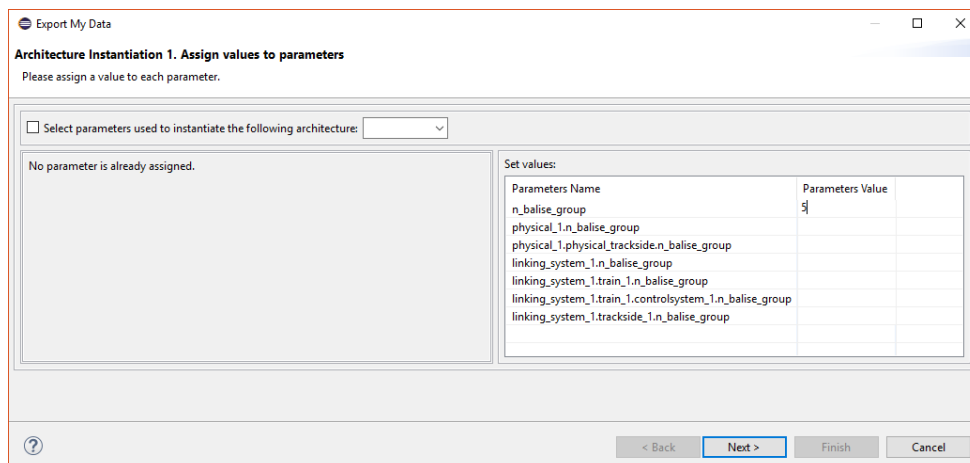
To parameterize an existing architecture, perform the following steps:

1. Create the parameter: In the Model Explorer View or in the BDD Editor, create a static FlowPort. The static attribute can be set to “true” in the “SysML” tab of the “Properties” view.
2. Give an assignment to the parameter (optional step). In the Model Explorer View or in the IBD Editor, select the owner of the parameter, create the “DelegationConstraint” element (see Section 3.2.5.1) and set a value or an expression to the parameter.
3. Give a constraint to the parameter (optional step). In the Model Explorer View, select the owner of the parameter, create the “Constraint” element and write a boolean expression in the OSS language.

The instantiation of an architecture is the procedure to assign the values of the parameters used in the input parameterized architecture. The outcome is an architecture with a fixed number of components, ports, connections, and static attributes of components.

To instantiate a parameterized architecture, perform the following steps:

1. Select the root component of the parameterized architecture: Select the root component (in the “Model Explorer” view) or the corresponding graphical representation (in the diagram editor) and open the menu using the right-button of the mouse. Select “Instantiate the parameterized architecture”, then a wizard appears.
2. Select the parameters used in existing instantiated architecture (optional step).
3. Assign a value to each parameter, see Figure 83. This step may require more iteration; a parameter may depend on another parameter, so the latter needs to be set first.
4. Import the instantiated architecture into the project, see Figure 84. Select the destination package on the right side of the wizard page.



**Export My Data**

**Architecture Instantiation 1. Assign values to parameters**

Please assign a value to each parameter.

☐ Select parameters used to instantiate the following architecture: v

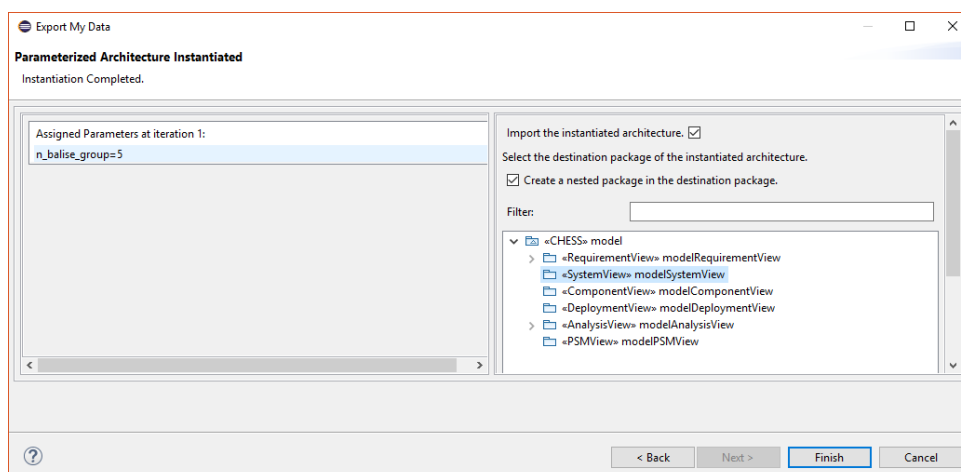
No parameter is already assigned.

Set values:

Parameters Name	Parameters Value
n_balise_group	5
physical_1.n_balise_group	
physical_1.physical_trackside.n_balise_group	
linking_system_1.n_balise_group	
linking_system_1.train_1.n_balise_group	
linking_system_1.train_1.controlsystem_1.n_balise_group	
linking_system_1.trackside_1.n_balise_group	

? < Back Next > Finish Cancel

**Figure 83.** Wizard to set the parameters of the parameterized architecture



**Figure 84.** Last page of the wizard to import the instantiated architecture into the current project

### 3.2.5.2 Contract Refinement

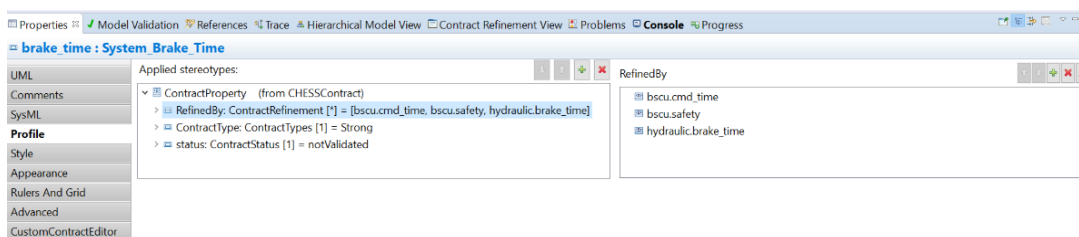
#### 3.2.5.2.1 Contract Refinement in CHES

The contract refinement requires one prior phase, the definition of the contract (see Section 3.2.2).

Contracts refinement must follow the requirements refinement. Indeed, a contract is implicitly linked to the formalized requirements (via its assumption and guarantee properties); so, the requirements referred by contracts appearing in a refinement relationship should also have a corresponding refinement traced (see section 3.2.5.1).

To refine the contract, perform the following steps:

1. Select the contract to refine: Select the contract instance (in the “Model Explorer” view) or the corresponding graphical representation (in the diagram editor) and open the menu using the right-button of the mouse. Select “Contract-Set contract refinement”. A popup appears showing the list of contracts that belong to the subcomponent of the current component. Select the refining contracts.
2. Edit the refining contracts: Select the contract instance (in the “Model Explorer” view) or the corresponding graphical representation (in the diagram editor) and open the tab “Profile” in the “Properties” view. In the area “Applied stereotypes” select “ContractProperty – Refined by”. Then, remove the contracts or browse the model and set one or more refining contracts.

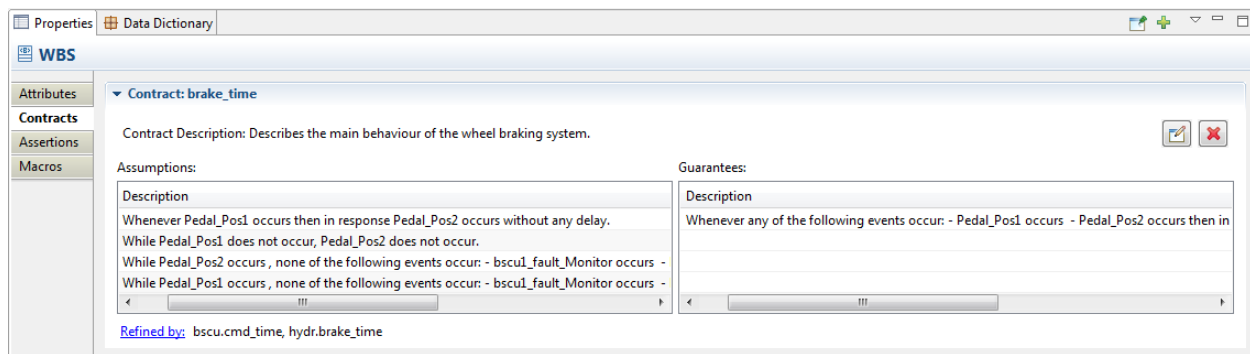


**Figure 85.** “Profile” tab to edit the refining contracts

#### 3.2.5.2.2 Contract Refinement in SAVONA

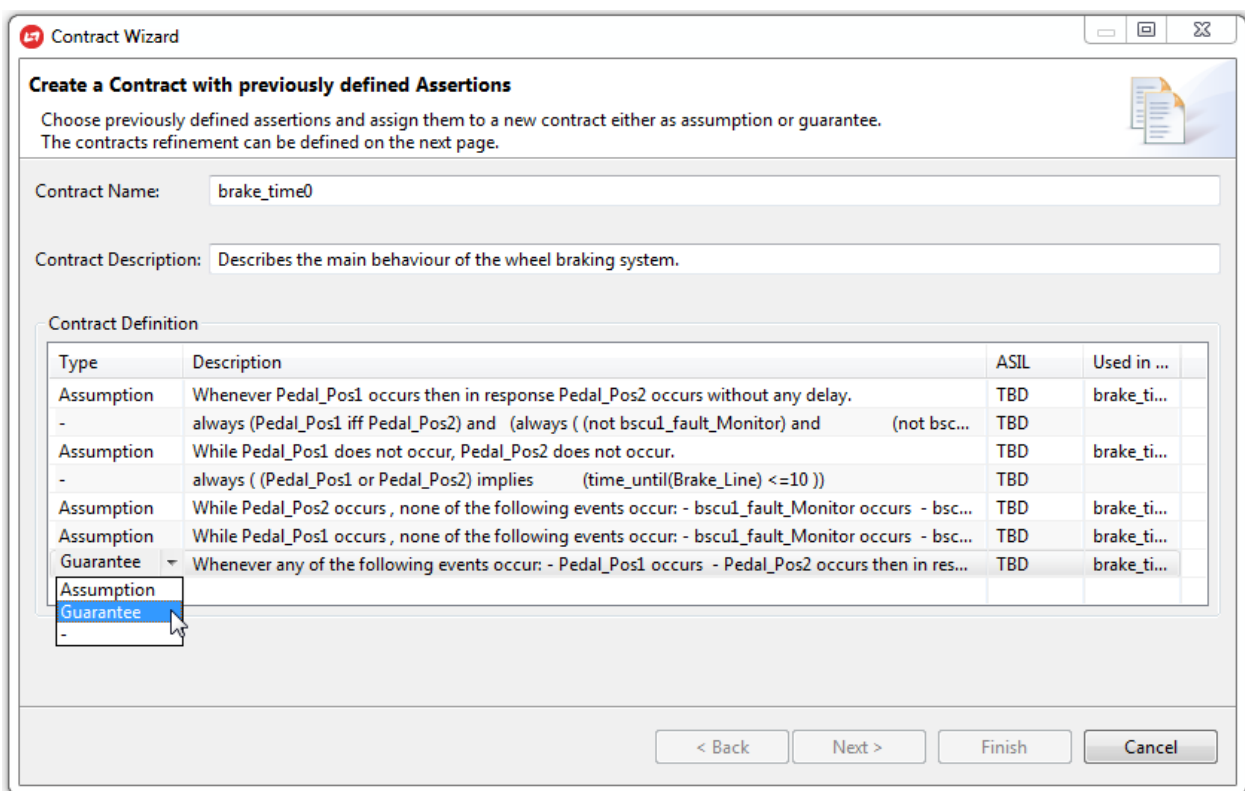
To specify a contract within SAVONA, assertions must be gathered that are later used either as assumptions or guarantees. The Assertions Section of the Properties View (see Figure 32) shows assertions that are defined for the currently selected SysML block, interface or connector. As contracts can only be defined on SysML blocks via the Contracts Section (see Figure 86), the Contracts section shows an overview of all contracts assigned to the currently selected type and allows the editing, creation and deletion of contracts.





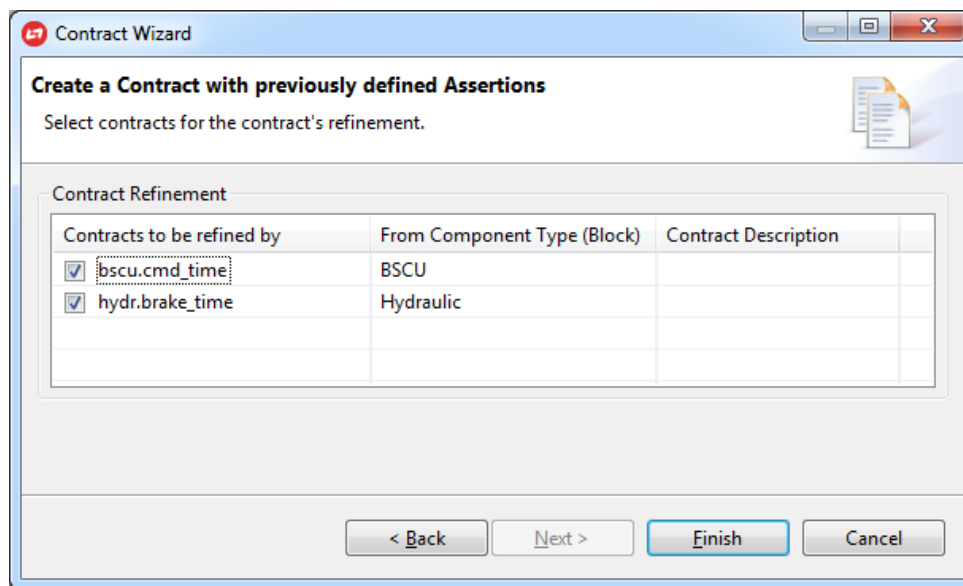
**Figure 86.** Contracts Section in the Properties View of SAVONA

When defining a new contract in SAVONA, the Contract Wizard is used to ease the process of assigning assertions as assumptions or guarantees. Previously defined assertions can be used to create a contract for a component. The wizard offers assertions that are defined on the currently selected SysML block and the owned ports of the block (see Figure 87). To use one of the offered assertions in a contract, simply assign a type (either assumption or guarantee) to it. At least one guarantee is needed to create a contract. Multiple assumptions as well as multiple guarantees are conjunct. Assertions without any type assignment will not be considered in the contract definition.



**Figure 87.** Contract Wizard of SAVONA: Assignment of assumptions and guarantees

The contract refinement can be defined at the last page of the Contract Wizard (see Figure 88). Contracts from parts can be selected that refine the currently selected component.



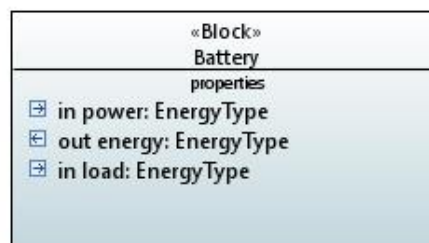
**Figure 88.** Contract Wizard of SAVONA: Definition of contract refinements

### 3.2.6 Component's Nominal and Faulty Behaviour Definition

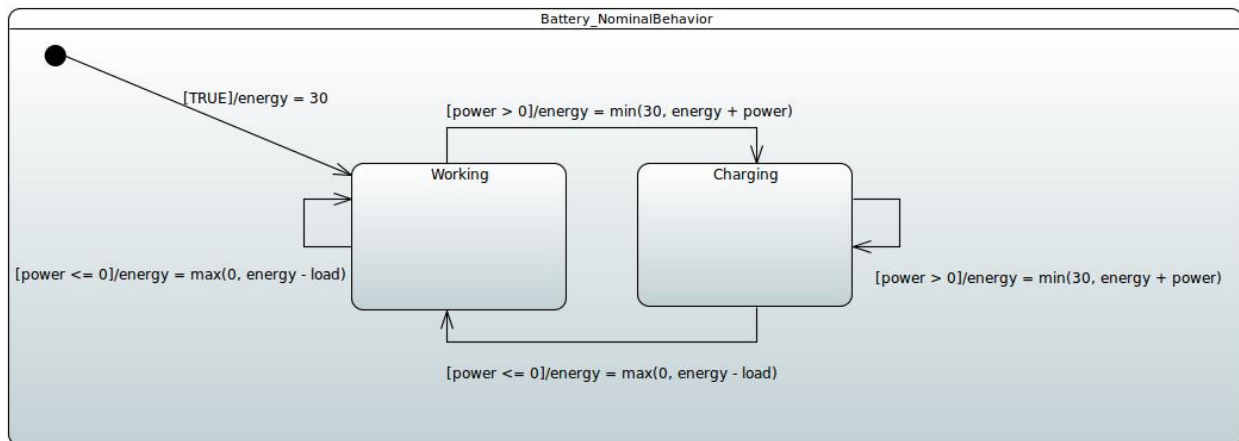
System component static definition, i.e. its properties, ports, contracts, can be enriched with behavioural models by means of UML state machine diagrams. In particular, in CHES, state machine diagrams can be used to model the nominal and the faulty behaviour of the component.

Nominal state machine are basically standard UML ones. In CHES, by using a subset of the UML support for state machine modelling, it is also possible to use model transformation to automatically generate an SMV (the input language of nuXmv) representation of the modelled nominal state machines, together with the components structure definition (this is then used to perform automatic generation of fault tree, see Section 3.3.2.2).

Figure 89 below is an example of component with its properties (input and output ports), while Figure 90 shows its nominal state machine which follows the restriction needed to run the CHES to NuSMV3 generation: basically, only UML initial, final, basic states and transitions are considered for the current implemented transformation (e.g., no UML concurrent regions, history states, choice or junction transition). A transition comes with a guard and an effect. The guard is a boolean condition upon the values of components properties. The effect must be expressed by using the SMV language, to model component properties assignment.



**Figure 89.** Example of component



**Figure 90.** Example of nominal state machine (using SMV as action language)

The behavioural model of the component can be enriched with the faulty behaviour. Faulty behaviour is modelled in CHES by using the dedicated dependability profile. The CHES dependability profile enables dependability architects to model dependability information necessary to conduct dependability analysis, basically by allowing the modelling of failure modes and criticality, the failure behaviours for a component in isolation (so how events can lead to components failure modes) and the failure modes propagation between connected components (via components ports). More information about this is provided in Section 3.3.2.

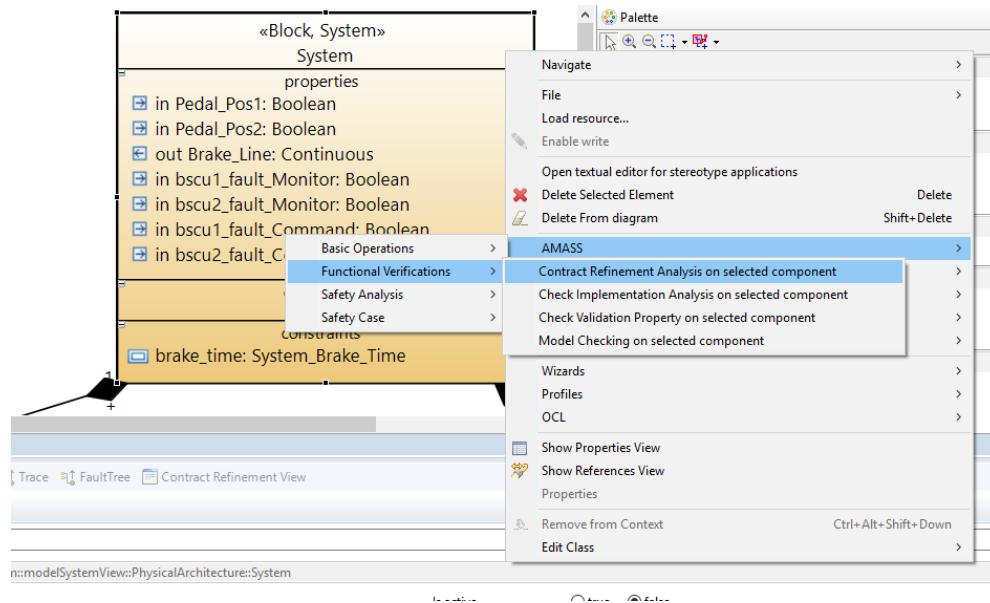
## 3.2.7 Functional Early Verification

### 3.2.7.1 Contract-Based Verification of Refinement

As an early verification of the architectural decomposition, the user can use the tool to verify if the contract refinement is correct (see Section 2.1.2). This verification step requires to have specified a contract refinement.

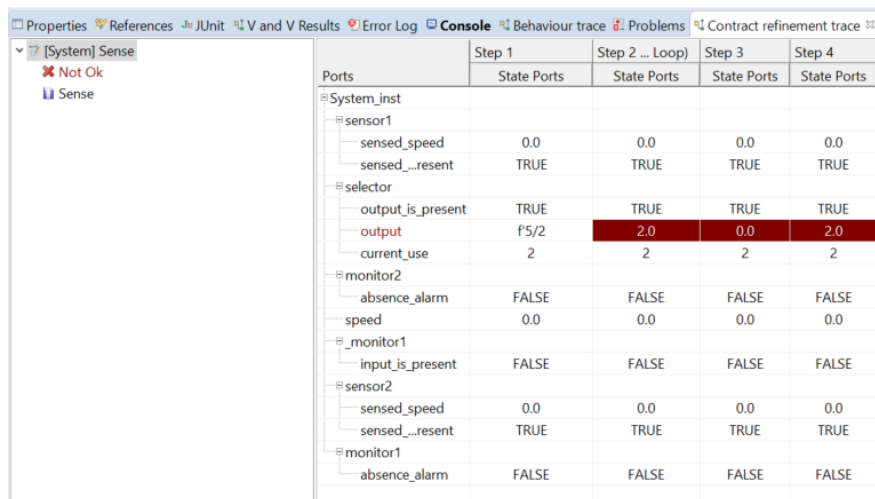
To verify that the contract refinements are done correctly, perform the following steps:

1. Choose which contract refinements must be checked: select a component (in the “Model Explorer” view) or the corresponding graphical representation (in the diagram editor). The contract refinements considered will be the ones associated to the selected component and the ones associated to its sub components. This operation includes recursively all the contracts along the subcomponents, from the root to the leaves of the system.
2. Perform the check contract refinement: right click on the selected component, then go to “AMASS-Functional Verification” – “Check Contract Refinement on Selected Component”, see Figure 92.



**Figure 91.** Dedicated menu to perform the check of the contract refinements

3. Receive the result of the analysis: when the analysis is completed, it is possible to see the status of each refinement in the “Trace” view.
4. If the check fails, it is possible to see the counter example, i.e. the instances of values to assign to the ports that cause the failure of the contract refinement. For example, the Figure 92 shows that the refinement of the contract “System\_Brake\_time” of the component “system” is failed. Looking at the list of values of the counter example, the user is facilitated to figure out whether the error is caused by the assumption or guarantee of the involved contracts, or by the choice of the refining contracts.



	Step 1	Step 2 ... Loop	Step 3	Step 4
<b>Ports</b>	<b>State Ports</b>	<b>State Ports</b>	<b>State Ports</b>	<b>State Ports</b>
System_inst				
sensor1				
sensed_speed	0.0	0.0	0.0	0.0
sensed_resent	TRUE	TRUE	TRUE	TRUE
selector				
output_is_present	TRUE	TRUE	TRUE	TRUE
output	f5/2	2.0	0.0	2.0
current_use	2	2	2	2
monitor2				
absence_alarm	FALSE	FALSE	FALSE	FALSE
speed	0.0	0.0	0.0	0.0
_monitor1				
input_is_present	FALSE	FALSE	FALSE	FALSE
sensor2				
sensed_speed	0.0	0.0	0.0	0.0
sensed_resent	TRUE	TRUE	TRUE	TRUE
monitor1				
absence_alarm	FALSE	FALSE	FALSE	FALSE

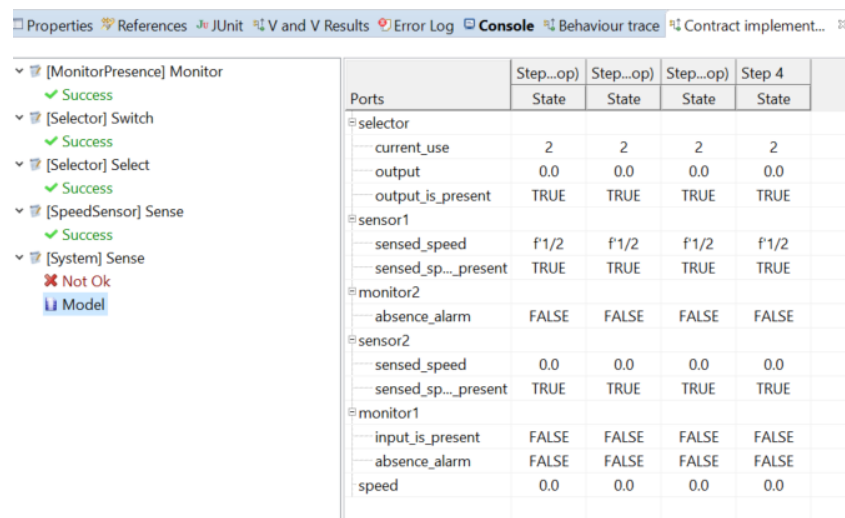
**Figure 92.** An example of the result of the check contract refinement shown in the Trace View. In this case, it is possible to see that one refinement is failed, and the counter example.

### 3.2.7.2 Contract-Based Verification of State Machines

As a further early verification step, the user can use the tool to verify that the behavioural specification of the system is compliant with the contract specification. This is currently restricted to behaviours specified by state machines.

To verify that the state machines defined in the model satisfy the contracts, perform the following steps:

1. Choose which contracts and state machine are involved: Select a component (in the “Model Explorer” view) or the corresponding graphical representation (in the diagram editor). The contracts and state machines considered will be the ones associated to the selected component and the ones associated to its sub components. This operation includes recursively all the contracts and state machines along the subcomponents, from the root to the leaves of the system.
2. Perform the validation: right click on the selected component, then go to “AMASS-Functional Verification” – “Check contract implementation on Selected Component”.
3. Receive the result of the analysis: when the analysis is completed, the status of the check for each contract is shown in the “Trace” view. If the check fails, it is possible to see the counter example, i.e. instances of values to assign to the ports that cause the not satisfiability of the contract for the specific state machine, see Figure 93.



	Step...op)	Step...op)	Step...op)	Step 4
Ports	State	State	State	State
selector				
current_use	2	2	2	2
output	0.0	0.0	0.0	0.0
output_is_present	TRUE	TRUE	TRUE	TRUE
sensor1				
sensed_speed	f1/2	f1/2	f1/2	f1/2
sensed_sp..._present	TRUE	TRUE	TRUE	TRUE
monitor2				
absence_alarm	FALSE	FALSE	FALSE	FALSE
sensor2				
sensed_speed	0.0	0.0	0.0	0.0
sensed_sp..._present	TRUE	TRUE	TRUE	TRUE
monitor1				
input_is_present	FALSE	FALSE	FALSE	FALSE
absence_alarm	FALSE	FALSE	FALSE	FALSE
speed	0.0	0.0	0.0	0.0

**Figure 93.** An example of the result of the check contract implementation shown in the “Trace” view. It is possible to see that four contracts are not satisfied by the state machines of their associated component and one contract not.

### 3.2.7.3 Model Checking

Besides the verification of contracts, the user may want to verify other properties on the state machine.

The specifications to be checked on the FSM can be expressed in temporal logics like Computation Tree Logic (CTL), Linear Temporal Logic (LTL) extended with Past Operators, and Property Specification Language (PSL) that includes CTL and LTL with Sequential Extended Regular Expressions (SERE), a variant of classical regular expressions. It is also possible to analyse quantitative characteristics of the FSM by specifying real-time CTL specifications.

CTL and LTL specifications are evaluated by nuXmv in order to determine their truth or falsity in the FSM. When a specification is discovered to be false, nuXmv constructs and prints a counterexample, i.e. a trace of the FSM that falsifies the property.

There are three possible model checking available (see [16] for more details):

- check ctlspec: it performs fair CTL model checking. A CTL specification is given as a formula in the temporal logic CTL. A CTL formula is true if it is true in all initial states.
- check invar: it performs model checking of invariants. Invariants are propositional formulas which must hold invariantly in the model.
- check ltlspec: it performs LTL model checking. An LTL formula is true if it is true at the initial time  $t_0$ .

To execute the model checking, perform the following steps:

1. Choose the components behaviour to check: Select a component (in the “Model Explorer” view) or the corresponding graphical representation (in the diagram editor). The components behaviour to check will be the behaviour of the selected component and the behaviour of its sub components. This operation includes recursively all the behaviours from the root to the leaves of the selected component.
2. Perform the check: right click on the selected component, then go to “AMASS-Functional Verification” – “Model Checking on Selected Component”.
3. Analyse the results of the verification

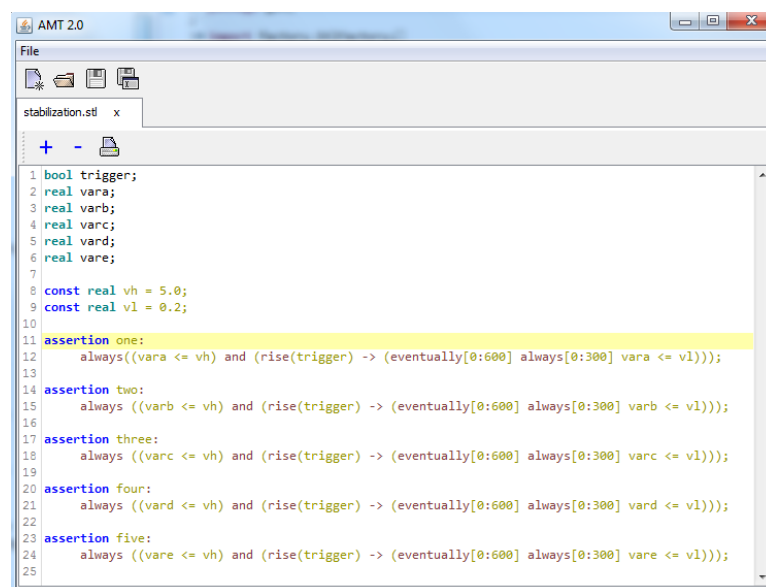
### 3.2.7.4 Contract-Based Monitoring

The contract-based monitoring technology can be used as a functional early verification technique that provides confidence in the models obtained during the concept design phase. This section describes how the AMT2.0 tool can be used to check the correctness of STL contracts with respect to MATLAB/Simulink simulation traces. The tool offers the following main functionalities:

- Checking the correctness of the simulation traces with respect to requirements formalized in STL in an offline fashion.
- Explaining contract violations in terms of temporal implicants – small simulation trace segments that are sufficient to imply the contract violation.
- Property-driven measurements from simulation traces.

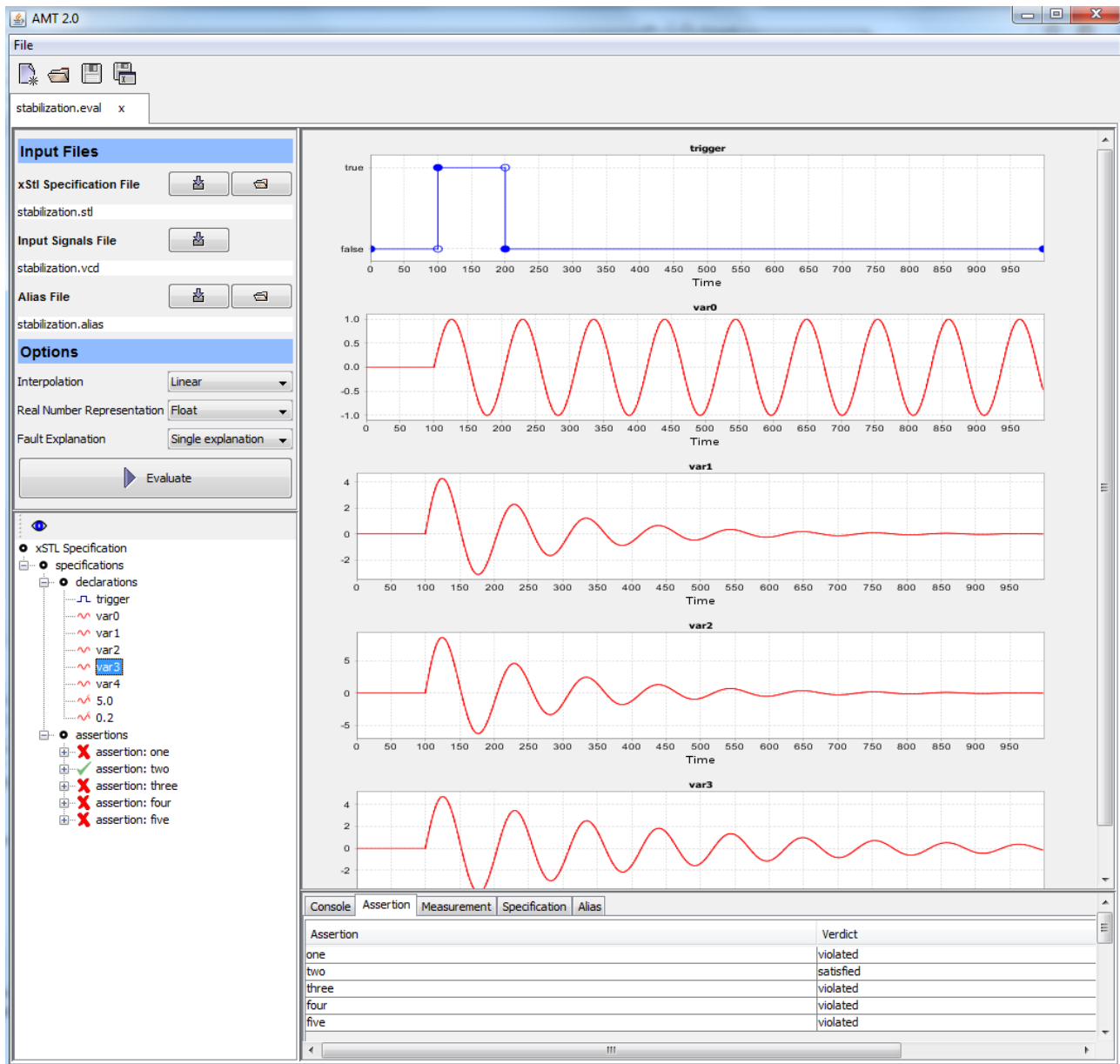
In this section we focus on the first two features of the tool and on its GUI interface. The user first needs to formalize functional requirements in the form of STL contracts. The tool provides an STL editor with syntax highlighting for expressing the temporal properties (see Figure 94). Then the user executes the MATLAB/Simulink model and save the simulation traces in the Comma Separated Value (CSV) format. The generation of test inputs can be done according to some coverage criteria. However, the test generation strategy is outside the scope of the tool.

The user then creates a new Evaluation view in the AMT2.0 tool and import both the contract and the simulation traces. Note that STL is interpreted over continuous-time signals while the simulation traces consist of a finite collection of (timestamp, value) pairs. As a consequence, the user needs to interpret the signal values that are in between two consecutive sample points. The tool allows the user to choose between the Step and the Linear interpolation. The user can also choose whether the real numbers are represented using Floats (more efficient representation) or Rationals (more precise representation).



**Figure 94.** AMT2.0 STL contract editor

After setting up the evaluation options, the user clicks the “Evaluate” button that triggers the offline monitoring of the property. In the Property View of the Evaluation tab, the tool displays the parse tree of the contract and highlights which assertions were satisfied and which were violated. The user can expect the satisfaction signal for each sub-formula of the contract indicating at what times the sub-formula was true and at what times it was false. The Evaluation view is depicted in Figure 95.

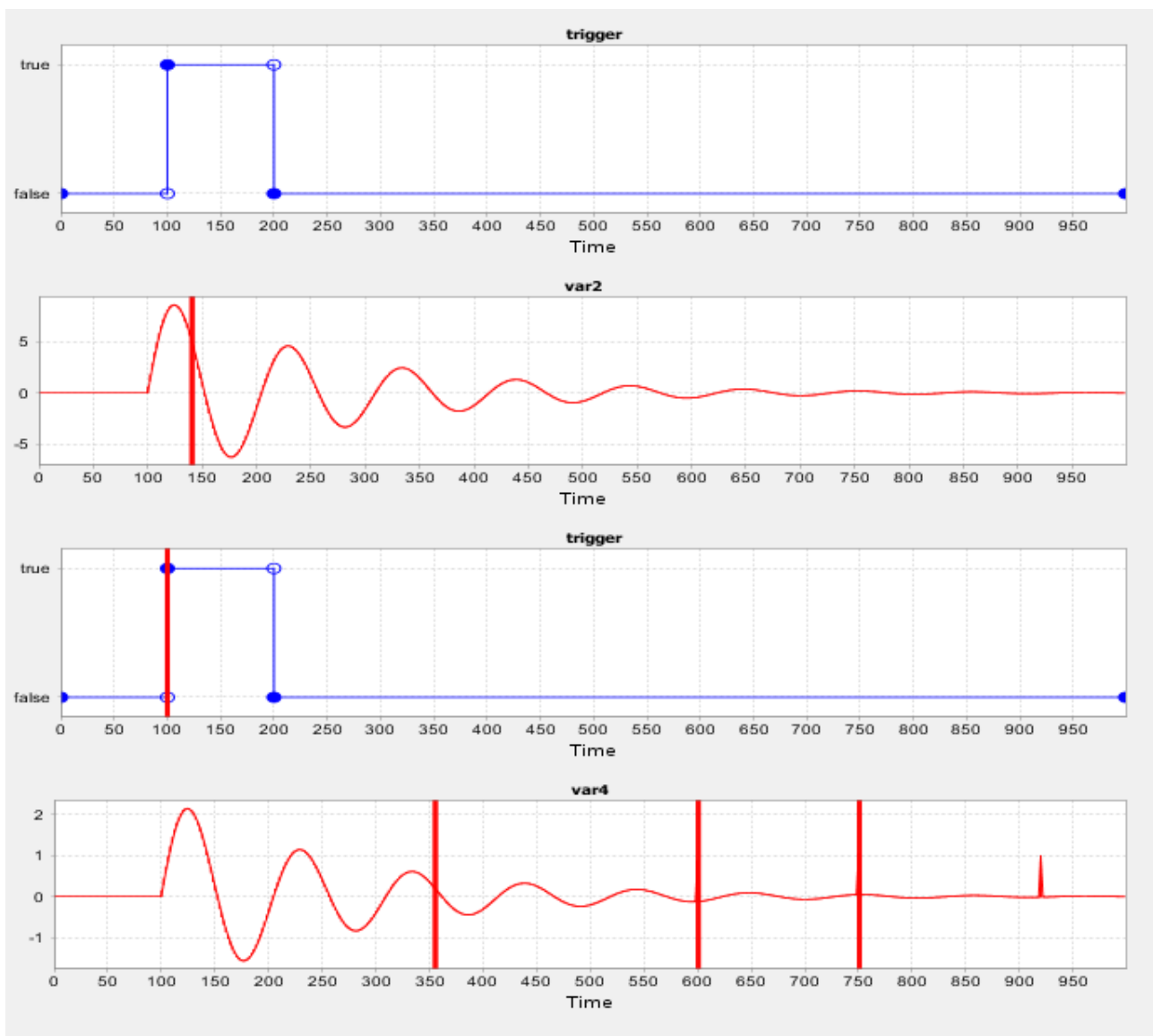


**Figure 95.** AMT2.0 Evaluation view

Finally, for the contract that was violated, the user can get the fault explanations in a graphical form. In particular, the tool highlights (red vertical bands) the input signal segments that were responsible for the violation of the contract. An example of the diagnostics outcomes is shown in Figure 96.

Finally, note that AMT2.0 can be also run in a batch mode. This allows to automate the testing and monitoring tasks on large numbers of simulations.





**Figure 96.** AMT2.0 diagnostics results

### 3.2.7.5 Contract-based Verification of Refinement for Strong and Weak Contracts

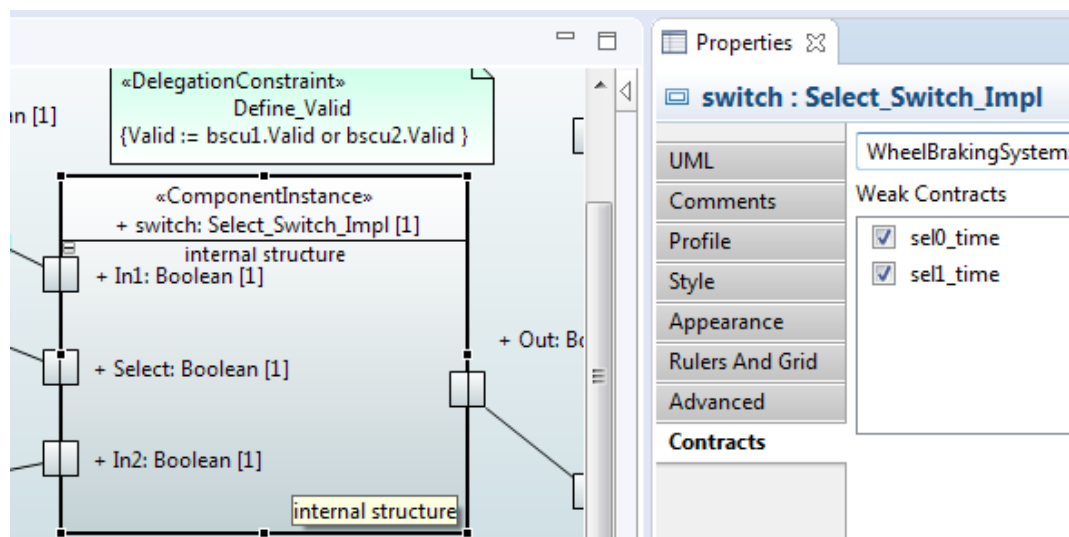
As presented in Section 2.1.2, strong and weak contracts are introduced to support out-of-context reasoning and component reuse across variety of environments. While strong contracts must hold in all environments, the weak ones are environment-specific. Prior to performing the refinement check using strong and weak contracts, contracts must be created and allocated to the component types, which represent out-of-context components. At the component type level, the user indicates if a contract is strong or weak. When the component type is instantiated in a particular system to a component instance, all the strong, and a subset of weak contracts can be identified as relevant in the particular system in which the component is instantiated [83]. As discussed in Section 2.1.2, identifying which are those relevant weak contracts can be done manually on the component instance level. For example, Figure 97 depicts the selected weak contracts for the `Select_Switch_Impl` component instance.

The “Contract Refinement Analysis (OCRA)”[16] command transforms the CHES model into Othello System Specification [16] (.oss) file readable by OCRA, then it runs the OCRA refinement check and outputs the results. Since .oss format does not explicitly distinguish between strong and weak contracts, but treats all contracts as strong, the weak contracts need to be accordingly transformed to strong contracts for .oss

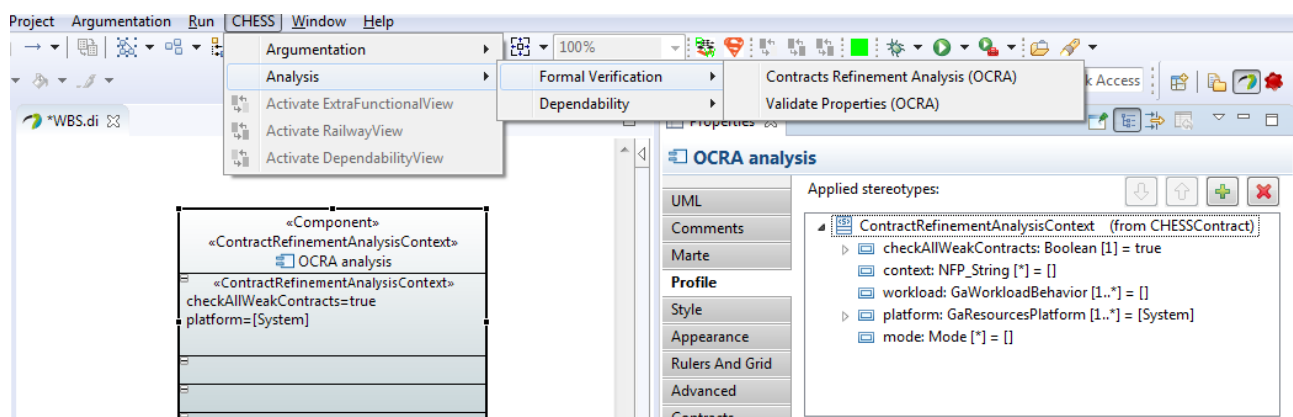


format. To perform the refinement check with strong and weak contracts, the user first creates a class diagram in the “*DependabilityAnalysisView*”, which is a part of the “*AnalysisView*”. Then, the user creates a new “*ContractRefinementAnalysisContext*” by selecting it from the Palette. She selects the newly created “*ContractRefinementAnalysisContext*” and go to the properties view, in particular the Profile tab. There, the user can select the platform that should be analysed (e.g., the system block), and set the attribute “*checkAllWeakContracts*” to true or false (Figure 98). If the “*checkAllWeakContract*” attribute is set to false, the refinement check will be performed such that the selected weak contracts will be treated equally as the strong contracts in the generated .oss file. In this case the user needs to manually ensure that we do not select contradictory weak contracts, otherwise the refinement check will prompt an inconsistency error.

Alternatively, the user can set the “*checkAllWeakContracts*” to true, where the selection will not be needed, but all weak contracts will be included in the generated .oss file such that they will be transformed to implications within the strong guarantees, as described in Section 2.1.2. In this case, to identify which contract is relevant in this particular context, an additional step is needed after running refinement analysis. The user needs to run the command “*Validate Properties (OCRA)*” (found in the menu under CHES->Analysis->Formal Verification, as shown in Figure 98). This command will check validity of each weak contract assumption and identify which weak contracts are relevant in the given system. Upon running the weak contract assumption validity check, the contract status is updated accordingly. It should be noted that the “*Validate Properties (OCRA)*” command can be run only with discrete-time specification, hence the usage of continuous variables or operators in the contracts disables the validity property check.



**Figure 97.** Weak contract selection for a component instance



**Figure 98.** Performing refinement analysis with strong and weak contracts

With this adapted usage of OCRA from the CHES environment, the AMASS platform is capable of automatically verifying contracts for reusable components, i.e., not only strong, but also the weak contracts [67]. Instead of manually selecting the weak contracts, the CHES environment executes different OCRA commands to identify which weak contracts should be selected and whether the refinement holds.

### **3.3 Safety Analysis**

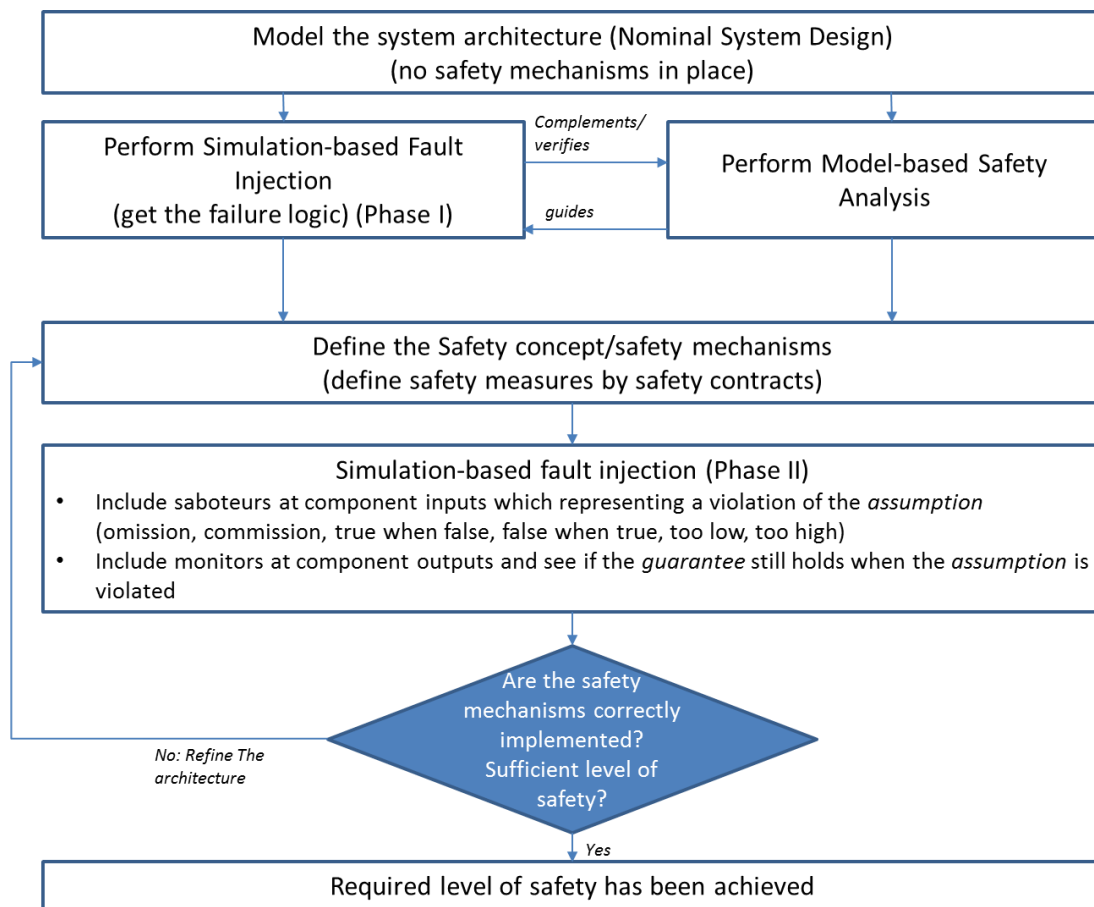
In the following, the Safety Analysis activities depicted in Figure 17 are detailed, one subsection for each activity.

#### **3.3.1 Simulation-based Fault Injection**

Fault injection emerges as a way to perform a simulation-based safety analysis at the same time that helps verifying and validating the safety of a certain design. Figure 99 illustrates what kind of role this technique plays in terms of architecture driven assurance. The user starts by modelling the system architecture with its corresponding components in Papyrus/CHES or in SAVONA. After doing so, traditional safety analysis techniques and fault injection are put together in order to perform a combined analysis of the system.

The simulation-based fault injection process starts by calling the Sabotage framework [27]. This framework helps the user to define some failures in the system model by adding in the Simulink environment some extra blocks that reproduce a failure effect on a component and visualize its effects. In contrast to xSAP [26], where the so-called error injection blocks are already included inside CHES tool, in Sabotage these extra blocks are included in the Simulink environment afterwards [12]. At present, the user needs to fill the fault list table or the fault injection policy (see Section 2.1.6) by means of a custom fault list. The fault list stands for the Fault Target, Fault Injection Trigger, Fault Duration and Fault Model [28].

Figure 99 presents the integration methodology to unify the current fault injection approach with component-based design and the inclusion of monitors in the component outputs.



**Figure 99.** Integration workflow: from contract-based design to the generation of saboteurs and monitors

The system architecture can be modelled in CHES or SAVONA; however, if it is modelled in SAVONA the user has the option to later import it into CHES tool. After modelling the system architecture, the information coming from contract specification could be used to set specific faults in the fault list. This allows completing information regarding where to inject a fault, which failure modes need to be provoked (fault model) and what faulty values should be inserted in place of the correct ones. As explained in [3], Component Fault Tree (CFT) frames are generated for each component and automatically linked according to the signal links. Based on the data types at their ports and the description of the expected (correct) behaviour by assertions from the contracts, it is even possible to automatically derive all applicable failure modes (e.g., too high, too low for continuous signals) and to specify their value and timing ranges by assertions (formal failure model). However, the inner failure propagation logic inside the CFT blocks had to be filled in manually so far. Fault injection can automatically derive or verify at early development process the behaviour of the fault propagation, which helps further formalizing the safety analysis activities.

Other values such as fault injection triggering by time and fault duration have no interconnection with the CHES/SAVONA environment, thus, they need to be set by the user in the fault list.

By implementing the previous process, the user reproduces all the possible failure modes for the inputs by violating the assumption. For instance, if an assumption states that a certain voltage value should be between Value 1 and 2, a “too low” failure mode would be reproduced by reading Value 1 and setting it to a too low stuck-at value.

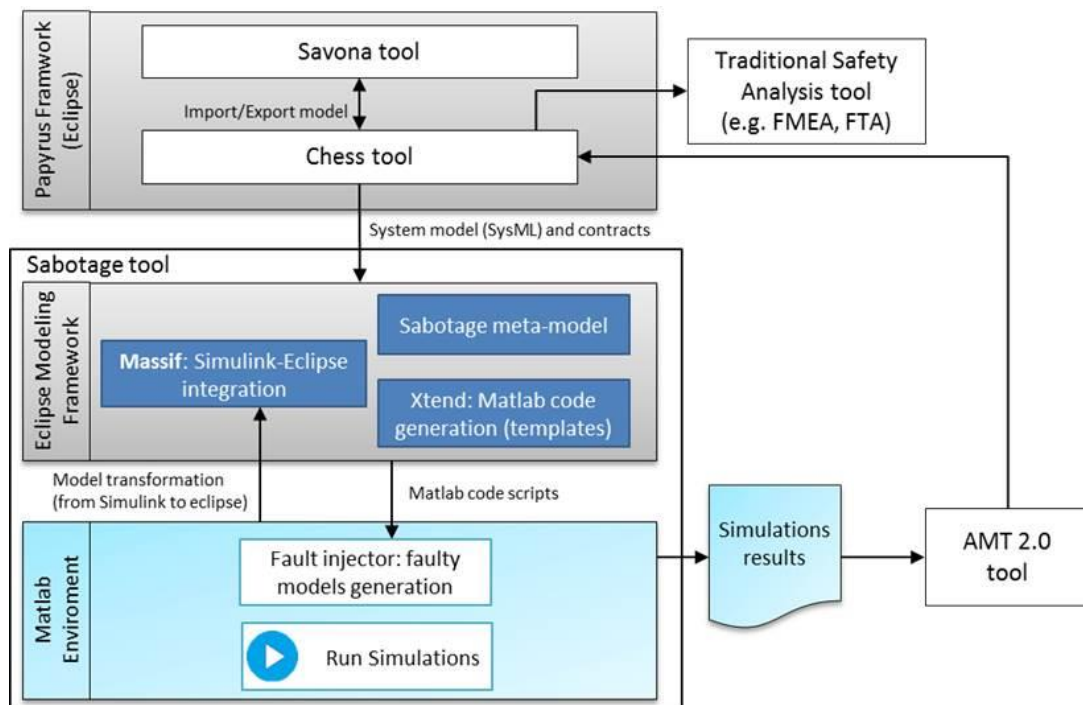
Including AMT 2.0 monitors (see Section 3.2.7.4) in the outputs of the components, it is possible to observe the behaviour of the signals related with the guarantees of system contracts. The analysis of these results is

done after the simulation of the experiments where the results are saved in a MLDATX file. AMT 2.0 tool is able to read and examine them.

After performing the first set of simulations, the results can be used as feedback by CHESSE/SAVONA in order to refine the architecture as follows: refine the guarantees and define the safety measures by safety contracts. There is no doubt that these results could be exploited directly in Simulink as well to define the safety measures and come up with the most appropriate safety concept. For example, these results can include information regarding the failure logic or fault tolerant time interval.

Once the safety mechanisms are included in the design, the user runs a second set of experiments getting different results. Thanks to the AMT 2.0 monitors the user analyses the experiments results and checks if the guarantees hold. If the guarantees hold, the required level of safety is achieved. If not, the architecture would need to be redefined as needed.

The Figure 100 illustrates a global overview of the implementation of all the steps mentioned above on the different tools.



**Figure 100.** Sabotage: global vision of the implementation with different tools

As depicted in the deliverable D3.6 [29], originally, the work was implemented within the Matlab environment. This means that both the configuration (fault list) and the fault injector scripts (matlab code generation) were directly coded in Matlab. The configuration of those experiments was performed by means of Matlab GUI.

During the second prototype, the creation of an experiments configuration view in Eclipse framework and code templates was developed. In addition, the goals of the Sabotage tool for the last prototype are the improvement of the configuration view, automatization of the experiments from Eclipse framework and the integration with other tools of AMASS platform.

D3.3 [28] highlighted how the Sabotage framework for simulation-based fault injection is constituted. As major steps, the workload generator, fault injector and monitor/controller need to be mentioned. From the user perspective, the configuration of the fault injection experiments, which is part of the workload

generator, needs to be completed. After doing so, a set of scripts are automatically called and the resulting data obtained.

### 3.3.1.1 Original solution: Simulink environment

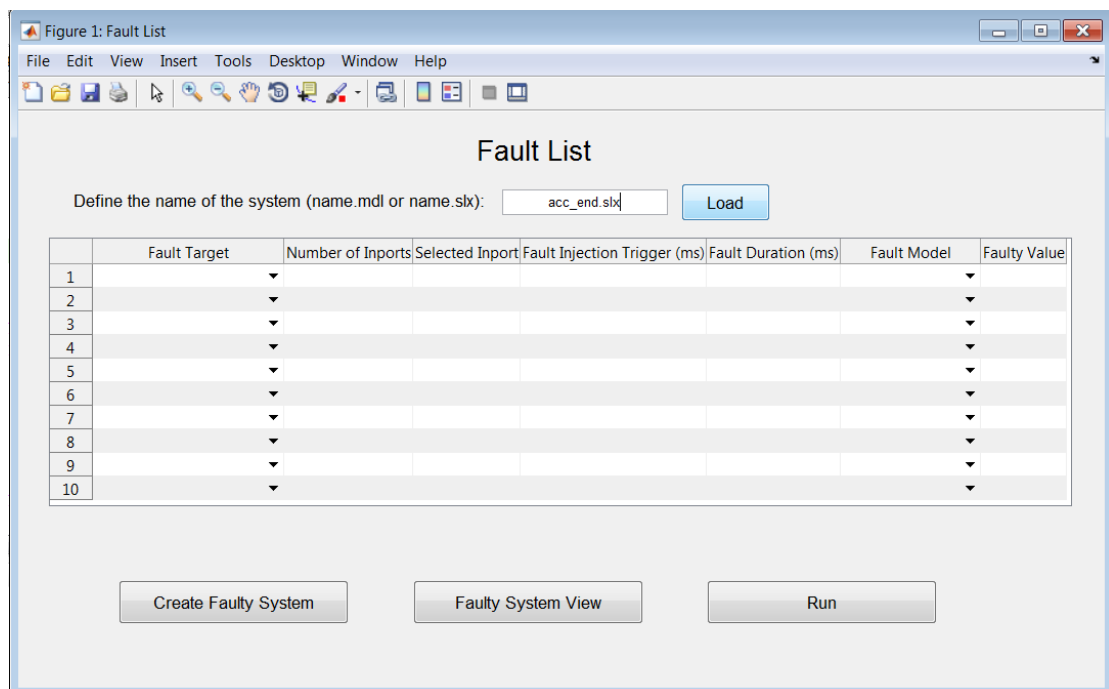
- **Configuration of the fault injection experiments and fault list creation**

In order to run the simulation-based FI experiments, the user should perform the system modelling first (see Section 4.2) in Simulink which is directly related to its corresponding SysML model.

Afterwards, the fault injection experiments need to be configured through the “Fault List” GUI. The information this list holds is necessary to configure the (extra blocks) saboteurs. These configuration values are manually included by the user.

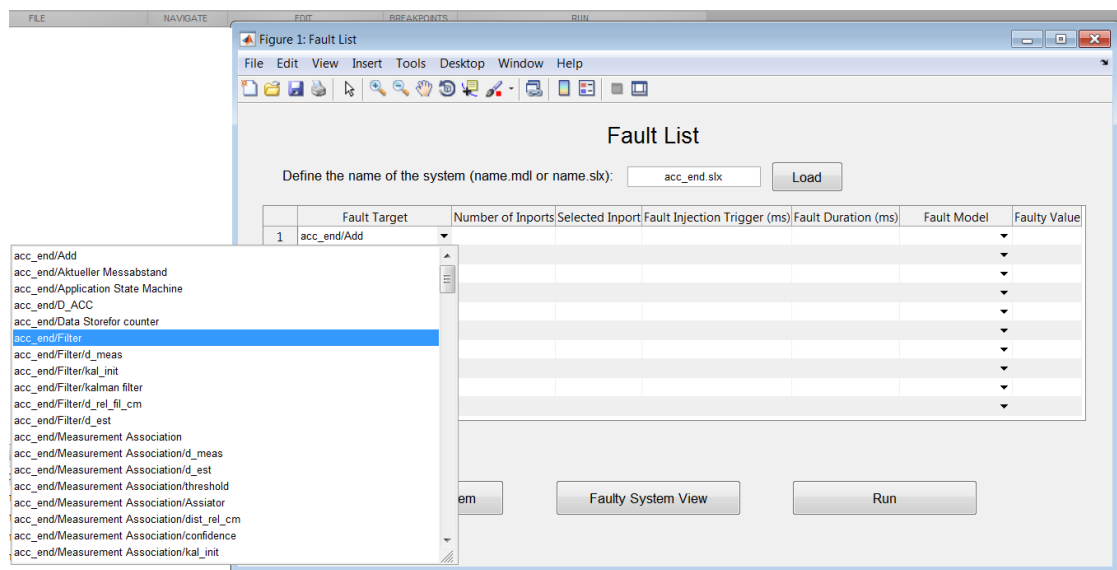
Follow the next steps in order to carry out fault injection simulations:

1. Open analysis context option for Sabotage, which will open Matlab and run the “SabotageGUI.m” script. After that, the main GUI configuration window opens (“FaultList GUI”).
2. Select the Simulink file (.mdl/.slx) that contains the system model under test. Figure 101 depicts an example for the Adaptive Cruise Control.



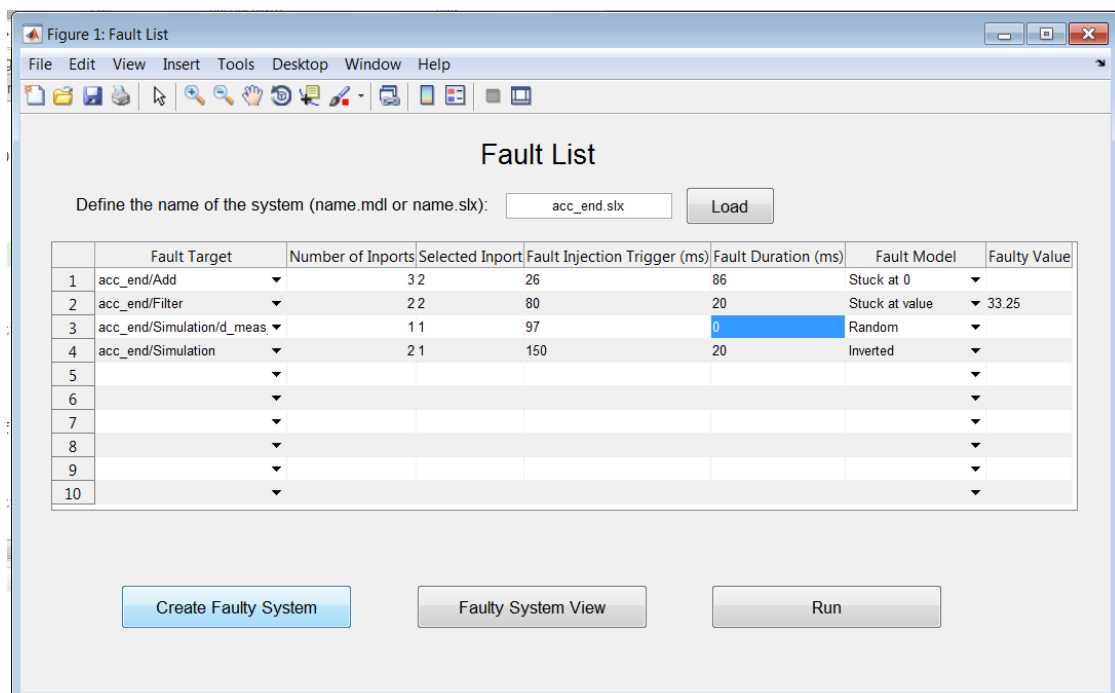
**Figure 101.** Select the Simulink file

3. Create the fault list. A fault list is completed according to the following properties:
  - a. “Fault Target Column” shows all the possible injection blocks of the system which are obtained by parsing the loaded system (See Figure 102). Select the block where to inject the saboteur.



**Figure 102.** Fault Target Block Selection

- b. "Number of Inports" depicts the number of potential input injection points. Choose one by typing the selected value in the "Selected Inport" column.
- c. "Fault Injection Trigger": include the time to trigger the injection of the fault in milliseconds (ms).
- d. "Fault Duration":
  - i. 0: represents a permanent fault
  - ii. <value>: set a duration by the user
  - iii. Random: sets a random duration value
- e. "Fault Model": stuck-at<0>, stuck-at<lastvalue>, stuck-at<value>, random, delay, noise, inverted, other.
- f. "Fault Value": optional. It is used to set a specific faulty value to a signal.

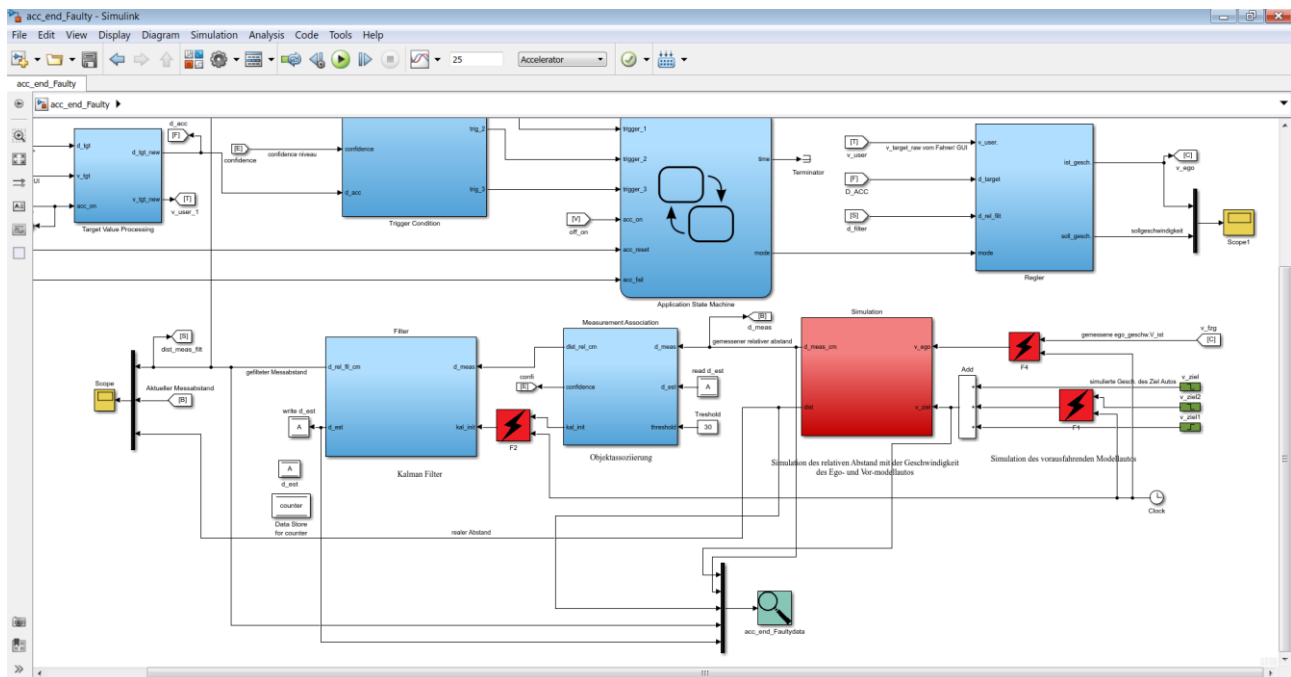


**Figure 103.** Fault List Creation

Follow this process to include the required number of faults in the same list. By doing so, the final fault list is constituted (see Figure 103).

### • Faulty System Generation (Fault Injector)

After having created the fault list, click “Create Faulty System” button. This creates the faulty system under test with its corresponding observation points included into the design. The user can click on to open the created faulty system. The file can be opened through the “Faulty system view” (see Figure 104).



**Figure 104.** Faulty System Generation

### • Fault Injector Controller

After having created the faulty system under test, click on the “Run” button. This allows running the fault free and the faulty simulations as well as the comparison between the simulation results.

#### 3.3.1.2 Eclipse solution: Integration with the AMASS Platform

As mentioned above, the user should perform the system modelling in Simulink, which is the equivalent of its corresponding system model in SysML. This equivalence allows the integration of system contracts in the configuration of the fault injection view.

### • Configuration of the fault injection experiments

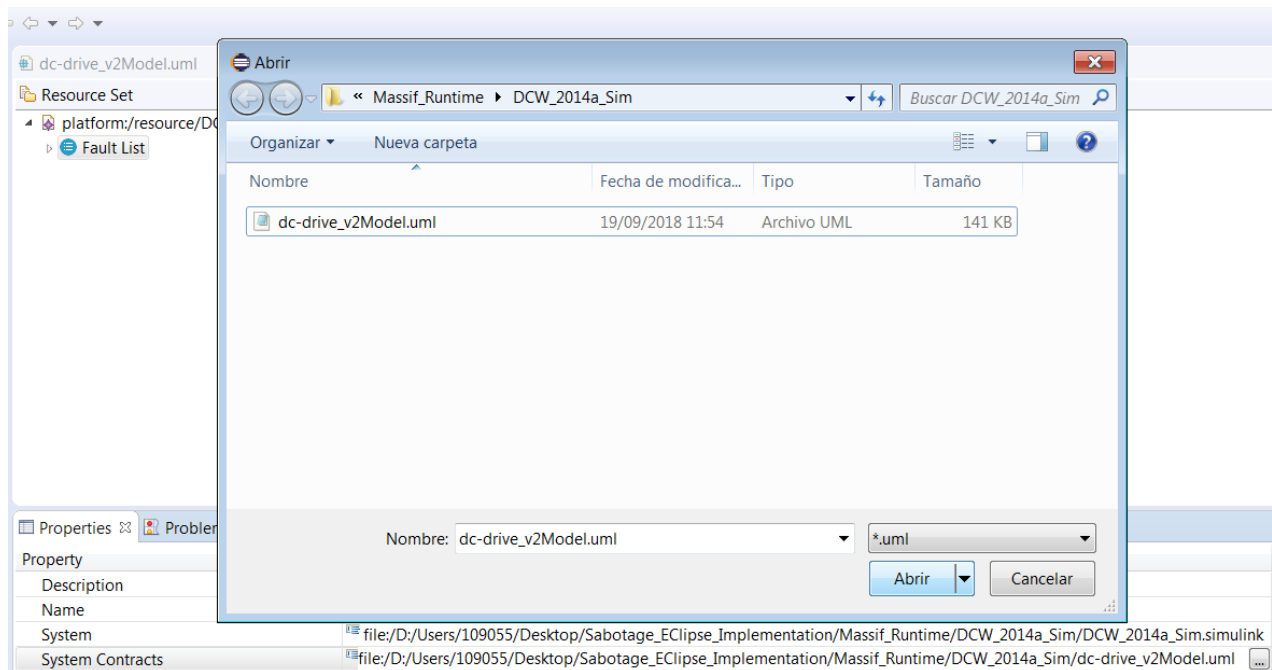
Before filling the fault list, it is necessary to transform the Simulink model to a Matlab Eclipse Modelling Framework model to obtain all the information of the system.

1. Add the .slx/.mdl file to the workspace.
2. Right click on the Simulink file and select import model. It is important to take into account that the workspace must be refreshed after the importation. A new simulink file will be created.

The fault injection experiments are configured through the fault list. To configure the fault, list the next steps should be followed:

1. Create a new Sabotage model.
2. Load the generated .simulink file on the *Fault list* property view.

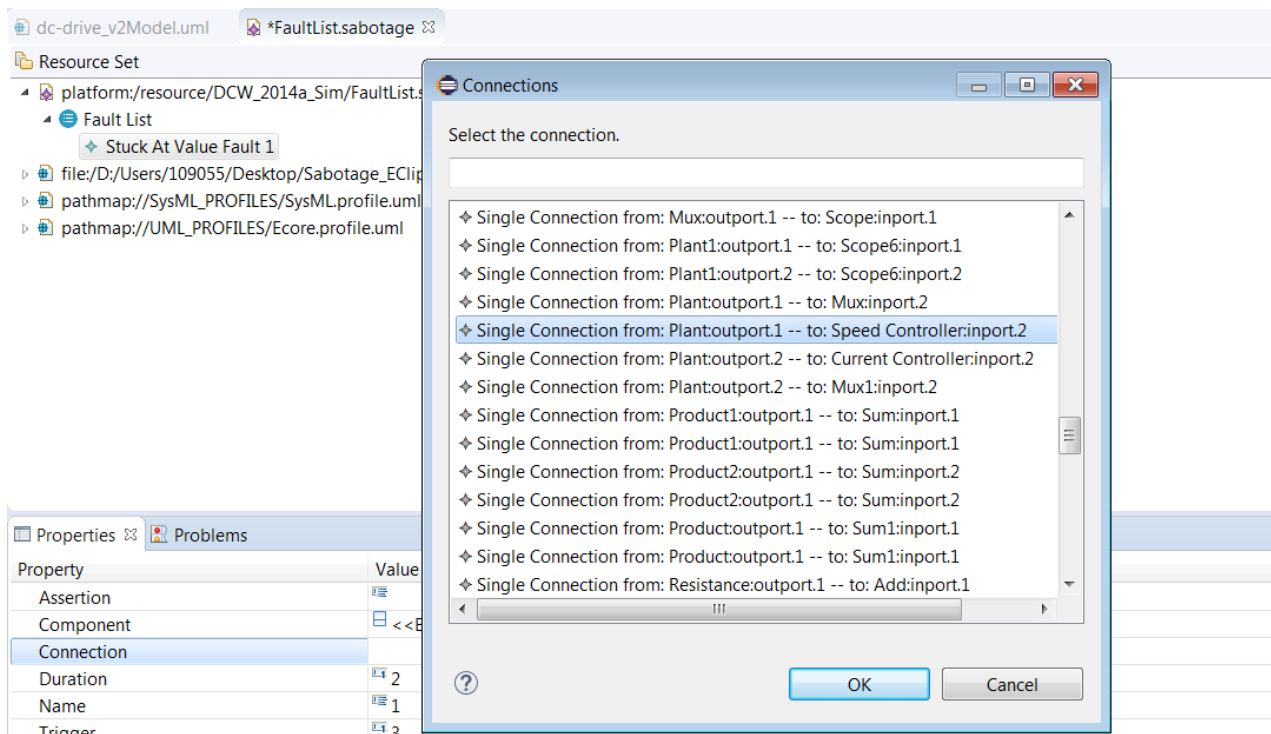
If it is wanted, the user can load the CHES file to integrate the contracts in the fault list so that the faults can be specified as the violation of its assumptions. To do so, the user should load the UML file on the *System Contracts* attribute defined in the Fault List property view as shown in Figure 105.



**Figure 105.** Loading the CHES file in Sabotage Framework

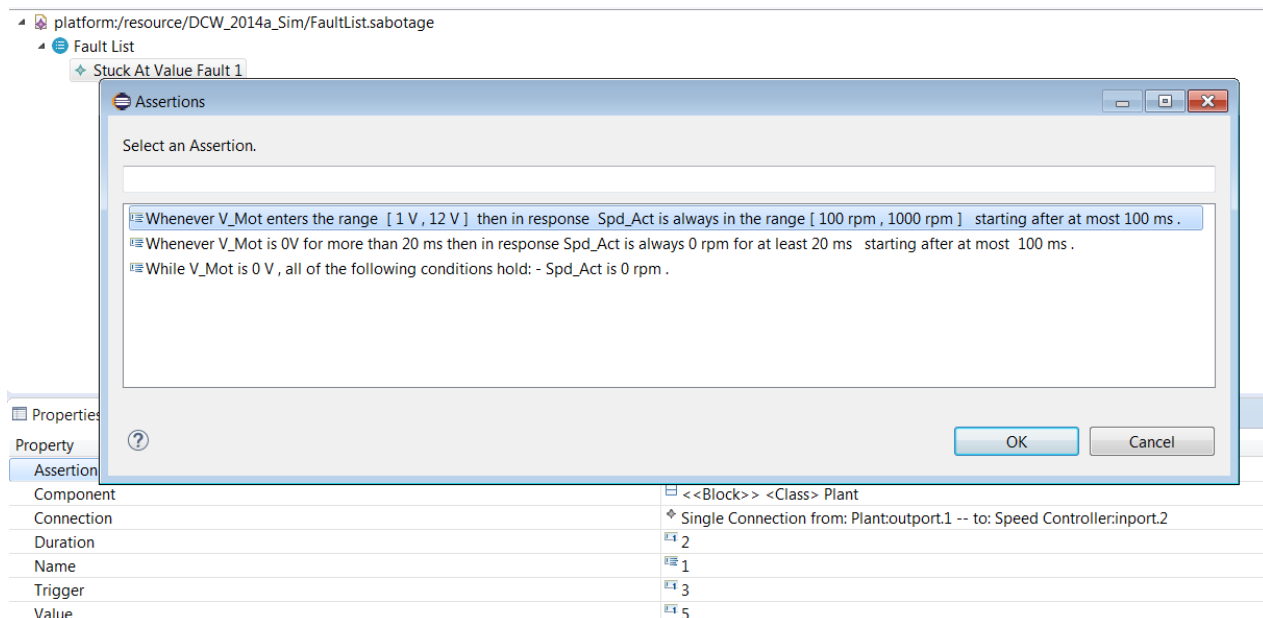
3. Complete the fault list filling the properties.
  - a. Fault model: Select stuck-at<0>, stuck-at<lastvalue>, stuck-at<value>, delay or inverted.
  - b. Connection (Fault Target): shows the possible connections of the system which are obtained by loading the *.simulink* file. Figure 106 addresses all the possible connections to the system.





**Figure 106.** The possible connections of the system model

- c. Assertion and Component: To define a violation of a certain assertion of the system the user should first select a certain component of the system and then the user will be able to proceed to the selection of one of its assertions. In Figure 107 all the assertions related with the component *Plant* are shown.



**Figure 107.** Assertions of a specific component

### • Faulty System Generation and simulations executions

The next step after filling the fault list is to right click on the Sabotage model and click on *Create Faulty system*. This step creates a Matlab code file automatically ready for creating the Golden and Faulty Simulink

model files. The last step is to right click again on Sabotage model and click on *Run Faulty System*. This action will open Matlab/Simulink tool, create the aforementioned files, run all the simulations and visualize the results.

### 3.3.2 Model-Based Safety Analysis

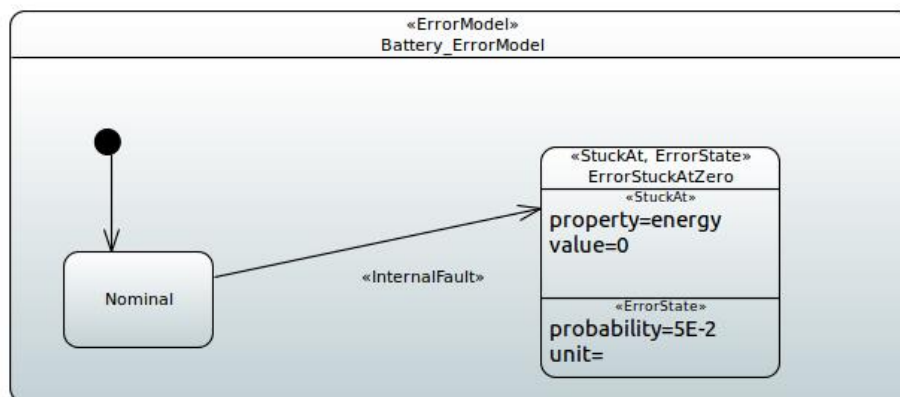
#### 3.3.2.1 Fault injection

The faulty behaviour for a component is provided through a state machine, the latter tagged with the `<<ErrorModel>>` stereotype defined in the CHESSE dependability profile<sup>13</sup>.

In the error model, the information about the error states can be provided by using the `<<ErrorState>>` stereotype. Then, for a given error state, the effect upon a property of the component, and so the effect on its nominal behaviour, can be also provided by using pre-defined effects:

- **StuckAt**: models the effect of being stuck at a fixed value.
- **StuckAtFixed**: models the effect of being stuck at a fixed random value.
- **Inverted**: models the effect of being stuck at the inverted last value.
- **RampDown**: models the effect of decreasing the value of a property by a certain value.

The state machine shown in Figure 108 represents the error model for the component depicted in Figure 89. In particular, in case of an internal fault, the component enters in an error state where the property *energy* is stuck at 0 value. The optional probability assigned to that transition is  $5 \times 10^{-2}$ .



**Figure 108.** State machine modelling faulty behaviour

#### 3.3.2.2 Fault-tree generation

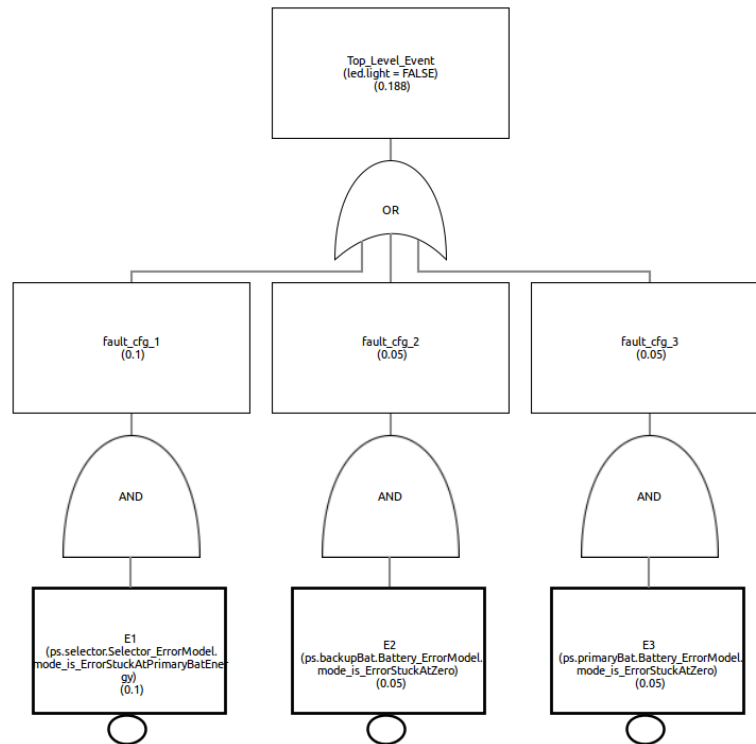
Once the system architecture has been provided, by mean of components definition and their nominal and error models, the fault tree generation can be obtained by invoking the xSAP symbolic model checker through the CHESSE environment. Please refer to the AMASS User Manual [33] for instructions on how to configure the analysis. To execute the FTA, perform the following steps:

1. Select a component (in the “Model Explorer” view) or the corresponding graphical representation (in the diagram editor). The component can be any element in the active package.
2. From the CHESSE menu select: Analysis – Dependability – FTA with xSAP.
3. From the *Select Analysis Context for FTA/FMEA analysis* window select the AnalysisContext from which the analysis has to be started: the FTA condition defined for the selected AnalysisContext appears in the window;

<sup>13</sup> CHESSE dependability profile is a part of the CHESSE profile [61]

#### 4. Select OK to run the transformation.

Once the analysis is executed, the fault tree is automatically shown in a dedicated panel in the frontend; see Figure 109 for an example of a resulting fault tree. If fault probabilities have been specified during the configuration of the error model, the fault tree will report their combination. The fault tree can be examined, in particular the minimal cut-set and so the basic fault conditions which can lead to the top-level failure.



**Figure 109.** A fault tree visualized in the CHES Editor View; note the probabilities associated to the top and basic events

#### 3.3.2.3 FMEA generation

Along with the fault-tree generation, it is possible to generate the FMEA table. FMEA analysis is configured in a similar mode as the fault-tree analysis, see the AMASS User Manual [33] for details. Steps to perform the analysis are the following:

1. Select a component (in the “Model Explorer” view) or the corresponding graphical representation (in the diagram editor). The component can be any element in the active package.
2. From the CHES menu select: Analysis – Dependability – FMEA with xSAP.
3. From the *Select Analysis Context for FTA/FMEA analysis* window select the AnalysisContext from which the analysis has to be started: the FMEA condition defined for the selected AnalysisContext appears in the window.
4. Select OK to run the transformation.

Once run, the resulting table is visualized in a specific view inside CHES, see Figure 110.

Properties ✓ Model Validation Console Hierarchical Model View V and V Results FmeaTable		
Entry ID	Failure Mode	Failure Effects
1-1	ps.selector.SelectorErrorModel.mode_is_ErrorStuckAtPrimaryBatEnergy = TRUE	led.light = FALSE
1-2	ps.selector.SelectorErrorModel.mode_is_ErrorStuckAtPrimaryBatEnergy = TRUE	led.light = TRUE
2-1	ps.backupBat.BatteryErrorModel.mode_is_ErrorStuckAtZero = TRUE	led.light = FALSE
2-2	ps.backupBat.BatteryErrorModel.mode_is_ErrorStuckAtZero = TRUE	led.light = TRUE
3-1	ps.primaryBat.BatteryErrorModel.mode_is_ErrorStuckAtZero = TRUE	led.light = FALSE
3-2	ps.primaryBat.BatteryErrorModel.mode_is_ErrorStuckAtZero = TRUE	led.light = TRUE

**Figure 110.** The FMEA table in the CHESS view

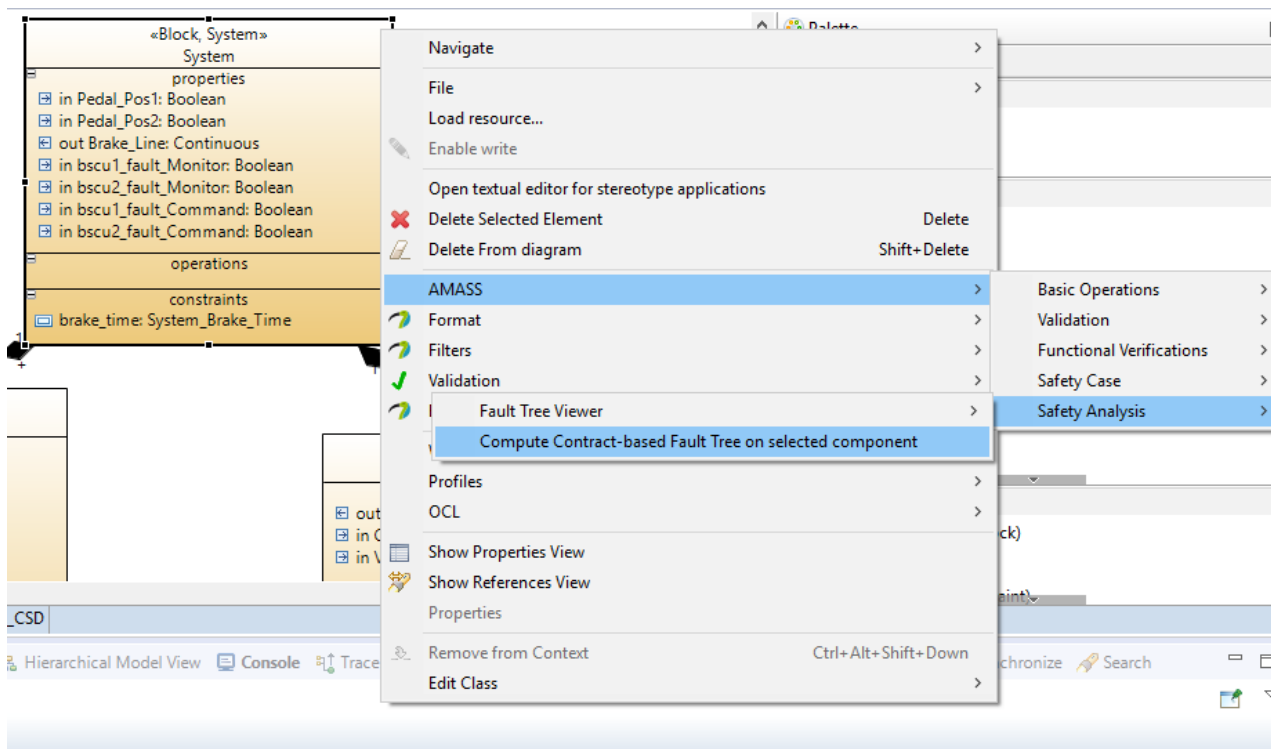
### 3.3.3 Contract-Based Safety Analysis

The contract-based safety analysis identifies the component failures as the failure of its implementation in satisfying the contract. When the component is composite, its failure can be caused by the failure of one or more subcomponents and/or the failure of the environment in satisfying the assumption.

As result, this analysis produces a fault tree in which each intermediate event represents the failure of a component or its environment and is linked to a Boolean combination of other nodes; the top-level event is the failure of the system component, while the basic events are the failures of the leaf components and the failure of the system environment (see [15] for more details).

To execute the contract-based safety analysis, perform the following steps:

1. Choose which contracts must be considered in the analysis: Select a component (in the “Model Explorer” view) or the corresponding graphical representation (in the diagram editor). The contracts considered will be the ones associated to the selected component and the ones associated to its sub components. This operation includes recursively all the contracts along the subcomponents, from the root to the leaves of the system.
2. Perform the analysis: right click on the selected component, then go to “AMASS” – “Safety Analysis” – “Compute Contract-based Fault Tree on selected component”, see Figure 111.
3. Receive the result of the analysis: when the analysis is completed, the fault tree is shown. The representation is the same as the ones used for the Model-based Safety Analysis, see Figure 109.



**Figure 111.** Dedicated menu to perform the compute the contract-based fault tree

## 3.4 Safety Case

In the following, the Safety Case activities depicted in Figure 19 are detailed, one subsection for each activity.

### 3.4.1 Evidence Generation

When a given analysis has been executed, and the analysis results have been inspected, the assurance engineer can decide to collect the artefact(s) containing the analysis results as evidence; this is typically done when the analysis results are satisfactory, i.e. the requirements under validation are satisfied. In this case the evidence editor coming with OpenCert (the base tool for management of assurance and compliance adopted in AMASS) is used to create a new evidence entity linked to the artefact(s) owning the analysis result. The architect can then trace the new produced evidence with the analysis context entity, the latter available in the CHES model, that has been used to run the given analysis; this will allow to implicitly link the context that has been analysed (e.g. system elements, specific analysis parameters values) to the obtained results. This traceability link between the “AnalysisContext” entity in the CHES model and the corresponding evidence in the evidence model (i.e. the evidence referring the artefact created by the analysis invoked through the same AnalysisContext) can be specified by using the Capra facilities (see AMASS user manual [33] about how to work with Capra in the context of the AMASS platform).

The just created evidence can then be used to support the assurance case, for instance to demonstrate that a given process step required by the process model has been executed. The evidence can also be used to support the assurance case, and can be referred directly in the assurance case editor.

If the analysis has been performed to validate a contract (and so the associated formalized requirements), the evidence has to be associated to the contract itself (see the AMASS user manual [33] about how to trace a Contract to an evidence): this information can then be used while editing the assurance case, see Section 3.4.2 about link to assurance case, or it will be used during the argument fragments generation.

Before the assurance case is available, the evidence can also be used to support a given contract or that a requirement is properly satisfied by the architecture; the traceability link between the evidence and the architecture (via the AnalysisContext) can be used to demonstrate what has been actually analysed and verified. If the executed analysis supports the validation of a given contract, the obtained evidence can also be linked to the contract itself.

Table 1 below summarizes the artefacts that can be generated by the analysis within the AMASS platform.

**Table 1.** Artefacts produced by analysis

Analysis	(Default) Location of the produced Artifact	Note
<b>Validation of contracts</b>	Under the NuSMV-OCRA/Results folder of the CHESS project (file name: selected_component_name + "_property_result.xml")	
<b>Requirement Semantic Analysis</b>	Under the NuSMV-OCRA/Results folder of the CHESS project (file name: "output_" + selected_component_name + "_property_result.xml")	
<b>Requirement Quality Metrics</b>		
<b>Contract-Based Verification of Refinement</b>	Under the NuSMV-OCRA/Results folder of the CHESS project (file name: selected_component_name + "_oss_refinement_result.xml")	
<b>Contract-Based Verification of State Machines</b>	Under the NuSMV-OCRA/Results folder of the CHESS project (file name: selected_component_name + "_oss_implementation_result.xml")	
<b>Model Checking</b>	Under the NuSMV3-XSAP/Results folder of the CHESS project (file name : output_selected_component_name + "_modelChecking_result.xml")	
<b>Contract-based Verification of Refinement for Strong and Weak Contracts</b>	Under the NuSMV-OCRA/Results folder under the CHESS project (file name xxx_result.txt and xxx_consistency_result.txt)	
<b>Fault Tree Generation (xSAP)</b>	Under the CHESS project folder.	.aird is a representation file used by the Sirius graphical tool <sup>14</sup> .
<b>Contract-Based Safety Analysis</b>	Under the NuSMV-OCRA/Results folder of the CHESS project (file name: selected_component_name + "_oss_faultTree_result.xml")	The file .xml is the result of the analysis
<b>Contract-Based Safety Analysis</b>	Under the CHESS project folder.	The file .aird is a representation file used by the Sirius graphical tool. It is used to graphically represent the

<sup>14</sup> <https://www.eclipse.org/sirius/doc/user/general/Modeling%20Project.html>

		file .xml produced by the analysis.
<b>Document Generation</b>	Chosen by the user via GUI.	

### 3.4.2 Link to Architectural Entities

One important result of the architecture-driven assurance is the possibility to have an adequate traceability between the assurance case and the architectural entities, e.g. to allow assurance architect and assurance manager/assessor to easily navigate all the available information.

The approach of contract-based design proposed in AMASS plays an important role on this aspect; e.g. the assurance case has to elaborate on each safety contract validity and upon the composition of contracts (derived from the composition of the associated components). So, traceability links can be provided between assurance case entities elaborating upon a contract and the contract itself.

According to the envisaged approach (see D3.3 [28] Section 3.2.2.5 and 3.2.2.6), traceability links between the following entities have been identified as useful:

- Contract and Assurance Case Package, the latter owning the assurance case entities related to the contract.
- Contract and Assurance Case Agreement, the latter owning the arguments about how the assumptions of a contract are fulfilled in the context of the system.
- Contract and Assurance Case Claim, the latter elaborating on the contract itself, e.g. that the contract is derived from some analysis or is based on some specification.
- System component and Argumentation element, the latter owning all the assurance case fragment information related to the associated component.
- Contract and Evidence, the latter supporting the contract statement, in particular its guarantee, e.g. by using some verification/test result

Evidences associated to a contract can then be reused and referred from the assurance case.

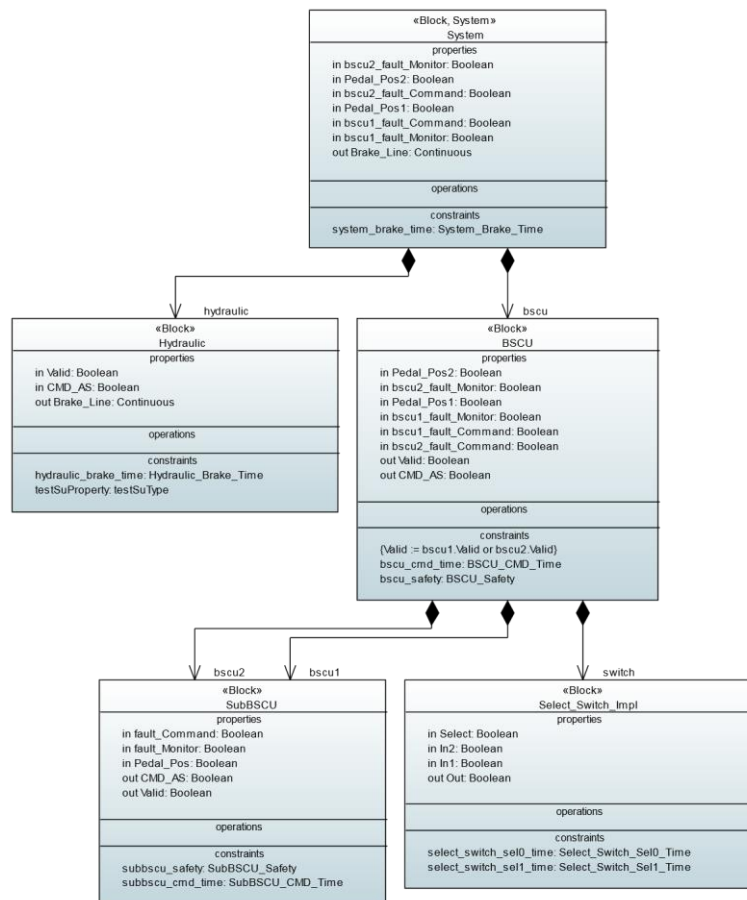
Some of the aforementioned links are automatically managed in case of argument fragment generation; for the parts of the assurance case not currently supported by such automatism, the links have to be created manually by using the Capra tool support (see the AMASS User manual [33]).

In addition to the aforementioned links between the architecture and assurance case related elements, traceability links should also be provided between the system model and the executed process, the latter also modelled in OpenCert e.g. as an evidence model listing all the artefacts to be produced according to a given standard/process step; these relationships can then be reused to support the demonstration of the compliance of the architecture with respect to a given process. Again, the links to the executed process can be manually managed by using the Capra tool support.

### 3.4.3 Document Generation

As part of the evidences to be used at support of the safety case, the tool can generate a document summarizing the modelling of the system components and the verification, validation, and analysis results.

The generated document is composed by a list of sections. The first section includes the diagrams that are not associated to a specific component, such as the block definition diagrams, an instance is shown in Figure 112.

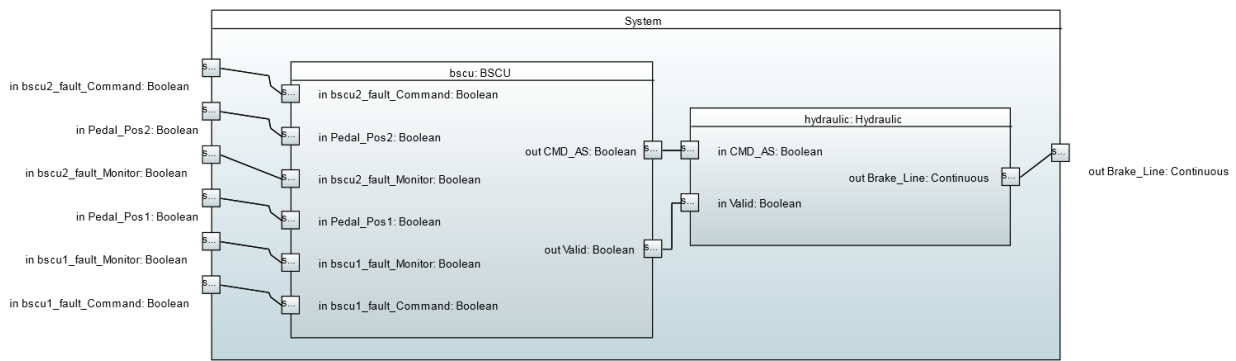


**Figure 112.** Block Definition Diagram exported as vector image

The other sections describe in detail every component. Each section contains:

- Name of the component
- Table of input ports with name and type
- Table of output ports with name and type
- Table of subcomponents with instance name and type name
- Table of interface assertions
- Table of refinement assertions
- Table of connectors between ports
- Table of contracts
- Table of contract refinements
- Table of uninterpreted functions
- Table of parameters
- Table of parameters assumptions
- Zero, one or more internal block diagrams, see Figure 113.

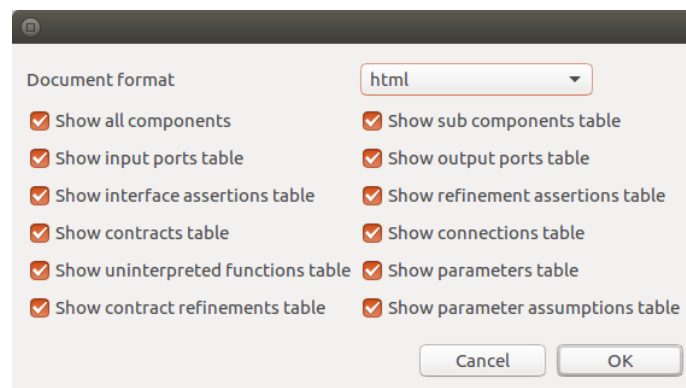




**Figure 113.** Instance of an Internal Block Diagram exported as vector image

To automatically generate the documentation, perform the following steps:

1. Select the root component: Select a component (in the “Model Explorer” view) or the corresponding graphical representation (in the diagram editor). The information used to generate the documentation will be related to the selected component and to its sub components. This operation includes recursively the information from the root to the leaves of the selected component;
2. Perform the document generation: right click on the selected component, then go to “CHESS” – “Safety Case” – “Document Generation” – “Generate documentation from selected component”.
  - a. A popup appears to set the options related to the format of the document and to the style of the diagrams, see Figure 114. For this prototype the supported formats are HTML and Latex.
  - b. A popup appears to select the folder that will contain the document and the diagrams.



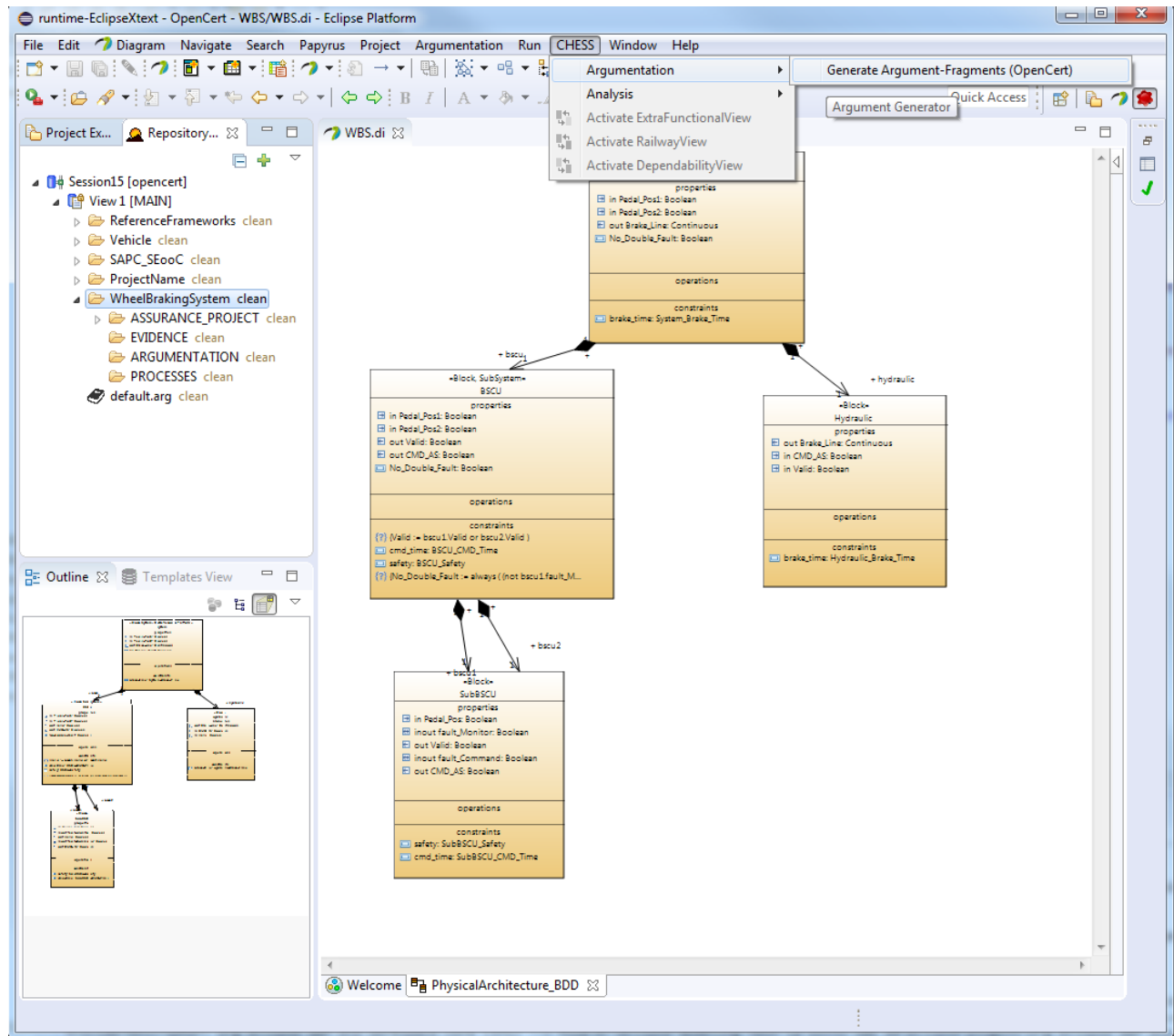
**Figure 114.** Popup to set the preferences of the generated document and diagrams

### 3.4.4 Argument Fragments Generation

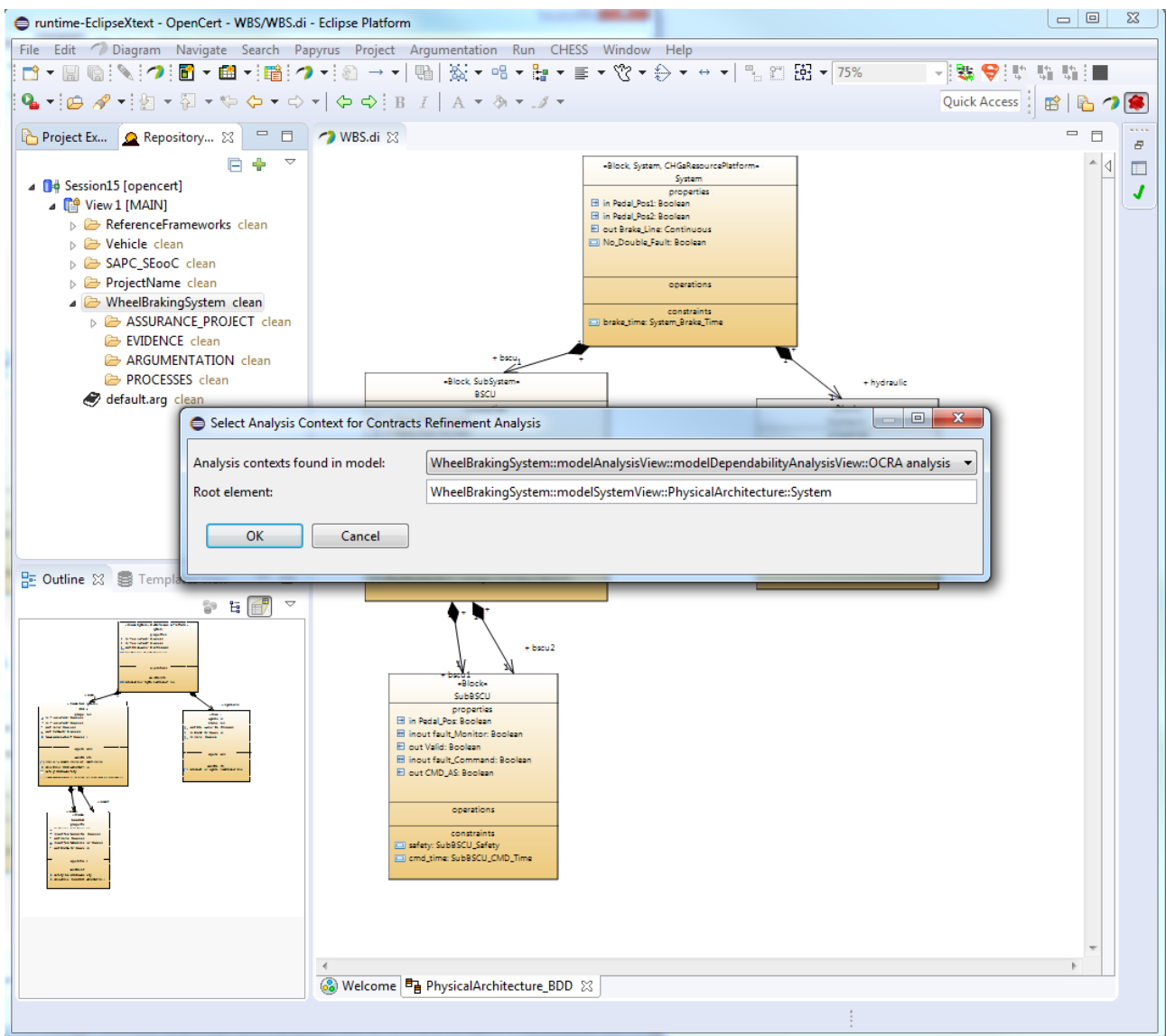
As mentioned in Section 2.1.7, safety argument is the backbone of a safety assurance case, which aims at assuring that the system is acceptably safe to operate in a given context. Argument-fragments represent parts of that safety argument focused on assuring a specific aspect of the system, e.g., a particular component or requirement [67]. The Argument Generator plugin is implemented in CHESS. It generates a set of argument-fragments from the selected CHESS model and stores them in the corresponding destination assurance case in the Connected Data Objects (CDO) repository stated in the OpenCert preferences. Note that the whole safety case is not generated automatically and the generated fragments should be reviewed and integrated with the rest of the assurance case.

The Argument Generator assumes that the CHESS model is enriched with contracts and that contract refinement has been performed such that contract status is updated to indicate if the contract is validated in the given context or not [83]. Moreover, the Argument Generator assumes that the analysed model and the

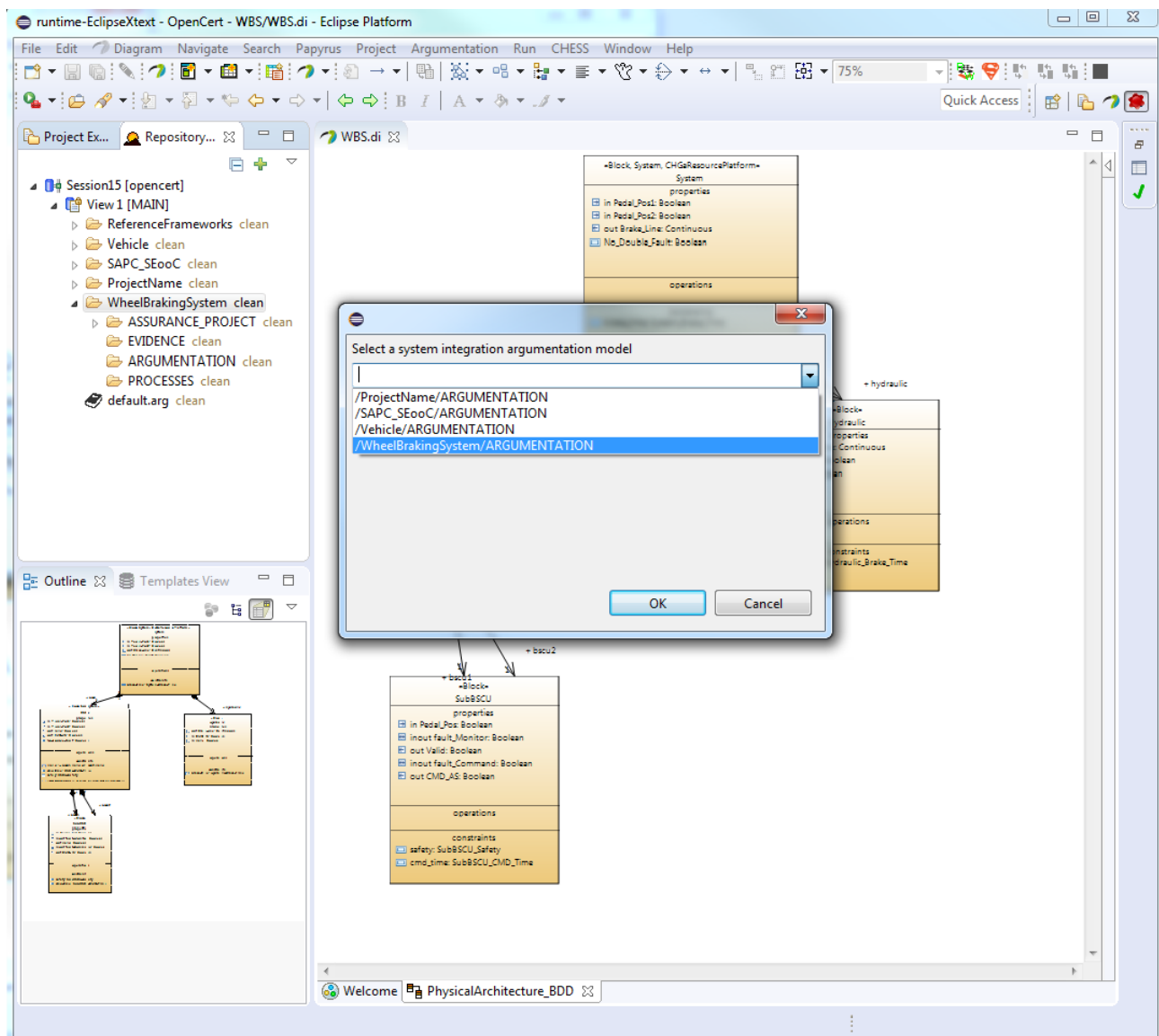
refinement check results are stored in the refinement analysis context. The attached screenshots (Figure 115 - Figure 119) illustrate the usage of the Argument Generator plugin. First, the plugin prompts selection of the OCRA refinement analysis context (Figure 115 - Figure 116). Then, the user needs to indicate to which assurance case on the corresponding CDO server the argument-fragments should be generated (Figure 117). The argument-generation is performed for each component and for each validated contract (Figure 118). The set of argument-fragments for each component can be viewed in the selected assurance case (Figure 119).



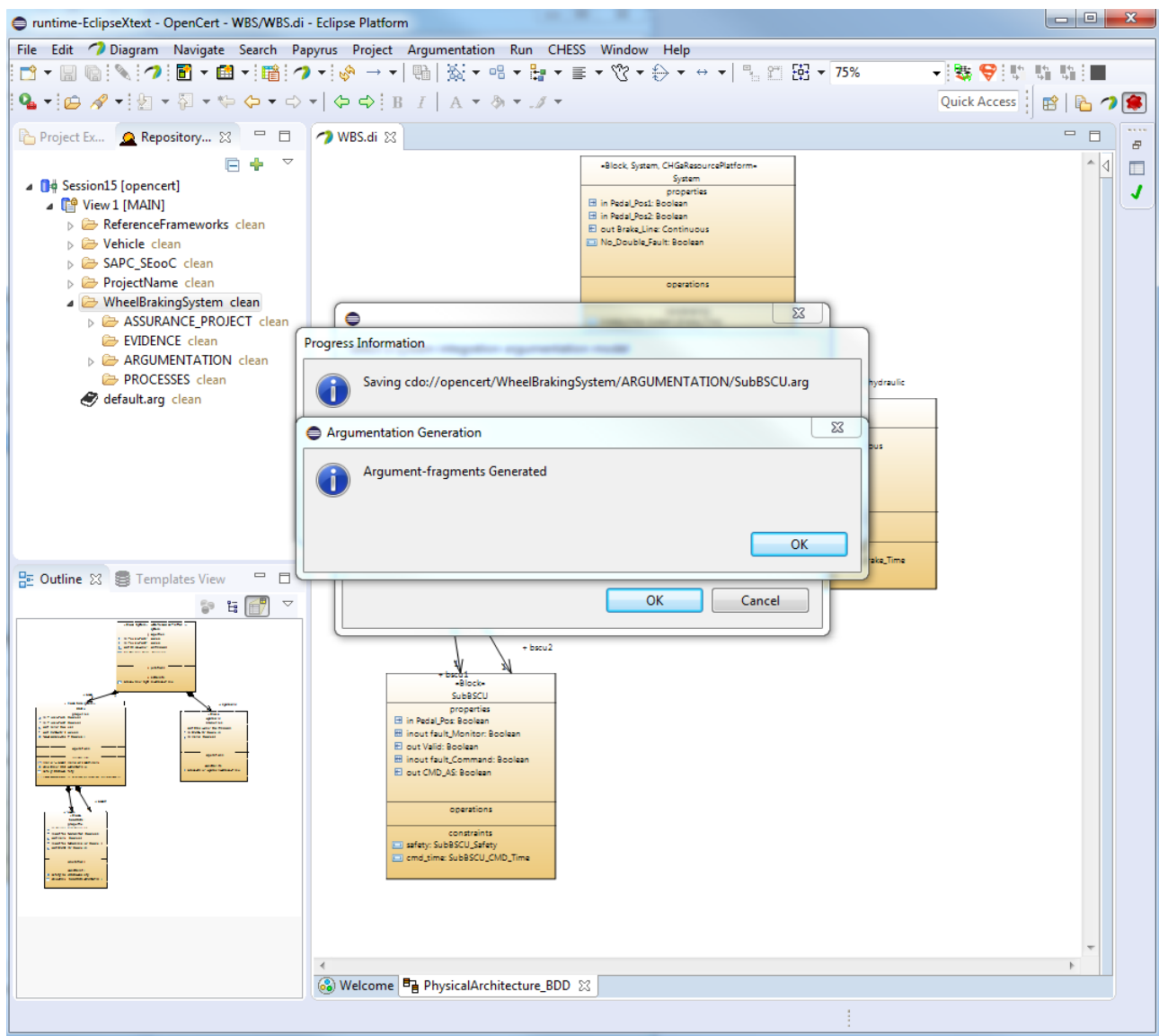
**Figure 115.** Initiating the argument-fragment generation



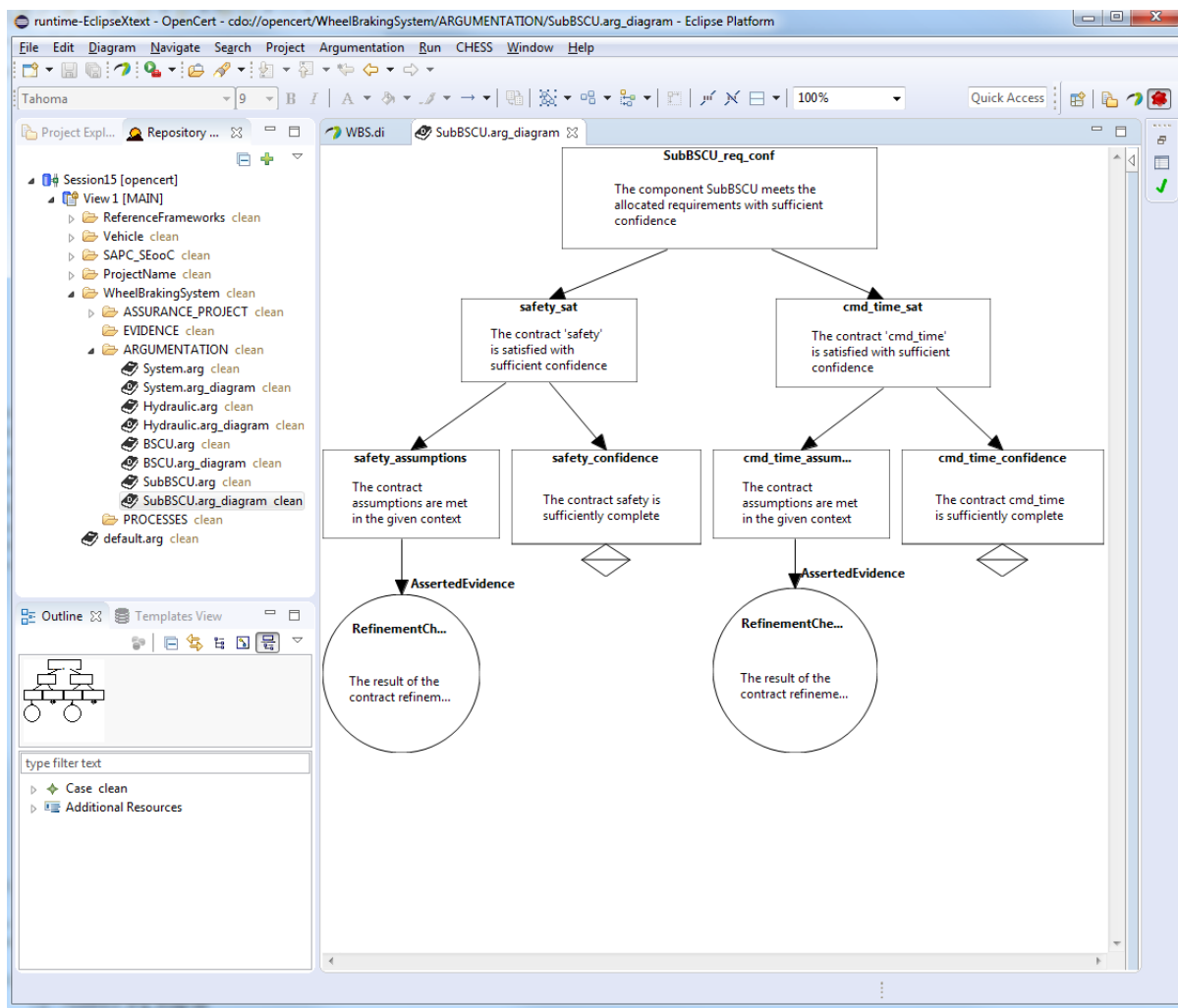
**Figure 116.** Selecting the source analysis context



**Figure 117.** Selecting the destination assurance case folder on the CDO repository



**Figure 118.** Generation successfully completed with argument-fragments for each block



**Figure 119.** An example of the generated argument-fragment

## 3.5 Summary

The following Table 2 summarizes the modules supporting the different activities:

**Table 2.** Summary of modules supporting the different phases of the AMASS architecture-driven assurance

PHASE	SUPPORTED ACTIVITIES
1. System definition	Using SysML BDD, IBD and Requirements Diagrams, possibility to import requirements from external tools, formalization of requirements supported with syntax-driven editors.
2. Requirements early validation	Formal semantic analysis of formal properties, analysis of requirements quality with machine learning techniques and requirements quality metrics.
3. Functional refinement	Using SysML IBD extended with contracts.
4. Component's nominal and faulty behaviour definition	Using State Machines or Simulink models extended with fault injection.
5. Functional early verification	Contract-based Verification of refinement and state-machines, Model Checking, Contract-based Monitoring of Simulink models, Contract-based Verification of Refinement with Strong and Weak Contracts.
6. Safety Analysis	Simulation-based Fault Injection, Model-Based Safety Analysis, Contract-



	Based Safety Analysis.
7. Safety Case	Evidence Generation, Link to Architectural Entities, Document Generation, Argument Fragments Generation.

## 4. Case Studies

In this section, three case studies applying the methodological guide are described. These case studies are not meant to cover completely the methodology, but they have been chosen based on the past experience of the partners involved in the development of the Architecture-Driven Assurance approach.

### 4.1 AIR6110

#### 4.1.1 Case Study Description

##### 4.1.1.1 AIR6110 and the FBK-Boeing Case Study

AIR6110 [18] is an informational document issued by the SAE that provides an example of the application of the ARP4754A<sup>15</sup> and ARP4761<sup>16</sup> processes to a specific aircraft function and implementing systems. The non-proprietary example of a Wheel Brake System (WBS) in this AIR, demonstrates the applicability of design and safety techniques to a specific architecture. In this example, The WBS comprises a complex hydraulic plant managing two landing gears each with four wheels and controlled by a redundant computer system with different operation modes. The WBS provides symmetrical and asymmetrical braking and anti-skid capabilities.

The joint case study between FBK and Boeing, presented in [19], enhances the AIR6110 with formal methods. First, WBS architectures in AIR6110 formerly using informal steps are recreated in a formal manner. Second, methods to automatically analyse and compare the behaviours of various architectures with additional, complementary information not included in the AIR6110 are presented. Third, an assessment of distinct formal methods ranging from contract-based design, including model checking, to model-based safety analysis is provided.

The models are imported into CHESS and perform the early V&V and safety analysis supported by the AMASS platform to trace the results and link them to the safety case. The AIR6110 document details the development of the WBS system architecture in four versions, each obtained after design choices of different types. The AMASS platform will enable to trace and compare the different architectures.

The git repository<sup>17</sup> (branch master) contains the 5 CHESS projects: AIR6110, AIR6110Arch2, AIR6110Arch3, AIR6110Arch4, AIR6110Arch5. They respectively describe the modelling of the first version of the WBS architecture Arch1, Arch2, Arch3, Arch4, Arch5, all described in [18] and formalized in [19]. Currently, in reference to the Table 2, this case study covers the following phases:

- System definition: creation of the System element of the model using the BDD, see Section 3.2.1.2.
- Requirement Formalization: creation of formal properties and contracts, see Section 3.2.3.1.
- Requirements Early Validation: Requirement Semantic Analysis, see Section 3.2.4.1 and Validation of Contracts, see Section 3.2.4.2.
- Functional refinement: definition of the sub-components and their interactions using IBDs, see Section 3.2.5.1.1. Refinement of contracts, see Section 3.2.5.2.
- Functional early verification: execution of the Check Validation Property and the Verification of Contract Refinement, see Section 3.2.7.1
- Safety Analysis: Execution of the Contract-based Fault-Tree Generation, see Section 3.3.3.

---

<sup>15</sup> <http://standards.sae.org/arp4754a/>

<sup>16</sup> <http://standards.sae.org/arp4761/>

<sup>17</sup> [https://gitlab.fbk.eu/CPS\\_Design/CHESS\\_SystemArchitectureProjects](https://gitlab.fbk.eu/CPS_Design/CHESS_SystemArchitectureProjects)



- Safety Case: Automatic generation of documentation, see Section 3.4.3.

#### 4.1.1.2 Overview of the Aircraft

The WBS is part of a hypothetical aircraft, designated model S18. The S18 is a passenger aircraft, with a capacity of 300-350 passengers, capable of a flight duration of approximately five hours. The S18 comprises two engines, two main landing gears and a nose landing gear. Each main landing gear contains four wheels.

The S18 aircraft systems manage nine basic functions:

- provide structural integrity;
- provide stability and control;
- provide control of energy;
- provide operational awareness;
- provide a controlled environment;
- provide power generation and distribution;
- provide loading, maintenance, ground handling and occupant accommodation;
- provide control on the ground;
- provide control in the air.

#### 4.1.1.3 Overview of the WBS

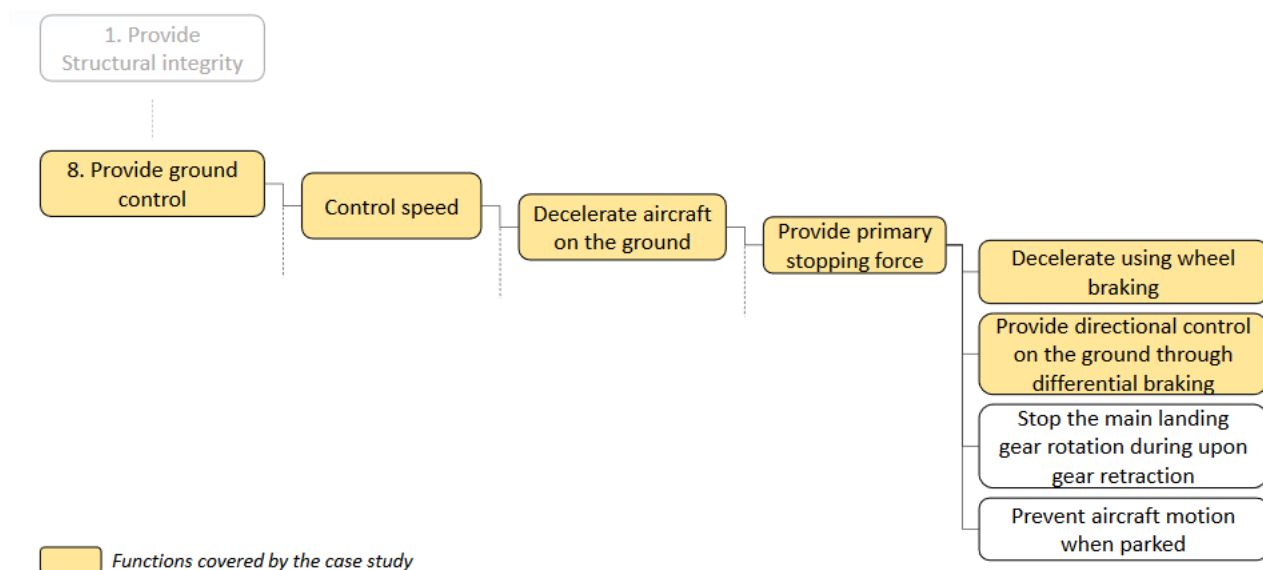
##### 4.1.1.3.1 Functions

The WBS manages part of the “provide control on the ground” aspect of basic aircraft function. In particular, it fully implements the subfunction “provide primary stopping force,” which decelerates the aircraft on the ground. The WBS must ensure the behaviour of four leaf functions:

- decelerate using wheel braking;
- provide directional control on the ground through differential braking;
- stop main landing gear wheel rotation upon gear retraction;
- prevent aircraft motion when parked.

An overview of the functions covered by the WBS is given in Figure 120, based on the understanding of AIR6110.

The case study used in this document takes into account two leaf functions: “Decelerate using wheel braking” and “Provide directional control on the ground through differential braking”.



**Figure 120.** Functional decomposition of the AIR6110 case study

#### **4.1.1.3.2 Structure**

The WBS manages the brakes on the eight wheels of the two main landing gears. The nose gear has no brakes. The WBS receives hydraulic and electric power, and displays information to the crew (e.g. the validity of the brake signal produced by the Control System). To enable these features, it interfaces with systems associated to other functions:

- the hydraulic and electric power plants, which cover the “provide power generation and distribution” function;
- the display system, which covers the “provide operational awareness” function.

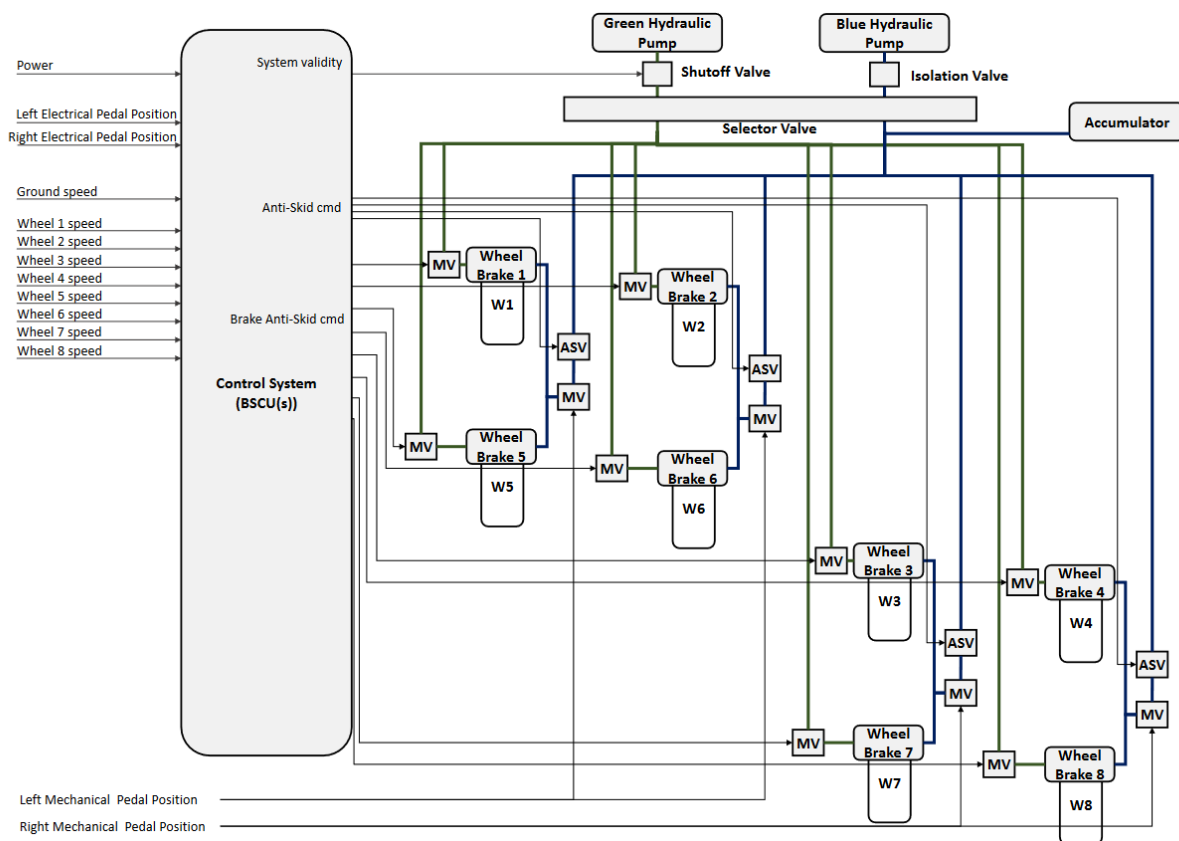
An overview of the WBS is given in Figure 121.

#### **4.1.1.4 WBS Architecture evolution**

The development of the WBS in the AIR6110 document is described through four evolutionary steps, each step resulting in a specific architecture:

- Arch1: The highest view of the architecture of the WBS comprises one BSCU and one hydraulic circuit backed by an accumulator. This is the first step in the architecture decomposition by defining the main functional elements of system.
- Arch2: This is the basic architecture of the WBS. It includes redundancy principles: There are two BSCUs, a green circuit and a blue circuit. At this step multiple braking modes are introduced.
- Arch3: This evolution of Arch2 replaces the two BSCUs of the control system by one dual channel BSCU.
- Arch4: This evolution of Arch3 modifies the placement of the accumulator and adds a link from the control system validity to the selector valve in the physical system.

Evolution from one architecture to the next occurs as a result of assessment activities during the development process.



**Figure 121.** WBS architecture overview (MV=Meter Valve; ASV=AntiskidShutoff Valve; W=Wheel)

## 4.2 ETCS Linking function

The present case study is intended to provide the final user with an example related to railway domain of the use of the AMASS platform for Architecture-Driven Assurance. The case study is mainly focused on the following activities:

- System Definition (see Section 3.2.1 of the present deliverable)
- Requirements Formalization (see Section 3.2.2 of the present deliverable)
- Functional Refinement (see Section 3.2.4 of the present deliverable)
- Functional Early Validation (see Section 3.2.6 of the present deliverable)

Furthermore, an example of Evidence Generation, as described in Section 3.4.1 of the present deliverable, is provided, even if Safety Case activities are not directly in the scope of the present case study.

The git repository<sup>18</sup> (branch master) contains the CHESS project ETCS\_linking\_02\_ParamArch, which is up to date version of the model of the linking function.

### 4.2.1 Case Study Description

The case study focused on the ERTMS/ETCS sub-system aims to model the Linking function through the AMASS platform.

<sup>18</sup> [https://gitlab.fbk.eu/CPS\\_Design/CHESS\\_SystemArchitectureProjects](https://gitlab.fbk.eu/CPS_Design/CHESS_SystemArchitectureProjects)

The European Rail Traffic Management System (ERTMS) is the system of standards for management and interoperation of signalling for railways within the European Union.

The main target of ERTMS is to promote the interoperability of trains in the European Union. It aims to greatly enhance safety, increase efficiency of train transports and enhance cross-border interoperability of rail transport in Europe. This is done by replacing former national signalling equipment and operational procedures with a single new Europe-wide standard for train control and command systems.

ETCS, acronym for European Train Control System, enables information to be transmitted to the driver related to line status, permitted speed and to continuously check that this information is being complied with. Together with the GSM-R, radio system based on the GSM mobile phone standard and used in the railway field to exchange information between the trackside equipment and the train, it forms the so-called ERTMS.

The ETCS system can be developed at four application levels. For the purpose of this model, the architecture will be focused on Level 1, where data transmission to the train is mainly performed discontinuously via EUROBALISE.

EUROBALISE is the European system for the transmission of information relevant to safety between the train and the trackside equipment. It's a function of ERTMS and one of the ETCS sub-systems.

The EUROBALISE system consists of:

- Balise: beacons, fixed or controlled type, located along the railway track, which are energized and enabled to transmit only when the train antenna is above them.
- On-board transmission system: consisting of the antenna unit and BTM (Balise Module Transmission) function.
- Trackside signalling system: consisting of the lineside electronic units and other external equipment involved in the signalling process constantly communicating with controlled Balise.

The requirements for the train ETCS on board subsystem and trackside are defined in the System Requirements Specification UNISIG Subset-026, [20] and, in particular the linking function, the model it refers to, is defined in section 3.4.4 of [20].

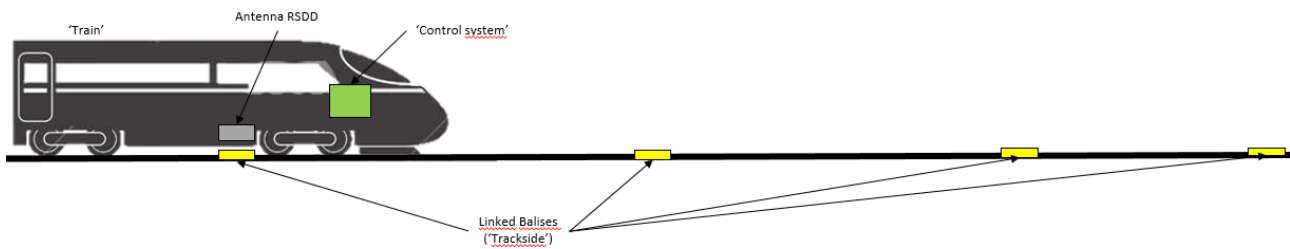
A balise is linked when its linking information containing location, location accuracy and direction is known in advance.

In general, the aim of linking can be summarized in the following targets:

- To determine whether a balise group has been missed or not found within the expectation window and take the appropriate action.
- To assign a co-ordinate system to balise groups consisting of a single balise.
- To correct the confidence interval due to odometer inaccuracy.

On the basis of this statement, a model of a simple railway system constituted by trackside and on-board components has been defined. In parallel, all the requirements related to linking function defined in UNISIG Subset-026 [20] as set of natural language has been imported in a UML model.

The Figure 122 depicts the architecture developed with focus on linking function implementation.



**Figure 122.** Pictorial view of the Linked Balises on the track

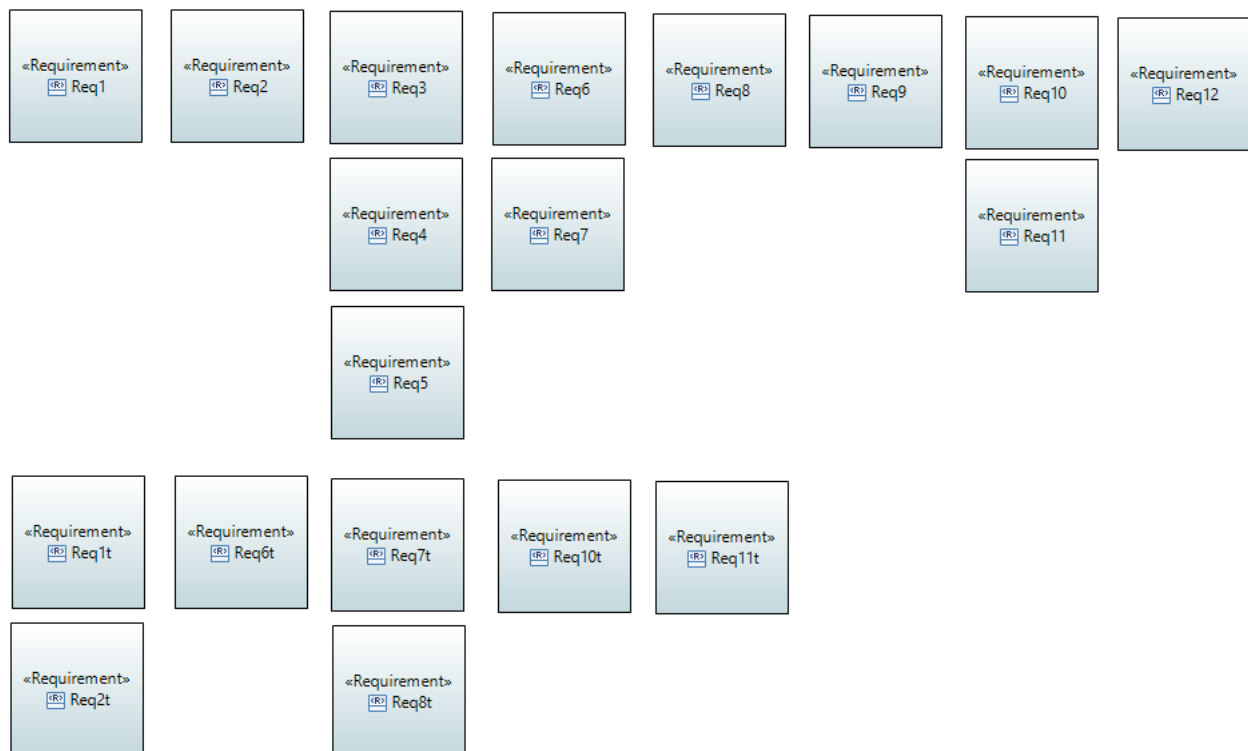
## 4.2.2 System Definition

The first phase of the modelling activity consisted in the specification of the requirements related to linking function as defined in UNISIG Subset-026 [20]. The requirements were represented in the model by using the SysML standard support, creating the Requirements Diagram illustrated in Figure 123. In particular, two subsets of requirements were graphically created: one related to train ETCS on board subsystem, which means requirements that the ETCS on board system has to meet and the other to trackside, which means requirements that the signalling trackside infrastructure has to meet.

The requirements were imported manually from the standard and the following fields of the SysML tab were filled:

- “Id” – the number of the requirement as defined in Subset-026 [20]
- “Name” – the name “Req.” followed by a progressive number
- “Text” – the requirement in natural language as defined in Subset-026 [20]

The requirements applicable to the model were then linked to the related formal properties, as described in the following sections.



**Figure 123.** ETCS Linking model - Requirements Diagram

#### 4.2.2.1 Basic assumptions

In the present section the basic assumptions and main constraints related to the creation of the model are reported. The model aims at the application of AMASS platform to model only partially a real railway scenario and consequently has some limitations and simplifications that must be taken into account by the end user.

The main assumption are the followings:

- The movement of the train is considered only in one direction (forward).
- The Balise Group are constituted by only one Balise.
- The Linking information cannot be iterated: only the next expected Balise is announced by the previous one.

#### 4.2.2.2 Architecture

In order to model the linking function, a simplified railway system has been implemented in the AMASS platform. The architecture of the system was designed by means of SysML Block Definition Diagram in which the system components and their relationship were produced.

The basic concept of the model consists in defining the core block, called “Linking\_System”, which is constituted by the trackside component block, called “Trackside”, and on-board component block called “Train”. These blocks can be considered as the logic functions related to linking with the apportionment of the linking requirements at on board and trackside level. The “Trackside” constitutes the signalling infrastructure of the railway line, which, for the purpose of this model, is only composed of the EUROBALISE system. The “Train” constitutes the group of real equipment normally installed on board, which receives input from external sources and acts on the train dynamic behaviour.

On the other hand, a physical part of the system has been defined by means of the block called “Physical” which is divided, as done for the block “Linking\_System”, in trackside and train components. This part of the system represents the physical behaviour of a real system: the block “Physical\_Trackside” contains the balises located along the route, while the block “Physical\_Train” contains the train location information during its journey along the route.

The blocks “Braking\_System” and “Driver” represent the third part of the model. The definition of these components is based on the consideration that the output of the elaboration performed by the on-board subsystem can be thought mainly oriented to the driver, by means of the display of icons and text in the DMI (Driver Machine Interface) located in the cabin in front of the driver, and to the train brake sub-system, by means of the command of emergency and/or service brakes.

A modelization of the braking sub-system and of the driver machine interface could be very complex and are considered out of the scope of the present model. As reported in next sections, the only interface implemented in the model consists in the reception of the brake triggers for the braking system and the reception of the icon and text to be displayed to the driver.

Figure 124 illustrates the complete SysML Block Definition Diagram of the model.

For each component, including the System component, designed in the SysML Block Definition Diagram a SysML Internal Block Diagram was created in order to model components input/output ports, its decomposition into parts and how the parts are connected.

In the following section, a description of each part of the model is reported together with a description of the main assumptions at the basis of the model.

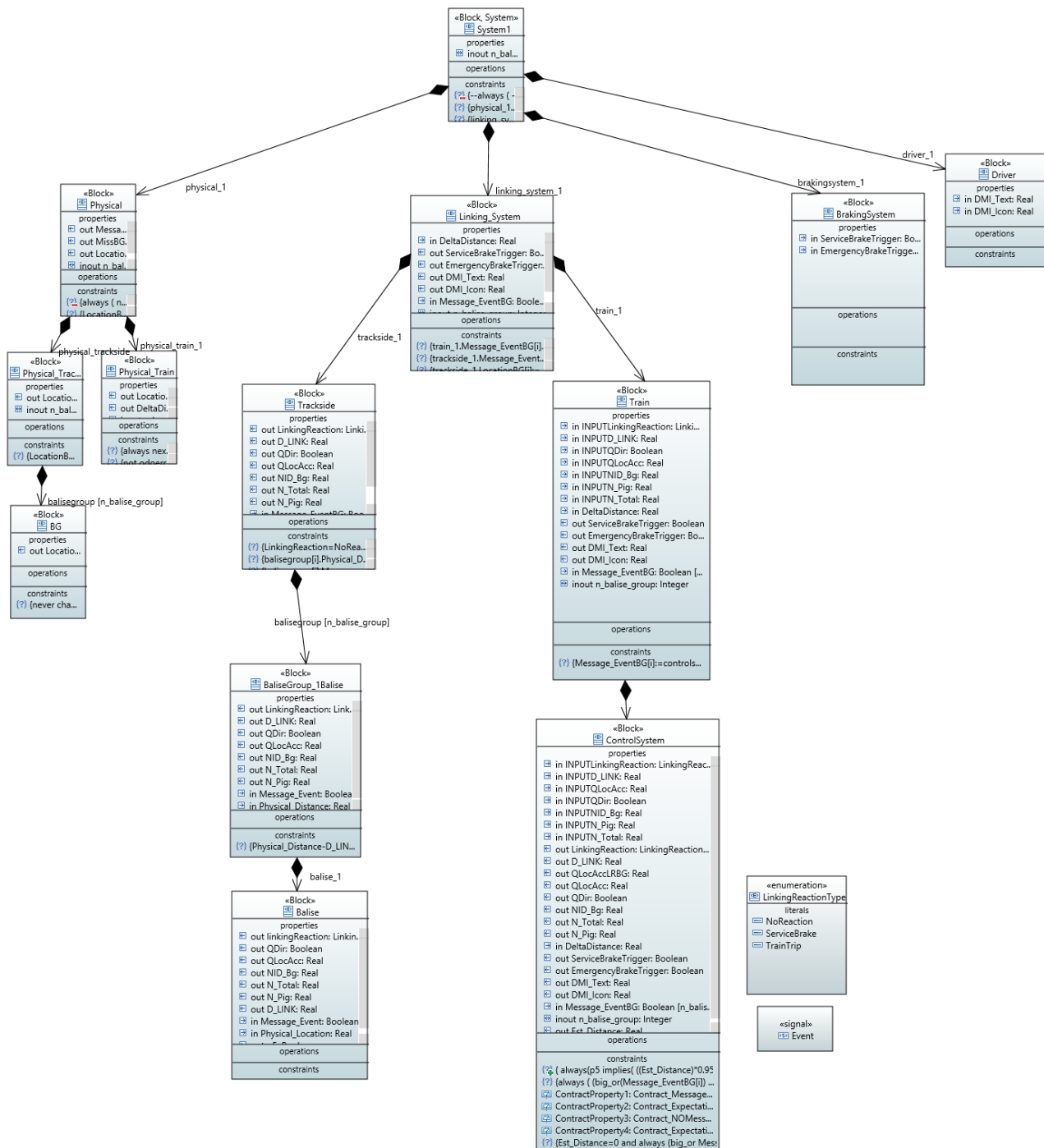
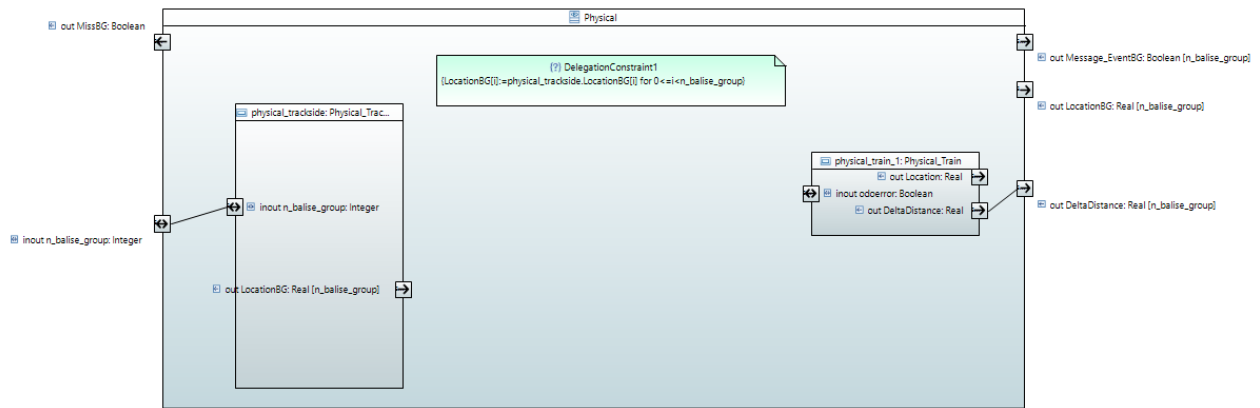


Figure 124. SysML Block Definition Diagram of the parameterized model

#### 4.2.2.3 Physical part

Physical part is composed by the block “Physical” and its sub-components, “Physical trackside” and a “Physical train”, as defined in its Internal Block Diagram, reported in Figure 125.



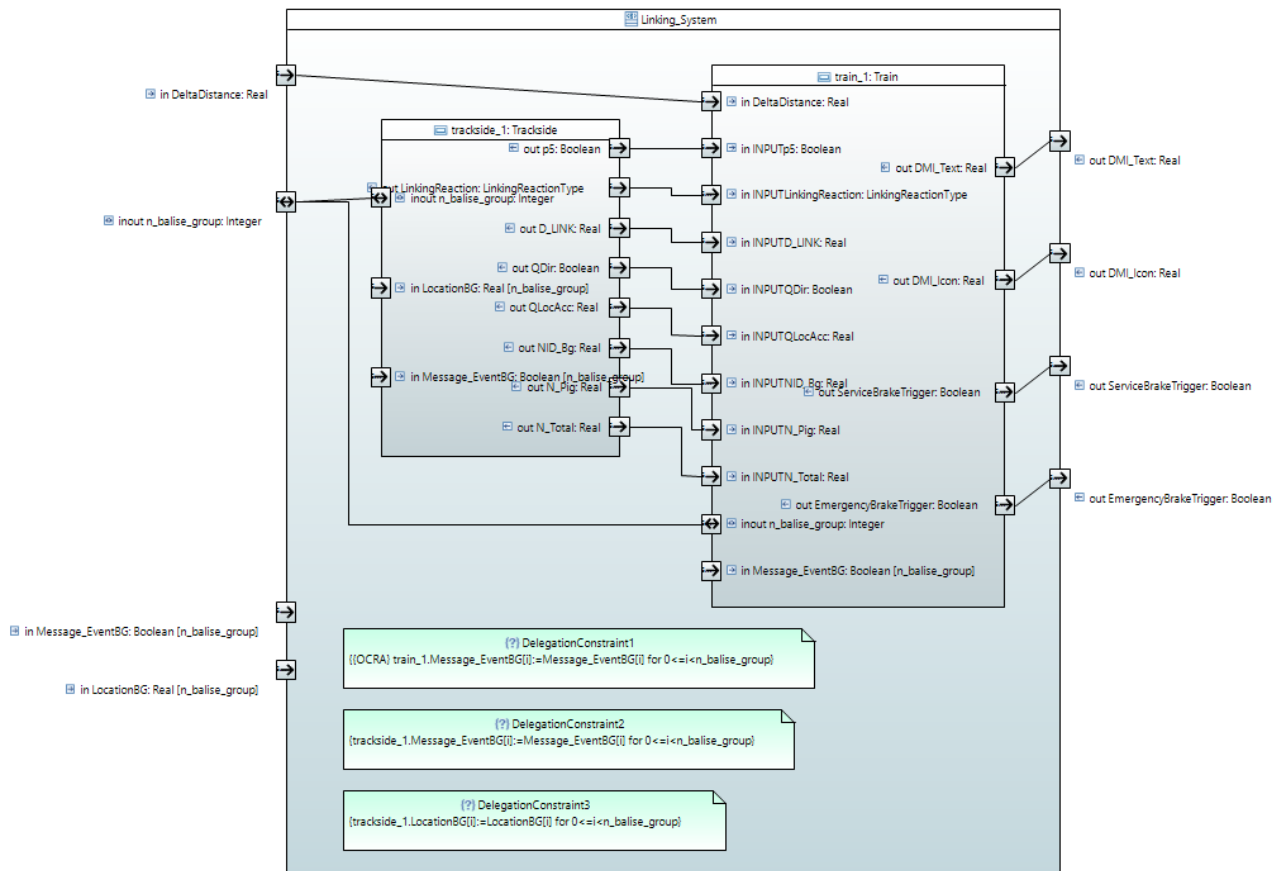
**Figure 125.** Internal Block Diagram of parameterized Physical block

In particular, Physical trackside is composed by the Balises (components “BG”) and provides to the model the location of each Balise along the route. The model is parametrized: a vector of BG is defined and in each instantiation of the general architecture the number of BG can be defined actualizing the value of the variable “n\_balise\_group”.

Physical train represents the real train with its movement along the route; it provides to the model the travelled distance of the train together with the odometric error data related to the travelled distance.

#### 4.2.2.4 Linking system part

Linking system part is composed by the block “Linking system” and its sub-components, “Trackside” and “Train”, as defined in its Internal Block Diagram, reported in Figure 126.



**Figure 126.** Internal Block Diagram of parameterized “Linking system” block



The Trackside is composed by the Balise groups and, as done for the Physical part, the model is parametrized: a vector of BG is defined and in each instantiation of the general architecture the number of BG can be defined actualizing the value of the variable “n\_balise\_group”.

The Trackside provides to the model the Linking information composed by a set of variables related to the next Balise to be overpassed by the train (distance to the next Balise, location accuracy of the next Balise, etc.). A main assumption of the model consists in considering that the Linking information refers only to the next Balise to be overpassed by the train: no iteration of this information related to further Balise is implemented. In this way, each Balise provides Linking information related to the next one.

Every time the location of the train is equal to the location of a Balise a Message is generated and the Linking information is transferred by the Trackside to the Train.

The Train is composed by the Control system block which represents the core of the ETCS on board subsystem, it constitutes the vital computer of the train, the processing unit that elaborates the inputs and determines the output in accordance to the requirements. As consequence, Control system block implements the following features:

- It receives in input the Linking information provided by trackside each time a Message is generated.
- It implements the on board Linking requirements formalized in dedicated formal properties.
- It provides as output triggers of the brakes and triggers of the text and icon to be displayed on the DMI.

#### 4.2.2.5 Brake and Driver part

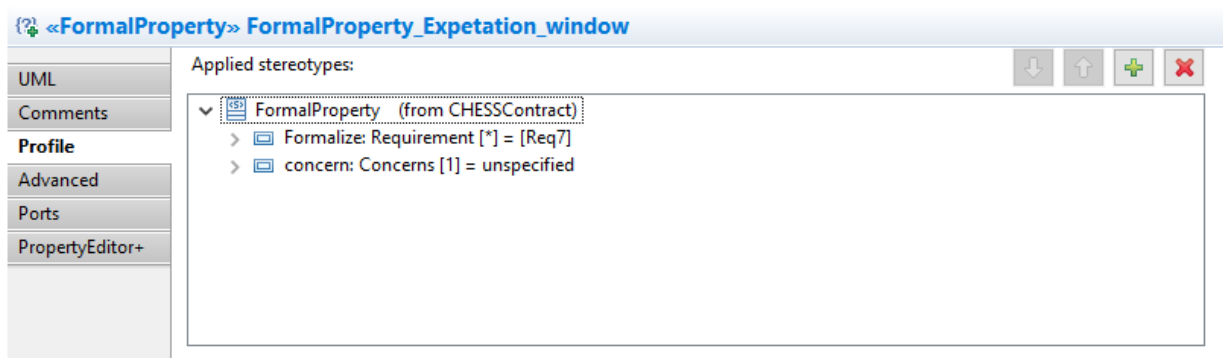
The blocks “Braking\_System” and “Driver” are block which receive as input from the Train block respectively the trigger of service and emergency brake and the text and icon to be displayed to the driver on the DMI. A more detailed modelization of the braking sub-system and of the driver machine interface could be very complex and are considered out of the scope of the present model.

### 4.2.3 Requirements Formalization

Once the system architecture was defined, the requirements of the Linking function were allocated to the corresponding components and translated into dedicated Formal properties. The Formal properties, defined by means of CHESSE platform, belongs to three different categories:

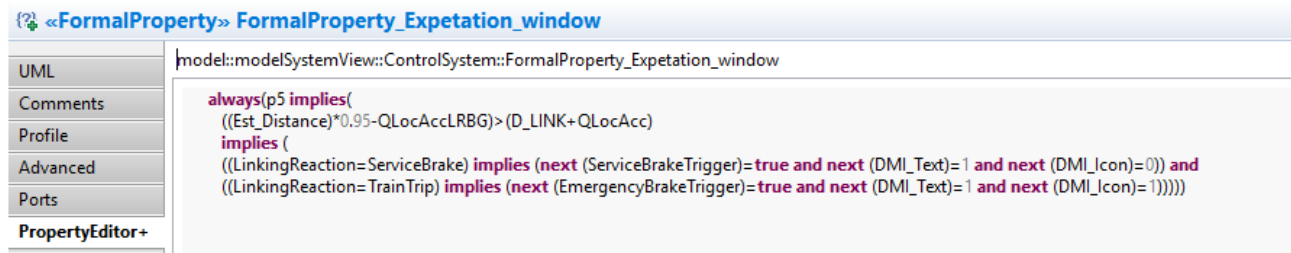
- Formal Properties which express the formalization of requirements.
- Formal Properties which formalize the assumptions defining the model boundary, domain knowledge and working hypothesis.
- Formal Properties which defines specific scenarios.

Formal properties which express the formalization of requirements were linked to the related requirement defined in the Requirements Diagram using the tab Profile, as illustrated in Figure 127.



**Figure 127.** Tab Profile to link formal properties to requirements

An example of this type of properties is the one called Expection\_window defined in “ControlSystem” block, reported in Figure 128, which defines the distance expectation window in which the on board shall expect the next linked Balise.



**Figure 128.** Expection\_window formal property

Formal Properties which formalize the assumptions defining the model boundary, domain knowledge and working hypothesis are constraints of the model related to the specific implementation. An example is the Formal properties defined in the “Physical\_Train” block formalizing that the movement of the train in the present model is considered always in positive direction.

Formal Properties which define specific scenarios are created after the instantiation of the architecture (see Section 4.2.4), because they represent the formalization of the specific instantiation of the general model with the attribution of specific values to the input variables. At the date of issue of the present deliverable, three scenarios were defined. Figure 129 illustrates the value of the basic variables and the expectation for each scenario.

Scenario 01									
BG1		BG2		BG3		BG4		BG5 Missed	
BG Physical loc.	0	BG Physical loc.	100	BG Physical loc.	200	BG Physical loc.	300	BG Physical loc.	400
Linking reac	No	Linking reac	No	Linking reac	No	Linking reac	SB	Linking reac	SB
D_LINK	100	D_LINK	100	D_LINK	100	D_LINK	100	D_LINK	100
Q_LOCACC	12	Q_LOCACC	12	Q_LOCACC	12	Q_LOCACC	12	Q_LOCACC	12
Q_DIR	1	Q_DIR	1	Q_DIR	1	Q_DIR	1	Q_DIR	1
Expectation									
Recieve correctly BG1, BG2, BG3, BG4 (BG in the correct posistion)									
Generate Service Brake trigger and Service Brake icon after the expextation window of BG5									
Scenario 02									
BG1		BG2		BG3 before					
BG Physical loc.	0	BG Physical loc.	100	BG Physical loc.	150				
Linking reac	No	Linking reac	TR	Linking reac	No				
D_LINK	100	D_LINK	100	D_LINK	100				
Q_LOCACC	12	Q_LOCACC	12	Q_LOCACC	12				
Q_DIR	1	Q_DIR	1	Q_DIR	1				
Expectation									
Recieve correctly BG1, BG2 (all BG in the correct posistion)									
Generate Emergency Brake trigger and TR icon after the expextation window of BG5									
Scenario 03									
BG1		BG2		BG3 after outside					
BG Physical loc.	0	BG Physical loc.	100	BG Physical loc.	250				
Linking reac	No	Linking reac	TR	Linking reac	No				
D_LINK	100	D_LINK	100	D_LINK	100				
Q_LOCACC	12	Q_LOCACC	12	Q_LOCACC	12				
Q_DIR	1	Q_DIR	1	Q_DIR	1				
Expectation									
Recieve correctly BG1, BG2 (all BG in the correct posistion)									
Recieve correctly BG3 but at 130m from BG2 -> outside the expectation window									
Generate Emergency Brake trigger and TR icon after the expextation window of BG3									

**Figure 129.** Scenarios formalization

Scenarios definition has the scope to create a specific implementation of the Linking model on which validation checks can be performed, see Section 4.2.6. The scope of these tests is to evaluate the correctness and coherence of the requirements formalization and to check if the expected behaviour is compatible with the requirements.

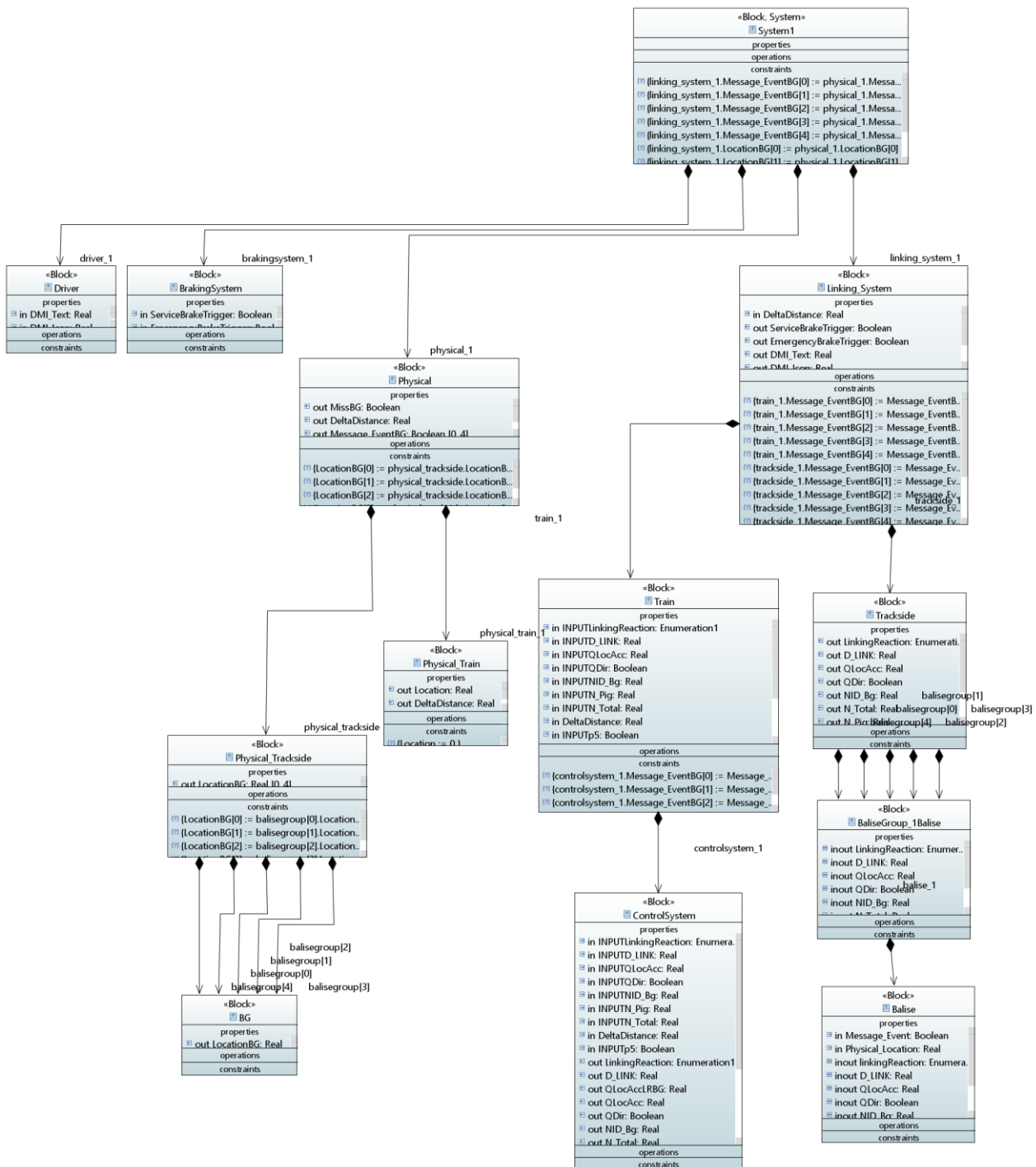
#### **4.2.4 Parametrization of the Architecture**

The present model has been designed with a parametric architecture in order to be able to create easily a specific scenario as a particular implementation of the railway system.

The parametric structure involves the definition of the number of Balise group located along the route. Instead of creating a finite number of Balise Groups, with the scope to reach a high degree of flexibility, only one Balise Group was defined both in Physical part and in Linking system part as a vector of Balise group with a multiplicity equal to the value of the variable “n\_balise\_group”. This variable has been defined as a static flow port and was replicated in each required block.

The procedure adopted to create the parametrization follows the description reported in Section 3.2.5.1.4 of the present deliverable.

The parametrized architecture was then instantiated assigning the value 5 to the variable “n\_balise\_group”, obtaining an architecture with a fixed number of components, ports, connections, and static attributes of components. Figure 130 shows the DBB of the instantiate architecture.



**Figure 130.** SysML Block Definition Diagram of the instantiated model

The parametrization of the architecture can be considered very suitable to the modelization of a railway infrastructure, that can be extremely various. Furthermore, with this feature, the AMASS platform allows the end user to define a reference architecture against which all the specific instantiation are compliant for definition.

In this sense, the parametrization of the architecture can be considered as an example of architecture driven assurance.

## 4.2.5 Error Parametrization

The parametric architecture has been extended also to the modelization of the odometric error. In a real train, the estimation of the travelled distance is derived by the signals provided by a set of sensors measuring speed and acceleration by different methods. This estimation is affected by an error which becomes higher more distance is travelled. Every time the train overpasses a linked Balise group, the odometric error is reset and the estimation of train position becomes more accurate being affected only by the accuracy of Balise group location.

The maximum error (given in percentage of the travelled distance) that train odometric function can do is defined by the reference standard: if the error entity is below this limit, the linking requirements can be met by the on board system and the expected behaviour of instantiated scenario is coherent with the defined formal properties; if the error entity is above this limit, the inaccuracy related to the estimation of the real location of the train is too high and the expected behaviour becomes no more coherent.

In order to enable the simulation of the two scenarios above mentioned, one with the odometric error inside the assurance range and the other outside the assurance range, a Boolean variable “odoerror” and related formal properties defining the assurance range of the odometric error value has been defined in the “Physical\_train” block.

## 4.2.6 Result Analysis

On the instantiated architecture, the check validation property (described in Section 3.2.4.2) was performed to check if the formalized scenarios are consistent with the other properties of the model.

The 3 scenarios mentioned in Section 4.2.3 are translated in LTL expressions as follows:

### Scenario 1:

<i>always(physical_1.physical_trackside.balisegroup[0].LocationBG=0</i>	<i>and</i>
<i>physical_1.physical_trackside.balisegroup[1].LocationBG=100</i>	<i>and</i>
<i>physical_1.physical_trackside.balisegroup[2].LocationBG=200</i>	<i>and</i>
<i>physical_1.physical_trackside.balisegroup[3].LocationBG=300</i>	<i>and</i>
<i>physical_1.physical_trackside.balisegroup[4].LocationBG=400</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[0].LinkingReaction=NoReaction</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[1].LinkingReaction=NoReaction</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[2].LinkingReaction=NoReaction</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[3].LinkingReaction=ServiceBrake</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[4].LinkingReaction=ServiceBrake</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[0].D_LINK=100</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[1].D_LINK=100</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[2].D_LINK=100</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[3].D_LINK=100</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[4].D_LINK=100</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[0].QLocAcc=12</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[1].QLocAcc=12</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[2].QLocAcc=12</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[3].QLocAcc=12</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[4].QLocAcc=12</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[0].QDir=true</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[1].QDir=true</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[2].QDir=true</i>	<i>and</i>
<i>linking_system_1.trackside_1.balisegroup[3].QDir=true</i>	<i>and</i>

*linking\_system\_1.tracksideside\_1.balisegroup[4].QDir=true) and in the future  
 (physical\_1.physical\_train\_1.Location=physical\_1.physical\_tracksideside.balisegroup[0].LocationBG and not  
 physical\_1.MissBG and in the future (*  
*physical\_1.physical\_train\_1.Location=physical\_1.physical\_tracksideside.balisegroup[1].LocationBG and not  
 physical\_1.MissBG and in the future*  
*(physical\_1.physical\_train\_1.Location=physical\_1.physical\_tracksideside.balisegroup[2].LocationBG and not  
 physical\_1.MissBG and in the future (*  
*physical\_1.physical\_train\_1.Location=physical\_1.physical\_tracksideside.balisegroup[3].LocationBG and not  
 physical\_1.MissBG and in the future*  
*(physical\_1.physical\_train\_1.Location=physical\_1.physical\_tracksideside.balisegroup[4].LocationBG and  
 physical\_1.MissBG))))*

### Scenario 2:

*always(physical\_1.physical\_tracksideside.balisegroup[0].LocationBG=0 and  
 physical\_1.physical\_tracksideside.balisegroup[1].LocationBG=100 and  
 physical\_1.physical\_tracksideside.balisegroup[2].LocationBG=150 and  
 physical\_1.physical\_tracksideside.balisegroup[3].LocationBG=300 and  
 physical\_1.physical\_tracksideside.balisegroup[4].LocationBG=400 and  
 linking\_system\_1.tracksideside\_1.balisegroup[0].LinkingReaction=NoReaction and  
 linking\_system\_1.tracksideside\_1.balisegroup[1].LinkingReaction=TrainTrip and  
 linking\_system\_1.tracksideside\_1.balisegroup[2].LinkingReaction=NoReaction and  
 linking\_system\_1.tracksideside\_1.balisegroup[3].LinkingReaction=ServiceBrake and  
 linking\_system\_1.tracksideside\_1.balisegroup[4].LinkingReaction=ServiceBrake and  
 linking\_system\_1.tracksideside\_1.balisegroup[0].D\_LINK=100 and  
 linking\_system\_1.tracksideside\_1.balisegroup[1].D\_LINK=100 and  
 linking\_system\_1.tracksideside\_1.balisegroup[2].D\_LINK=100 and  
 linking\_system\_1.tracksideside\_1.balisegroup[3].D\_LINK=100 and  
 linking\_system\_1.tracksideside\_1.balisegroup[4].D\_LINK=100 and  
 linking\_system\_1.tracksideside\_1.balisegroup[0].QLocAcc=12 and  
 linking\_system\_1.tracksideside\_1.balisegroup[1].QLocAcc=12 and  
 linking\_system\_1.tracksideside\_1.balisegroup[2].QLocAcc=12 and  
 linking\_system\_1.tracksideside\_1.balisegroup[3].QLocAcc=12 and  
 linking\_system\_1.tracksideside\_1.balisegroup[4].QLocAcc=12 and  
 linking\_system\_1.tracksideside\_1.balisegroup[0].QDir=true and  
 linking\_system\_1.tracksideside\_1.balisegroup[1].QDir=true and  
 linking\_system\_1.tracksideside\_1.balisegroup[2].QDir=true and  
 linking\_system\_1.tracksideside\_1.balisegroup[3].QDir=true and  
 linking\_system\_1.tracksideside\_1.balisegroup[4].QDir=true ) and in the future (*  
*physical\_1.physical\_train\_1.Location=physical\_1.physical\_tracksideside.balisegroup[0].LocationBG and not  
 physical\_1.MissBG and in the future*  
*(physical\_1.physical\_train\_1.Location=physical\_1.physical\_tracksideside.balisegroup[1].LocationBG and not  
 physical\_1.MissBG and in the future*  
*(physical\_1.physical\_train\_1.Location=physical\_1.physical\_tracksideside.balisegroup[2].LocationBG and not  
 physical\_1.MissBG and in the future*  
*(physical\_1.physical\_train\_1.Location=physical\_1.physical\_tracksideside.balisegroup[3].LocationBG and not  
 physical\_1.MissBG and in the future (*  
*physical\_1.physical\_train\_1.Location=physical\_1.physical\_tracksideside.balisegroup[4].LocationBG and not  
 physical\_1.MissBG ))))))*

### Scenario 3:



```

always(physical_1.physical_trackside.balisegroup[0].LocationBG=0                                and
physical_1.physical_trackside.balisegroup[1].LocationBG=100                                and
    physical_1.physical_trackside.balisegroup[2].LocationBG=250                                and
physical_1.physical_trackside.balisegroup[3].LocationBG=300                                and
    physical_1.physical_trackside.balisegroup[4].LocationBG=400                                and
linking_system_1.trackside_1.balisegroup[0].LinkingReaction=NoReaction                    and
linking_system_1.trackside_1.balisegroup[1].LinkingReaction=TrainTrip                    and
linking_system_1.trackside_1.balisegroup[2].LinkingReaction=NoReaction                    and
linking_system_1.trackside_1.balisegroup[3].LinkingReaction=ServiceBrake                and
linking_system_1.trackside_1.balisegroup[4].LinkingReaction=ServiceBrake                and
linking_system_1.trackside_1.balisegroup[0].D_LINK=100                                and
linking_system_1.trackside_1.balisegroup[1].D_LINK=100                                and
linking_system_1.trackside_1.balisegroup[2].D_LINK=100                                and
linking_system_1.trackside_1.balisegroup[3].D_LINK=100                                and
linking_system_1.trackside_1.balisegroup[4].D_LINK=100                                and
linking_system_1.trackside_1.balisegroup[0].QLocAcc=12                                and
linking_system_1.trackside_1.balisegroup[1].QLocAcc=12                                and
linking_system_1.trackside_1.balisegroup[2].QLocAcc=12                                and
linking_system_1.trackside_1.balisegroup[3].QLocAcc=12                                and
linking_system_1.trackside_1.balisegroup[4].QLocAcc=12                                and
linking_system_1.trackside_1.balisegroup[0].QDir=true                                and
linking_system_1.trackside_1.balisegroup[1].QDir=true                                and
linking_system_1.trackside_1.balisegroup[2].QDir=true                                and
linking_system_1.trackside_1.balisegroup[3].QDir=true                                and
linking_system_1.trackside_1.balisegroup[4].QDir=true) and in the future (
physical_1.physical_train_1.Location=physical_1.physical_trackside.balisegroup[0].LocationBG and not
physical_1.MissBG and in the future (
physical_1.physical_train_1.Location=physical_1.physical_trackside.balisegroup[1].LocationBG and not
physical_1.MissBG and in the future (
physical_1.physical_train_1.Location=physical_1.physical_trackside.balisegroup[2].LocationBG and not
physical_1.MissBG and in the future (
physical_1.physical_train_1.Location=physical_1.physical_trackside.balisegroup[3].LocationBG and not
physical_1.MissBG and in the future (
physical_1.physical_train_1.Location=physical_1.physical_trackside.balisegroup[4].LocationBG and not
physical_1.MissBG )))))

```

#### 4.2.7 Evidence Generation

At the end of the model implementation and result analysis, the collection of the artefacts containing the evidence of the model architecture and analysis results was performed: a document summarizing the modelling of the system components was generated through the AMASS platform, in accordance to the description reported in Section 3.4.3 of the present deliverable, see Figure 131.

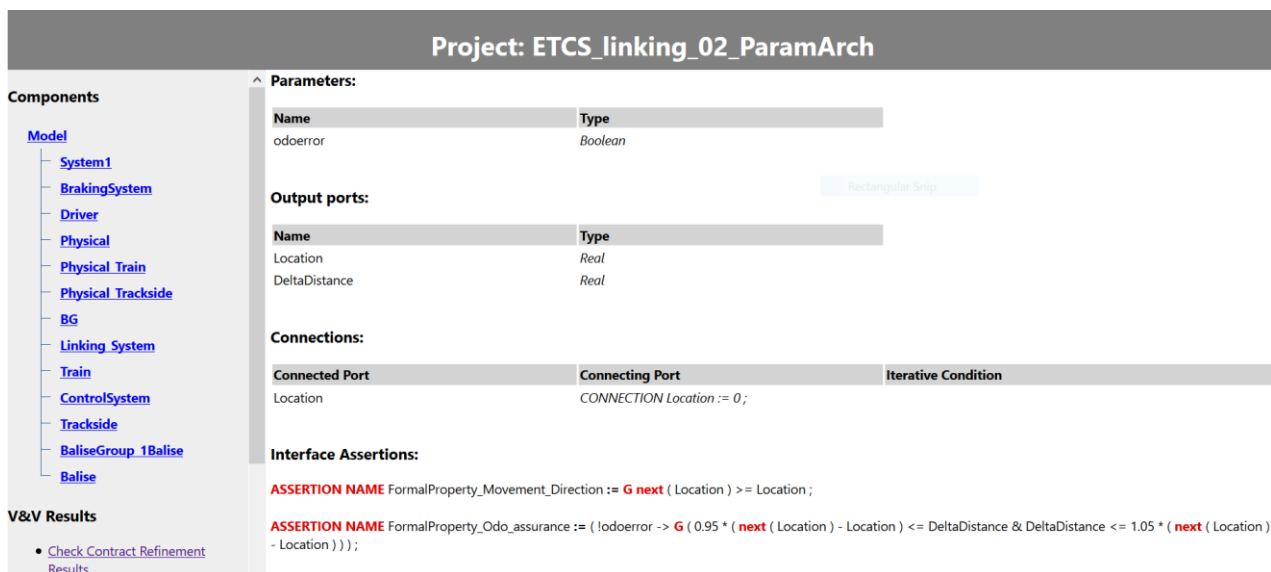


Figure 131. Screenshot of a part of the documentation generated in HTML format

## 4.3 DC Motor Drive

### 4.3.1 Case Study Description

As a simplified version of the electric powertrain of the Velox Model Car (see [68]) this case study features a standalone *direct current* (DC) motor drive system. This system is so simple that it allows easy understanding and verification according to the process described in this document, while still exhibiting all relevant properties of a typical embedded system, in particular the combination of discrete-state logic (e.g. simple run/stop state machine) with event typed input signals (e.g. “Run” button pressed) and continuous-value dynamics (e.g. proportional and integral (PI) speed controller) with continuous input and output signals (e.g. speed actual, current actual). So, it allows exploiting our approach in a nutshell, before porting the approach to the much more complex small-scale model car.

The DC drive case study is described by a set of natural language requirements and an architecture specification in SysML, followed by Simulink/Stateflow models (partner B&M) or SCADE Architect / SCADE Suite models (partner ANSYS medini) from which the actual-time C-code for the application software has been generated (a simple base software consisting of hardware abstraction layer, scheduler, signal pool and parameter system has been provided). Also, the circuit diagrams, part lists and mechanical drawings are available.

Using tools like SCADE Suite or Simulink/Stateflow for behavioural description and subsequent code generation accelerates the software development, and allow assessing the system behaviour in advance. In case of SCADE, even the code generation is performed by a formally verified code generator, certified for the highest safety integrity levels in automotive and aerospace industries, so that some verification steps on code level become obsolete.

The scientific considerations in the AMASS project regarding this case study are mainly based upon the models. For the purpose of safety assurance, a top-level hazard (unauthorized acceleration) has been defined, and safety analysis in terms of Fault Tree Analysis and FMEA has been performed, examining different typical examples of sensor failures, e.g. speed sensor wire break, and their causal relationship to the top-level hazard if no safety mechanism is built in. Subsequently, the case study is extended by a modular safety concept, including practical implementation of some of the resulting safety mechanisms (e.g. runtime wire break diagnosis for the speed sensor, based on motor voltage and current measurement and a model of



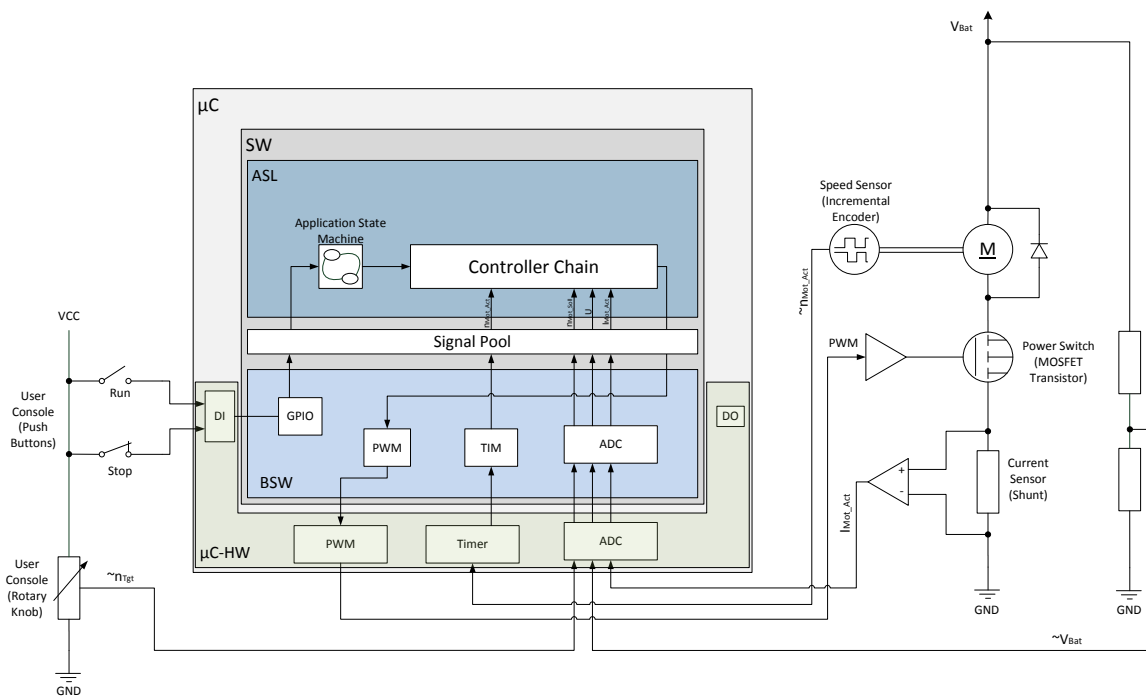
the physical laws governing the DC motor) in the actual software that runs on the DC motor drive demonstrator.

The DC motor drive system comes in several variants:

1. A very simple version with minimal hardware, allowing just operation in one sense of rotation. It features a RUN and a STOP button, a potentiometer for speed selection, a green status LED and all necessary sensors.
2. An extended version with an integrated circuit motor driver (full Transistor H-bridge) allowing accelerate/decelerate operation in both senses of rotation (clockwise and counter-clockwise), and consequently a slightly extended user interface (RUN\_FORWARD, RUN\_REVERSE, STOP). Version 1 and 2 do not include any safety features.
3. A version enhanced with some functional safety mechanisms (as a result of a FMEA safety analysis and contract-based safety concept creation), allowing to demonstrate the effectiveness of safety mechanisms that detect certain examples of a (provoked) sensor failure and reacts by putting the drive system into its “safe state” (power stage switched off and red indicator LED switched on)
4. A version that further enhances the simple safety mechanisms (allowing just shutting off the DC motor in case of any failure) towards a degradation cascade, referring to model-based replacement values in case of sensor failures, so that a continuation of operation with slightly degraded performance is still safely possible, even in presence of a single sensor failure. This latter version comes close to the degradation cascades we will have to implement for vehicles forming a platoon (CS3) for the case of communication loss or failures in a remote vehicle (in loosely coupled systems-of-systems, unreliable communication is a standard use case, and due to the partial “fail operational” requirements such systems usually exhibit, degradation cascades are a necessity in this case, see [72]).

All variants of the DC motor drive case study have not just been specified as models and specifications, but have been actually built up in a lab demonstrator (see Figure 133), which allows Hardware in the loop (HIL) testing as an additional verification method and comparison of model predictions with real-world observations. Of course, the lab demonstrator has been constructed in a way that there is no actual danger to the operator: it contains a small toy motor, is surrounded by a transparent, but massive housing, and operates at a supply voltage of 12V.

The case study has been described in some of our publications before [2][3], so that the rough description can be quoted from there: “The DC motor drive system consists of a DC motor, a power electronics board, a microcontroller board and a small user terminal with Run and Stop (in case of variant 1) push buttons, an operation mode LED and a potentiometer to select the target speed. The power board contains a metal-oxide-semiconductor *field-effect* transistor (MOSFET) to modulate the power supply voltage that is fed to the motor armature circuit according to the Pulse Width Modulated (PWM) signal generated by the microcontroller. The control software implements a state machine for the operation modes and a cascaded PI controller, implemented in Simulink. The outer speed controller generates a current target value for the inner current control loop. It evaluates the actual rotational speed, measured by an incremental encoder. The current controller reads the motor current from a sensor via an Analog to Digital Converter (ADC) and generates a target PWM signal for the power electronics corresponding to the required motor voltage. The Run and Stop buttons (BTN\_RUN and BTN\_STOP) are evaluated by a simple discrete state machine (the state machine containing just two states, namely Running and Stopped) that enables or disables the controller output. An overview is given in Figure 132. The extension to the more advanced versions works analogously, adding more buttons for forward/reverse operation and for reset of detected failures, and additional LEDs for failure and degraded status.”



**Figure 132.** Architecture Overview of DC Drive Case Study (simplest version)

The case study has been physically implemented in version 2 to 4 (which one of them depends only on software), enabling forward/reverse operation and some exemplary safety features (e.g. a detection for a wire break of the speed sensor, which is an incremental encoder). To enable a test of the safety mechanisms, the hardware test rig offers possibilities for failure injection (by pulling some bridges on the front panel), as well as a multi-pin socket for connection with a HIL (hardware-in-the-loop) system, as shown in Figure 133. This case study allows execution and validation of many of the AMASS methods explained in this document, such as specification and architecture refinement, model-based safety analysis and enhancement of the architecture with safety mechanisms, as well as fault injection testing.



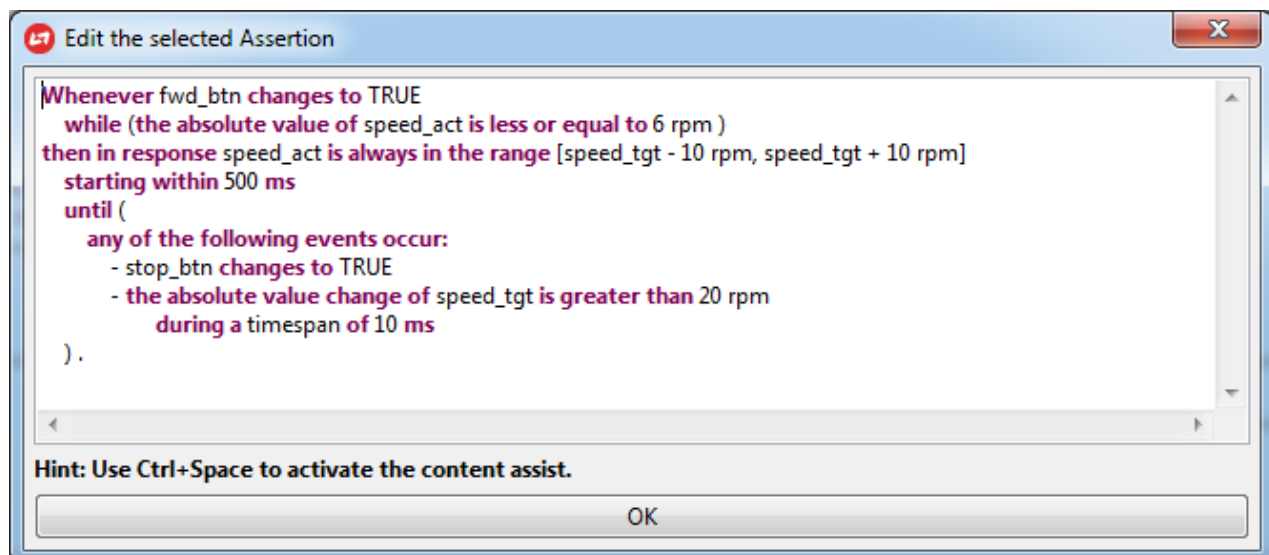
**Figure 133.** The physical implementation of the DC Drive Case Study as an experimental setting in AMASS

### 4.3.2 Experience gained with DC Drive Case Study at the stage of AMASS P2

The DC Drive has been modelled hierarchically with the prototypical tool SAVONA [12] which allows contract-based system modelling; the technical architecture has, in parallel, been modelled using SCADE Architect (the automated interface between both tools has not yet been implemented at the time of the experiments carried out). Based on Papyrus, SAVONA aims at an easy creation and validation of static system models (SysML Internal Block Diagrams) and offers support for specifying and decomposing system behaviour contracts with template expressions [12]. Several features have been added to the base functionality of Papyrus to simplify development and enhance the user experience, such as automatic creation of white-box internal block diagrams (IBD), tabular model representation with edit support, etc.

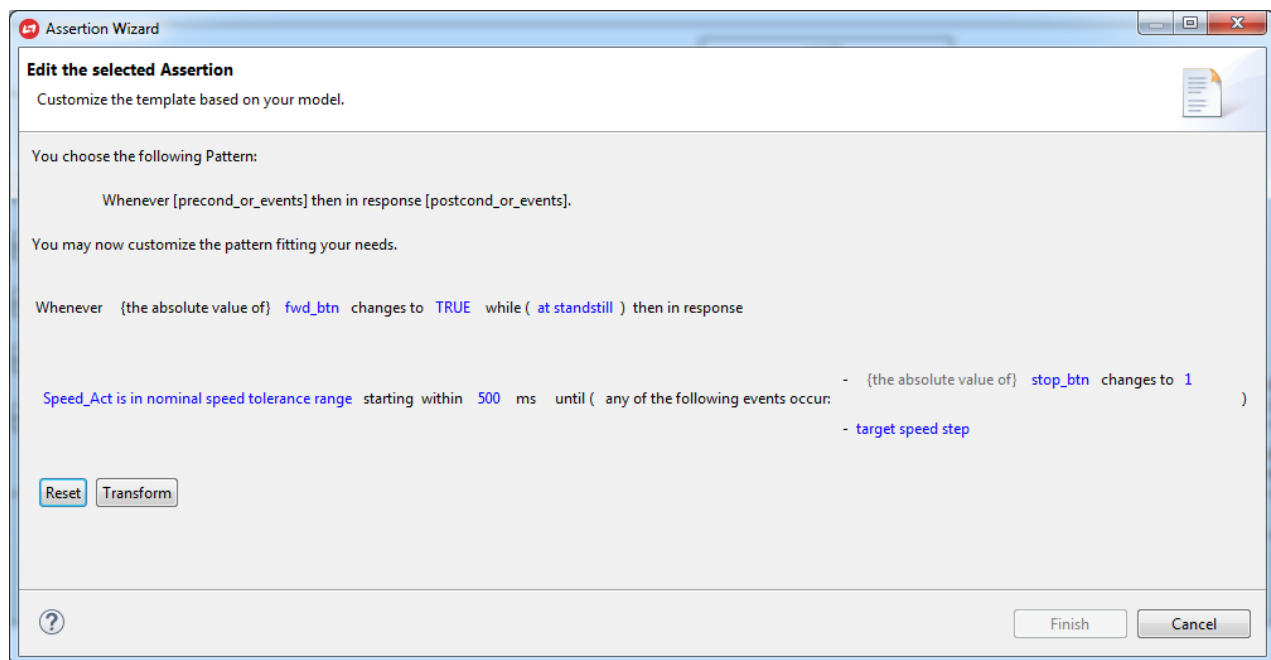
In a stepwise decomposition process, one system level is modelled after another. Starting at the top level, the system is modelled as a component (SysML Block) and its interfaces to the environment are represented as input and output ports (SysML FlowPort). At this point assertions on the system inputs, outputs and their dependencies can be expressed with semi-formal expressions in the language SSPL. By using the system model as an ontology, the tool assists the user by providing names of existing model entities to complete template based system specifications (the name of every object present in the SysML model can be referenced in a requirement or assertion). An integrated Assertion Wizard guides the user through possible pattern constructs until a matching template is found. For the more advanced users an Assertion Editor (Xtext Editor) is offered, with which assertions can be typed in as free text, but with “completion assistance”. A more detailed description of the Assertion Wizard and Assertion Editor can be found in deliverable D3.6 “Prototype for architecture-driven assurance (c)” [29]. The semi-formal assertions can be used as assumptions or guarantees in contracts of the systems components.

At the beginning, only the top-level architecture with the external interfaces were modelled in SysML. In a session with experienced engineers, a set of requirements to a typical drive system was collected and written down in natural English language, managed by a usual requirements tool (DOORS in this case). From these requirements, a selection of requirements dealing with the system's behaviour was picked for further formalization with the SAVONA tool. These requirements covered discrete-state behaviour (e.g. that the drive shall start operating when the RUN FORWARD button is pressed), continuous-signal behaviour (e.g. that the speed shall drop under a limit specified as "standstill speed limit" at max. 2000 ms after the STOP BUTTON has been hit), performance requirements (e.g. that the actual speed shall enter in a tolerance range around the selected target speed after 500 ms and stay in this range unless the target speed is changed or the STOP BUTTON is hit), and, after completion of a simplified HARA and Safety Concept, also some safety requirements (e.g. that wire break failure of the speed sensor shall not lead to self-acceleration hazard). These requirements have been put into the semi-formal specification language SSPL (an example is shown in Figure 134, the creation using a macro wizard in the AMASS tool SAVONA is shown in Figure 135). The xtext based specification wizard of SAVONA featuring syntax highlighting and auto-complete as well as dropdown boxes, in combination with the full access to the SysML architecture model as an ontology and to the parameter and signal tables, helped the analyst specifying the requirements.



**Figure 134.** A semiformal requirement for the DC Drive

The final requirements were then interpreted as guarantees (w.r.t contract-based design) of the top-level system, whereas the assumptions came from given assumptions about the usage context (e.g. that the supply voltage is always between 11V and 13V, or that the motor runs without external load torque).



**Figure 135.** Example of a macro expansion wizard allowing specification of semantically correct requirements

After the design and specification of the systems outer layer, the refinement process continues with creating subcomponents, their respective interfaces and their connections to each other and the upper system level. At this point the model can automatically be validated by detecting inconsistent ports and invalid connections to reduce system errors early during development. Semi-formal contracts are also created in the same way as for the upper system layer and assigned to the subcomponents. The stepwise refinement is then repeated for all of the subcomponents until the static functional architecture model is complete and a sufficient level of detail is reached. As we were opting for late formalization (see Section 2.2.2), the refinement from one level of the SysML architecture to the next lower level had to be performed manually, supported by the refinement check wizard. An alternative solution could be to formalize the requirements on top level (as far as they are expressible in one of the offered languages, in particular, Othello) and then export them into the CHES tool (a filter from SAVONA is available). In CHES, the system architecture and semi-formal contract specifications serve as a foundation for various V&V techniques of the AMASS tool platform, such as contract verification with OCRA [16] and fault injection via Sabotage [27].

Currently, in reference to the Table 2, this case study covers partly the phases System definition (IBD) and Functional refinement (IBD and Contracts).

## 5. Conclusions

The Architecture-Driven Assurance approach of AMASS aims at enriching the architectural design with a variety of model-based techniques that provide evidence and argument fragments to the assurance case. These model-based techniques are mainly based on SysML diagrams and various analysis techniques that comprise requirements semantic analysis, requirements quality analysis, contract-based reasoning, model checking, runtime verification, simulation-based fault-injection and monitoring, and model-based safety analysis (including FTA and FMEA), and analysis based on parametrized architectures.

This document defines a methodological guide to use the AMASS Architecture-Driven Assurance approach and the related tool support by the AMASS platform. It describes the workflow, the steps to perform each activity, and some case studies that are provided with the AMASS platform prototype.

This guide refers to the final Prototype P2 of the AMASS platform.

## Abbreviations and Definitions

<i>Abbreviation</i>	<i>Explanation</i>
ACC	Adaptive Cruise Control
ADC	Analog to Digital Converter
API	Application Programming Interface
ARP	Aerospace Recommended Practice
ASIL	Automotive Safety Integrity Level
BDD	Block Definition Diagram
BSCU	Braking System Control Unit
BTM	Balise Module Transmission
CAE	Claims, Arguments and Evidence
CHESSML	CHESS Modelling Language
CDO	Connected Data Objects
CFT	Component Fault Tree
CPS	Cyber-Physical Systems
CSV	Comma Separated Value
CTL	Computation-tree Temporal Logic
DAL	Design Assurance Level
DC	Direct Current
DMI	Driver Machine Interface
DOORS	Dynamic Object-Oriented Requirements System
ERTMS	European Rail Traffic Management System
ETCS	European Train Control System
FHA	Functional Hazard Analysis
FI	Fault Injection
FMEA	Failure Mode and Effect Analysis
FTA	Fault Tree Analysis
FSM	Finite-State Machine
GCSL	Goal and Contract Specification Language
GSM	Global System for Mobile communications
GSN	Goal-Structuring-Notation
GUI	Graphical User Interface
HARA	Hazard Analysis and Risk Assessment
HAZOP	Hazard and Operability study
HIL	Hardware in the loop
HRELTL	Hybrid LTL
HTML	HyperText Markup Language
IBD	Internal Block Diagram
IEC	International Electrotechnical Commission
IESE	Institute for Experimental Software Engineering
ISO	International Organization for Standardization



<i>Abbreviation</i>	<i>Explanation</i>
KM	Knowledge Management
LTL	Linear-time Temporal Logic
MARTE	Modeling and Analysis of Real Time and Embedded systems
MBSA	Model-Based Safety Analysis
MIL	Model in the Loop
MOSFET	Metal-oxide-semiconductor field-effect transistor
NLP	Natural-Language Processing
OCRA	Othello Contracts Refinement Analysis
OMG	Object Management Group
OSLC	Open Services for Lifecycle Collaboration
OSS	Othello System Specification
PI	Proportional and Integral
PSL	Property Specification Language
PWM	Pulse Width Modulated
RSL	Requirement Specification Language
RQA	Requirement Quality Analyzer
SAE	Society of Automotive Engineers
SCM	System Conceptual Model
SERE	Sequential Extended Regular Expressions
SIL	Software in the Loop
SKB	System Knowledge Base
SMV	Symbolic Model Verifier
SOTIF	Safety Of The Intended Functionality
SSPL	System Specification Pattern Language
STL	Signal Temporal Logic
STO	Scientific Technical Objective
SUT	System Under Test
SW	Software
SysML	System Modelling Language
UML	Unified Modelling Language
V&V	Verification and Validation
VHDL	Very High speed integrated circuit Hardware Description Language
WBS	Wheel Brake System
XSAP	eXtended Safety Assessment Platform



## References

- [1] P. Koopman: Better Embedded System Software, Carnegie Melon University, 2010.
- [2] B. Kaiser, B. Monajemi Nejad, D. Kusche and H. Schulte: Systematic Design and Validation of Degradation Cascades for Safety-Relevant Systems. In Proc. ESREL 2017
- [3] B. Kaiser, R. Weber, M. Oertel, E. Böde, B. Monajemi and J. Zander: Contract-Based Design of Embedded Systems Integrating Nominal Behaviour and Safety. In: Complex Systems Informatics and Modeling Quarterly (CSIMQ), Issue 4, October 2015, Pages 66-91
- [4] Requirements Working Group. International Council on Systems Engineering (INCOSE), Guide for Writing Requirements, (2012) 59. <http://www.incose.org>
- [5] A. Mavin, P. Wilkinson, A. Harwood & M. Novak: Easy approach to requirements syntax (EARS). Requirements Engineering Conference, 2009. RE'09. 17th IEEE International, 2009, 317-322
- [6] J. Eckhardt, A. Vogelsang, H. Femmer & P. Mager: Challenging incompleteness of performance requirements by sentence patterns. Requirements Engineering Conference (RE), 2016 IEEE 24th International, 2016, 46-55
- [7] O. Daramola, G. Sindre & T. Stalhane: Pattern-based security requirements specification using ontologies and boilerplates. Requirements Patterns (RePa), 2012 IEEE Second International Workshop on, 2012, 54-59
- [8] Definition and exemplification of RSL and RMM. Deliverable DSP2R2.1M1, Cost-efficient methods and processes for safety relevant embedded systems. CESAR, CESAR, 2010
- [9] DANSE "GCSL syntax, semantics and meta-model" deliverable. DANSE Research Project, DANSE Research Project, 2015
- [10] CESAR "Cost-efficient Methods and Processes for Safety-relevant Embedded Systems" deliverable. CESAR Research Project. <https://artemis-ia.eu/project/1-cesar.html>
- [11] DANSE "Designing for Adaptability and evolution in System of systems Engineering" deliverable. DANSE Research Project <http://danse-ip.eu/home/>
- [12] M. Grabowski: Why Templates on System Behavior Are Not Used in Practice Yet: A Proposal for Enhancements, Application and Formalization. MSc Thesis. Technical University of Berlin, 2017
- [13] R. Adler, D. Domis, K. Höfig, S. Kemmann, T. Kuhn, J.P. Schwinn and M. Trapp, "Integration of Component Fault Trees into the UML", MODELS 2010
- [14] D. Larsson and Reiner Hähnle, "Symbolic Fault Injection", VERIFY 2007
- [15] M. Bozzano, A. Cimatti, C. Mattarei, and S. Tonetta. Formal Safety Assessment via Contract-Based Design. In ATVA, 2014.
- [16] OCRA: A tool for Contract-Based Analysis. Available at <https://es.fbk.eu/tools/ocra/>
- [17] nuXmv: a new eXtended model verifier. Available at <http://nuxmv.fbk.eu>
- [18] Society of Automotive Engineers (SAE). AIR 6110, Contiguous Aircraft/ System Development Process Example, December 2011
- [19] M. Bozzano, A. Cimatti, A.F. Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, S. Tonetta: Formal Design and Safety Analysis of AIR6110 Wheel Brake System. CAV (1) 2015: 518-535
- [20] ERA UNISIG EEIG ERTMS USERS GROUP, ERTMS/ETCS Subset-026 System Requirements Specification, 15/06/2016
- [21] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.C. Fabre, J.C. Laprie, E. Martins, D. Powell: Fault injection for dependability validation: A methodology and some applications. In: IEEE Trans. Software Engineering., vol. 16, pp. 166–182, Basic Concepts and Taxonomy of Dependable and Secure Computing, 1990
- [22] A. Benso, S. Di Carlo: The art of fault injection. In: Journal of Control Engineering and Applied Informatics, Vol.13, No. 4, pp. 9-18, 2011

- [23] J. Vinter, L. Bromander, P. Raistrick, H. Edler: Fiscade - a fault injection tool for SCADE models. In: Automotive Electronics 2007 3rd Institution of Engineering and Technology Conference, pp. 1–9., 2007
- [24] S. Reiter, M. Zeller, K. Höfig, A. Viehl, O. Bringmann, W. Rosenstiel: Verification of Component Fault Trees Using Error Effect Simulations, IMBSA 2017
- [25] B. Bittner, M. Bozzano, R. Cavada, A. Cimatti, M. Gario, A. Griggio, C. Mattarei, A. Micheli and G. Zampedri (2016) The xSAP Safety Analysis Platform. In Proceedings of TACAS 2016
- [26] The xSAP safety analysis platform, <https://xsap.fbk.eu/>
- [27] G. Juez, E. Amparan, R. Lattarulo, A. Ruiz, J. Perez, H. Espinoza: Early Safety Assessment of Automotive Systems Using Sabotage Simulation-Based Fault Injection Framework, SAFECOMP 2017
- [28] AMASS D3.3 “Design of the AMASS tools and methods for architecture-driven assurance (b)” deliverable, 31 March 2018
- [29] AMASS D3.6 “Prototype for architecture-driven assurance (c)” deliverable, 31 August 2018
- [30] AMASS D3.7 “Methodological guide for architecture-driven assurance (a)” deliverable, 30 November 2017
- [31] AMASS D5.6 “Prototype for seamless interoperability (c)” deliverable, 20 September 2018
- [32] AMASS D2.4 “AMASS Reference Architecture (c)” deliverable, 04 June 2018
- [33] AMASS D2.5 “AMASS user guidance and methodological framework” deliverable, October 2018
- [34] M. Kooli, G. Di Natale: A Survey on Simulation-Based Fault Injection Tools for Complex Systems, 2014 9th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS), 2014
- [35] H. Ziade, R. Ayoubi, R. Velazco: A Survey on Fault Injection Techniques, The International Arab Journal of Information Technology, Vol. 1, No. 2, July 2004
- [36] A. Sangiovanni-Vincentelli, W. Damm and R. Passerone. Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. European Journal of Control, 18(3):217-238, 2012
- [37] A. Benveniste, B. Caillaud, R. Passerone: A Generic Model of Contracts for Embedded Systems. CoRR abs/0706.1456 (2007)
- [38] S.S. Bauer, A. David, R. Hennicker, K.G. Larsen, A. Legay, U. Nyman, A. Wasowski: Moving from Specifications to Contracts in Component-Based Design. FASE 2012: 43-58
- [39] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A.L. Sangiovanni-Vincentelli, W. Damm, T.A. Henzinger, K.G. Larsen: Contracts for System Design. Foundations and Trends in Electronic Design Automation 12(2-3): 124-400 (2018)
- [40] A. Cimatti, S. Tonetta: A Property-Based Proof System for Contract-Based Design. EUROMICRO-SEAA 2012: 21-28
- [41] D.D. Cofer, A. Gacek, S.P. Miller, M.W. Whalen, B. LaValley, L. Sha: Compositional Verification of Architectural Models. NASA Formal Methods 2012: 126-140
- [42] A. Cimatti, M. Dorigatti, S. Tonetta: OCRA: A tool for checking the refinement of temporal contracts. ASE 2013: 702-705
- [43] The PROSYD project on property-based system design, 2007. <http://www.prosyd.org>.
- [44] K. Claessen. A coverage analysis for safety property lists. In FMCAD, pages 139–145. IEEE, 2007.
- [45] I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, and A. Cimatti. Formal analysis of hardware requirements. In DAC, pages 821–826, 2006.
- [46] A. Cimatti, M. Roveri, A. Susi, S. Tonetta: Validation of requirements for hybrid systems: A formal approach. ACM Trans. Softw. Eng. Methodol. 21(4): 22:1-22:34 (2012)
- [47] E.M. Clarke, O. Grumberg, D.A. Peled: Model checking. MIT Press 2001, ISBN 978-0-262-03270-4, pp. I-XIV, 1-314

- [48] The SafeCer project (Certification of Software-intensive Systems with Reusable Components) [http://cordis.europa.eu/project/rcn/103721\\_en.html](http://cordis.europa.eu/project/rcn/103721_en.html) and [http://cordis.europa.eu/project/rcn/105610\\_en.html](http://cordis.europa.eu/project/rcn/105610_en.html)
- [49] The OPENCROSS project (Open Platform for Evolutionary Certification Of Safety-critical Systems) <http://www.opencross-project.eu/>
- [50] The CHES tool <https://www.polarsys.org/projects/polarsys.chess>
- [51] M. Bozzano, A. Villafiorita: Design and Safety Assessment of Critical Systems. CRC Press (Taylor and Francis), an Auerbach Book (2010)
- [52] A. Pnueli: The Temporal Logic of Programs. FOCS 1977: 46-57
- [53] O. Lichtenstein, A. Pnueli, L.D. Zuck: The Glory of the Past. Logic of Programs 1985: 196-218
- [54] Z. Manna, A. Pnueli: The temporal logic of reactive and concurrent systems - specification. Springer 1992, ISBN 978-3-540-97664-6, pp. I-XIV, 1-427
- [55] R. Alur, T.A. Henzinger: Logics and Models of Real Time: A Survey. REX Workshop 1991: 74-106
- [56] S. Tonetta: Linear-time Temporal Logic with Event Freezing Functions. GandALF 2017: 195-209
- [57] A. Cimatti, M. Roveri, S. Tonetta: HRELTL: A temporal logic for hybrid systems. Inf. Comput. 245: 54-71 (2015)
- [58] S. Wilson, T. Kelly and J. McDermid: Safety Case Development: Current Practice, Future Prospects. In: Safety and Reliability of Software Based Systems. 1997.
- [59] O. Maler, D. Nickovic: Monitoring Temporal Properties of Continuous Signals. FORMATS/FTRTFT 2004: 152-166
- [60] J. A. Estefan, "Survey of Model-Based System Engineering (MBSE) Methodologies", INCOSE-TD-2007-003-02, INCOSE MBSE Initiative, Revision B, June 2008.
- [61] <https://www.polarsys.org/chess/publis/CHESMLprofile.pdf>
- [62] E. Parra, C. Dimou, J. Llorens, V. Moreno, A. Fraga. "A methodology for the classification of quality of requirements using machine learning techniques." Information and Software Technology 67 (2015): 180-195.
- [63] <https://www.reusecompany.com/requirements-quality-analyzer>
- [64] <https://www.reusecompany.com/knowledgemanager>
- [65] R.E. Bloomfield, P.G. Bishop: Safety and Assurance Cases: Past, Present and Possible Future - an Adelard Perspective. SSS 2010: 51-67
- [66] OMG SysML Home, <http://www.omgsysml.org/>
- [67] I. Šljivo: "Assurance Aware Contract-Based Design for Safety-Critical Systems." PhD Thesis. Mälardalen University. (2018).
- [68] B. Kaiser, O. Kreuzmann, A. Ferdowsizadeh et al: VeloxCar – A model car as a demonstrator for automated and networked vehicle systems. Technical Report. Assystem Germany, 2017
- [69] M. Bender, T. Maibaum, M. Lawford, A. Wassyl: Positioning Verification in the Context of Software/System Certification. Proceedings of the 11th International Workshop on Automated Verification of Critical Systems, 2011
- [70] D. Domis, M. Forster, S. Kemmann, M. Trapp: Safety Concept Trees, Annual Reliability and Maintainability Symposium (RAMS) 55, 2009, Fort Worth, Texas, 2009
- [71] M. Grabowski, B. Kaiser, and Y. Bai: Systematic Refinement of CPS Requirements using SysML, Template Language and Contracts. In: Schaefer, I., Karagiannis, D., Vogelsang, A., Méndez, D. & Seidl, C. (Hrsg.), Modellierung 2018. Bonn: Gesellschaft für Informatik e.V.. (S. 245-260).
- [72] I. Šljivo, B. Gallina, and B. Kaiser: Assuring Degradation Cascades of Car Platoons via Contracts. In: Proceedings 6th International Workshop on Next Generation of System Assurance Approaches for Safety-Critical Systems. 2017

- 
- [73] P. Nuzzo, J.B. Finn, A. Iannopolo, A.L. Sangiovanni-Vincentelli: Contract-based design of control protocols for safety-critical cyber-physical systems. DATE 2014: 1-4
  - [74] J. Li, P. Nuzzo, A.L. Sangiovanni-Vincentelli, Y. Xi, D. Li: Stochastic contracts for cyber-physical system design under probabilistic requirements. MEMOCODE 2017: 5-14
  - [75] A. Zutshi, S. Sankaranarayanan, J.V. Deshmukh, J. Kapinski, X. Jin: Falsification of safety properties for closed loop control systems. HSCC 2015: 299-300
  - [76] J.V. Deshmukh, X. Jin, J. Kapinski, O. Maler: Stochastic Local Search for Falsification of Hybrid Systems. ATVA 2015: 500-517
  - [77] J. Kapinski, J.V. Deshmukh, X. Jin, H. Ito, K.R. Butts: Simulation-guided approaches for verification of automotive powertrain control systems. ACC 2015: 4086-4095
  - [78] H. Roehm, J. Oehlerking, T. Heinz, M. Althoff: STL Model Checking of Continuous and Hybrid Systems. ATVA 2016: 412-427
  - [79] J. Eddeland, S. Miremadi, M. Fabian, K. Åkesson: Objective functions for falsification of signal temporal logic properties in cyber-physical systems. CASE 2017: 1326-1331
  - [80] K. Claessen, N. Smallbone, J. Liden Eddeland, Z. Ramezani, K. Åkesson, S. Miremadi: Applying Valued Booleans in Testing of Cyber-Physical Systems. MT@CPSWeek 2018: 8-9
  - [81] ISO: 26262 -"Road vehicles-Functional safety" (2011)
  - [82] I. Šljivo, B. Gallina, J. Carlson, and H. Hansson: Strong and Weak Contract Formalism for Third-Party Component Reuse. In: Proceedings of 3rd International Workshop on Software Certification, pages 359—364. November 2013.
  - [83] I. Šljivo, B. Gallina, J. Carlson, H. Hansson, and S. Puri: Tool-Supported Safety-Relevant Component Reuse: From Specification to Argumentation. In: Proceedings of 23rd International Conference on Reliable Software Technologies - Ada-Europe 2018, pages 19—33. June 2018.

## Appendix A: Document changes respect to D3.7

New Sections:

Section	Title
2.2.2	Transfer of the AMASS Main Idea to an Industry-Proof Working Process
3.2.3.2	Requirements Formalization Using SAVONA
3.2.4.3.6	Correctness metric for models
3.2.4.3.7	Checklist metrics
3.2.5.1.2	Architectural Refinement in SAVONA
3.2.5.1.3	Architectural Pattern-based support
3.2.5.1.4	Parameters and Configurations
3.2.5.2.2	Contract Refinement in SAVONA
3.3.2.3	FMEA generation
3.3.1.2	Eclipse solution: Integration with the AMASS Platform

Modified Sections:

Section	Title	Change
	Executive Summary	Updated
2.1.2	Contract-Based Design	Rephrased some parts.
2.1.5	Verification and Validation of Behavioural Models	Added note on AMASS support
2.1.6.1	Simulation-Based	Revised section including simulation-based analysis and minimal cut sets analysis into one section on model-based safety analysis.
2.2.3	Tool Support Overview	Extended with clearer overview of external tools
3.1	Workflow Overview	Figure 13 and 14 changed
3.2.1.1	Requirements Specification/Import	Added info about requirements traceability
3.2.4.3	Requirement Quality Analyser in the AMASS platform	Extended section of the integration of RQA with AMASS. Included new metrics developed: correctness metrics for models and metrics of checklists.
3.2.5.2.1	Contract Refinement in CHESS	Added info about requirements refinement
3.2.6	Component's Nominal and Faulty Behaviour Definition	Updated with extended support for state machines and error model.
3.3.1	Simulation-based Fault Injection	Introduction modified and figure 88 changed
3.4.2	Link to Architectural Entities	Minor updates.
3.4.3	Document Generation	Updated with new features added to the reports.
4.2	ETCS	Extended section with the description of the model and link to the methodological guide.
4.3	DC Motor Drive	Minor updates.
5	Conclusions	Minor updates.