

## ECSEL Research and Innovation actions (RIA)



# AMASS

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

### Prototype for Architecture-Driven Assurance (b) D3.5

<b>Work Package:</b>	WP3: Architecture-Driven Assurance
<b>Dissemination level:</b>	PU = Public
<b>Status:</b>	Final
<b>Date:</b>	September,29 2017
<b>Responsible partner:</b>	B&M
<b>Contact information:</b>	Peter M. Kruse <peter.kruse@berner-mattner.com>
<b>Document reference:</b>	AMASS_D3.5_WP3_B&M_V1.0

#### PROPRIETARY RIGHTS STATEMENT

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

---

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

---

## Contributors

Names	Organisation
Stefano Puri	Intecs (INT)
Peter M. Kruse	Assystem Germany (B&M)
Eugenio Parra, José Luis de la Vara, Gonzalo Génova, Valentín Moreno	Universidad Carlos III de Madrid (UC3)
Luis Alonso	The REUSE Company (TRC)
Alberto Debiasi	Fondazione Bruno Kessler (FBK)
Garazi Juez, Estibaliz Amparan, Huáscar Espinoza, Alejandra Ruiz	Tecnalia Research & Innovation (TEC)
Tomáš Kratochvíla, Petr Bauch, Vít Koksa	Honeywell (HON)
Jaroslav Bendík	Masaryk University (UOM)

## Reviewers

Names	Organisation
Gael Blondell (Peer reviewer)	Eclipse Foundation (ECL)
Martin Helmut (Peer reviewer)	Virtual Vehicle (VIF)
Cristina Martinez (Quality Manager)	Tecnalia Research & Innovation (TEC)
Jose Luis de la Vara (Technical Committee Review)	Universidad Carlos III de Madrid (UC3)

# TABLE OF CONTENTS

<b>Executive Summary.....</b>	<b>6</b>
<b>1. Introduction .....</b>	<b>7</b>
<b>2. Implemented Functionality .....</b>	<b>10</b>
2.1 Scope .....	10
2.2 Implemented Requirements .....	11
2.2.1 System Component Specification.....	11
2.2.2 System Architecture Modelling for Assurance .....	20
2.2.3 V&V-based Assurance Impact Assessment .....	22
2.2.4 Contract-based Assurance Composition .....	43
<b>3. Installation and User Manuals.....</b>	<b>48</b>
<b>4. Implementation Description .....</b>	<b>49</b>
4.1 Implemented Modules.....	49
4.1.1 System Component Specification Block .....	49
4.1.2 Architecture-Driven Assurance Block.....	49
4.2 Source Code Description .....	50
4.2.1 System Component Specification Block .....	50
4.2.2 Architecture-Driven Assurance Block.....	52
<b>5. Conclusions .....</b>	<b>59</b>
<b>Abbreviations .....</b>	<b>60</b>
<b>References.....</b>	<b>62</b>

## List of Figures

Figure 1.	AMASS Building blocks .....	8
Figure 2.	Layered structure of AMASS tool modules .....	10
Figure 3.	Description of main building blocks: System component specification and architecture driven assurance .....	11
Figure 4.	Papyrus editor .....	12
Figure 5.	Modelling FormalProperty .....	14
Figure 6.	First step in the Assertion-Wizard: Select a General Pattern Type to formulate an assertion. Each selection features a short description and example to offer the user an easy decision. ....	16
Figure 7.	Second step in the Assertion-Wizard: Choose a pattern instantiation of the previously selected general pattern type .....	16
Figure 8.	Last step of the Assertion-Wizard: Refine the pattern instance with names of available model elements. Only element names which are valid for the corresponding placeholder are allowed to be used .....	17
Figure 9.	Pattern-suggestion feature of the Assertion Editor.....	17
Figure 10.	Contract and FormalProperty example.....	18
Figure 11.	Assign Contract to Component .....	19
Figure 12.	After the creation of a ContractProperty, a Popup appears to decide whether a new contract has to be created or an existing one has to be instantiated .....	20
Figure 13.	Links through EAnnotation.....	21
Figure 14.	Links through traceability meta-model.....	22
Figure 15.	Automatic translation general diagram - From NL to LTL.....	23
Figure 16.	Configuration window to setup the K number .....	23
Figure 17.	Metric results visualization .....	24
Figure 18.	Result window for reliable experiment.....	24
Figure 19.	Window result for not reliable experiment .....	25
Figure 20.	Correctness metrics related to nouns.....	26
Figure 21.	Correctness metrics related to verbs.....	27
Figure 22.	Tool to generate classifiers using machine learning with metrics of RQA .....	28
Figure 23.	Example of classifier .....	28
Figure 24.	Completeness metrics for models .....	29
Figure 25.	Consistency metrics for models .....	29
Figure 26.	Saving snapshot with the quality of the project .....	30
Figure 27.	Graphical representation of the quality evolution .....	30
Figure 28.	Information of the snapshot .....	31
Figure 29.	GUI element used to run the V&V Manager.....	32
Figure 30.	Switch in on the V&V Result view .....	32
Figure 31.	Example of requirements from Gesture Recognition system (Case Study 7) that are only trivially realisable .....	34
Figure 32.	Example of requirements that are consistent, non-redundant and not realisable .....	34
Figure 33.	Details for requirements checking .....	35
Figure 34.	Checking and proposed error handling .....	36

Figure 35. CHESS error model state machine .....	37
Figure 36. Example of fault tree represented as a table.....	37
Figure 37. Example of fault tree represented as tree .....	38
Figure 38. Sabotage design architecture.....	39
Figure 39. Example of a Massif model .....	40
Figure 40. Example of the generated Fault Injector code.....	41
Figure 41. Example of a saboteur code.....	41
Figure 42. Analysis Context .....	42
Figure 43. Contract Editor with content assist.....	44
Figure 44. Hierarchical view of the system decomposed into sub-components and contracts .....	44
Figure 45. Contract Refinement View .....	45
Figure 46. Part of the OCRA input file, also called OSS (OCRA System Specification). It describes the system architecture represented by a tree of components (given by the decomposition into sub-components) .....	46
Figure 47. Selecting analysis context for contract refinement .....	46
Figure 48. Part of an .SMV file representing the behaviour of the leaf components of the model.....	47
Figure 49. In this example, for each contract the results of the Contract-based verification are listed in the Trace View .....	48
Figure 50. Tool modules for System Component Specification .....	49
Figure 51. CHESS plugins supporting Contract Based Design .....	51
Figure 52. CHESS methodology constraint.....	52
Figure 53. Massif and Sabotage meta-models.....	53
Figure 54. Connection between Sabotage and Massif meta-models .....	53
Figure 55. Code Generation workspace .....	54
Figure 56. Diagram showing the dependences among the plugins. The direction of the arrow means that the origin plugin depends on the target plugin.....	58

## Executive Summary

The deliverable D3.5 “Prototype for Architecture-Driven Assurance (b)” is the second output of the AMASS task T3.3 *Implementation for Architecture-driven Assurance*, whose objective is the development of a tooling framework to support architecture-driven assurance. D3.5 is the evolution of D3.4, which described the first prototype.

AMASS has planned three prototype iterations for the framework; this deliverable reports the status of the aforementioned tooling framework for the second prototype release (Prototype P1), in particular for what regards the system component specification and the tooling framework supporting architecture-driven assurance, by describing the supported functionalities and the details about implementation.

This deliverable takes into account the work performed in the other project work-packages, mainly WP2, WP4, WP5 and WP6 because they have strong dependencies with T3.3. Indeed, in this deliverable a set of functionalities regarding the system component specification have been selected from AMASS deliverable D2.1 “Business cases and high-level requirements”. D3.5 describes the technologies that allow implementing all selected functionality.

The logical structural view of the AMASS reference tool architecture elaborated in deliverable D2.3 “AMASS Reference Architecture” [6] has been also considered in this deliverable; in particular physical components have been mapped to the logical tool components *Component Editor* and *Contract Editor* identified in deliverable D2.3.

WP4 and WP5 results have been particularly useful for what concerns the argumentation and evidence metamodel specification; indeed one important point related to the implementation for architecture-driven assurance is related to the way system architecture-related information can be traced to the argumentation and evidence models. Two possible solutions are currently under investigation about the implementation of traceability between system architecture and other assurance-related information. These solutions are also presented.

The deliverable D3.6 “Prototype for architecture-driven assurance (c)” will be the evolution of this deliverable; in particular, D3.6 will document the progress about the implementation of the tooling framework supporting architecture-driven assurance.

# 1. Introduction

The AMASS approach focuses on the development and consolidation of an open and holistic assurance and certification framework for Cyber Physical Systems (CPS), which constitutes the evolution of the OPENCOS<sup>1</sup> and SafeCer<sup>2</sup> approaches towards an architecture-driven, multi-concern assurance, and seamlessly interoperable tool platform.

The AMASS tangible expected results are:

- a) The **AMASS Reference Tool Architecture**, which will extend the OPENCOS and SafeCer conceptual, modelling and methodological frameworks for architecture-driven and multi-concern assurance, as well as for further cross-domain and intra-domain reuse capabilities and seamless interoperability mechanisms (e.g. based on Open Services for Lifecycle Collaboration (OSLC)<sup>3</sup> specifications).
- b) The **AMASS Open Tool Platform**, which will correspond to a collaborative tool environment supporting CPS assurance and certification. This platform represents a concrete implementation of the AMASS Reference Tool Architecture, with a capability for evolution and adaptation, which will be released as an open technological solution by the AMASS project. AMASS openness is based on both standard OSLC Application Programming Interfaces (APIs) with external tools (e.g. engineering tools including V&V tools) and on open-source release of the AMASS building blocks.
- c) The **Open AMASS Community**, which will manage the project outcomes for maintenance, evolution and industrialization. The Open Community will be supported by a governance board, and by rules, policies, and quality models. This includes support for AMASS base tools (tool infrastructure for database and access management, among others) and extension tools (enriching AMASS functionality). As Eclipse Foundation is part of the AMASS consortium, the Polarsys/Eclipse community<sup>4</sup> is a strong candidate to host AMASS (See D7.3 and D7.5 for further details).

To achieve the AMASS results, as depicted in Figure 1, the multiple challenges and corresponding project scientific and technical objectives are addressed by different work-packages.

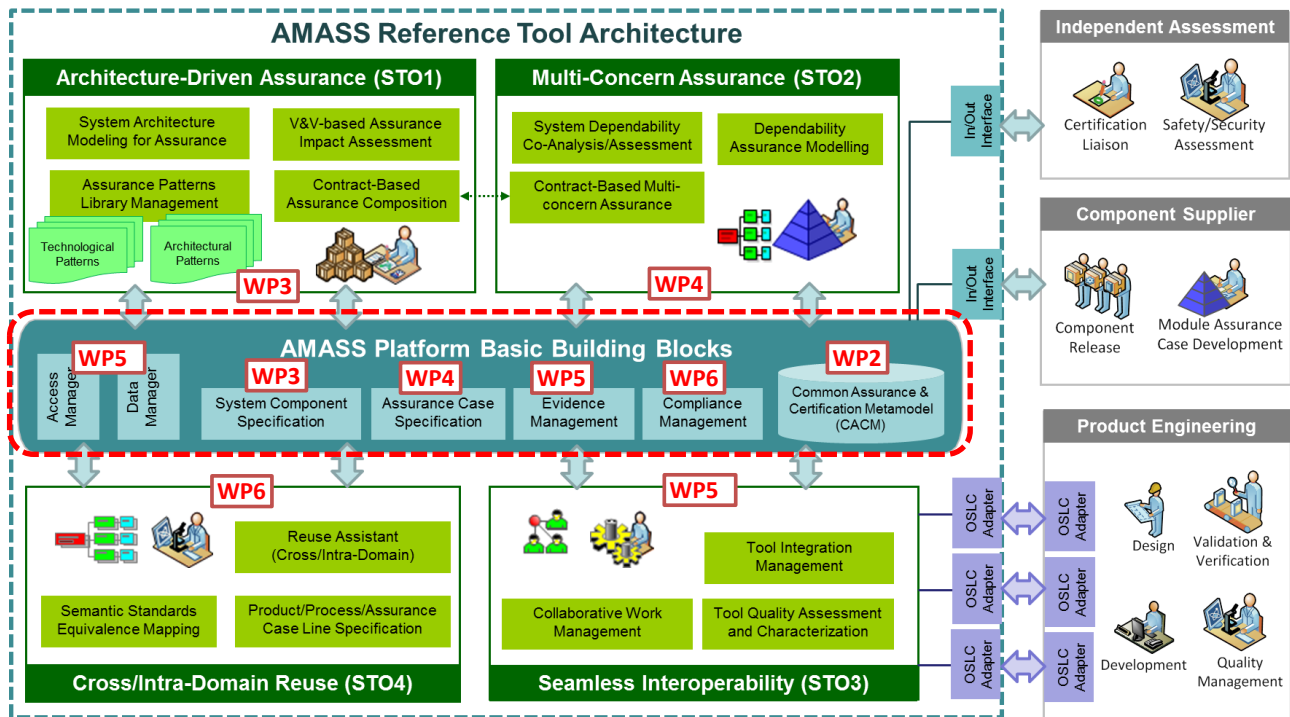
---

<sup>1</sup> [www.opencoss-project.eu](http://www.opencoss-project.eu)

<sup>2</sup> [www.safecer.eu](http://www.safecer.eu)

<sup>3</sup> <https://open-services.net>

<sup>4</sup> [www.polarsys.org](http://www.polarsys.org)



**Figure 1.** AMASS Building blocks

Since AMASS targets ambitious objectives related to architecture-driven assurance, multi-concern assurance, seamless interoperability support and cross-domain and intra domain assurance reuse, the AMASS Consortium has decided to follow an incremental approach by developing rapid and early prototypes.

The benefits of following a prototyping approach are:

- Better assessment of ideas by focusing on a few aspects of the solution.
- Ability to change critical decisions by using practical and industrial feedback (case studies).

AMASS has planned three prototyping iterations:

1. During the **first prototyping** iteration (Prototype Core), the AMASS Platform Basic Building Blocks, are aligned, merged and consolidated at Technology Readiness Level (TRL) 4 (technology validated in laboratory).
2. During the **second prototyping** iteration (Prototype P1), the single AMASS-specific Building Blocks will be developed and benchmarked at TRL 4.
3. Finally, at the **third prototyping** iteration (Prototype P2), all AMASS building blocks will be integrated in a comprehensive toolset operating at TRL 5 (technology validated in relevant environment).

Each of these iterations has the following three prototyping dimensions:

- **Conceptual/research development:** development of solutions from a conceptual perspective.
- **Tool development:** development of tools implementing conceptual solutions.



- **Case study development:** development of industrial case studies using the conceptual and tooling solutions.

As part of the Prototype P1, WP3 is responsible for driving the work resulting on architecture specification in order to design and implement the basic building block called “**System Component Specification**” (see Figure 1). This part of the AMASS platform manages component and contract-based design (see D3.1 [9] Section 3.1.1), also related to architecture-driven assurance.

This deliverable reports the **tool development** results of the “System Component Specification” basic building block. It presents in detail the pieces of functionality implemented in the AMASS platform tools, their software architecture, the technology used, and some source code references.

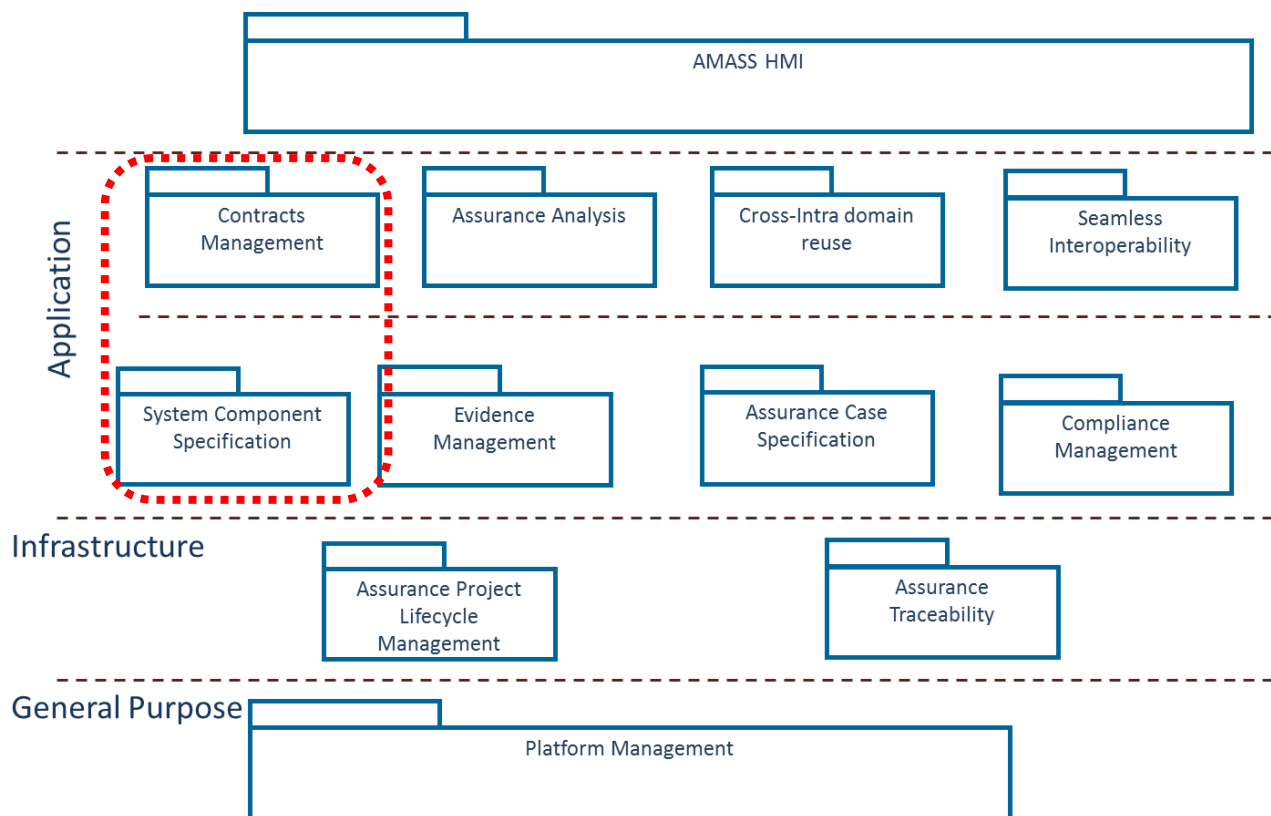
Other important parts of D3.5 document are:

- Installable AMASS Platform tools for the first prototype
- User Manuals and installation Instructions
- Source code description

## 2. Implemented Functionality

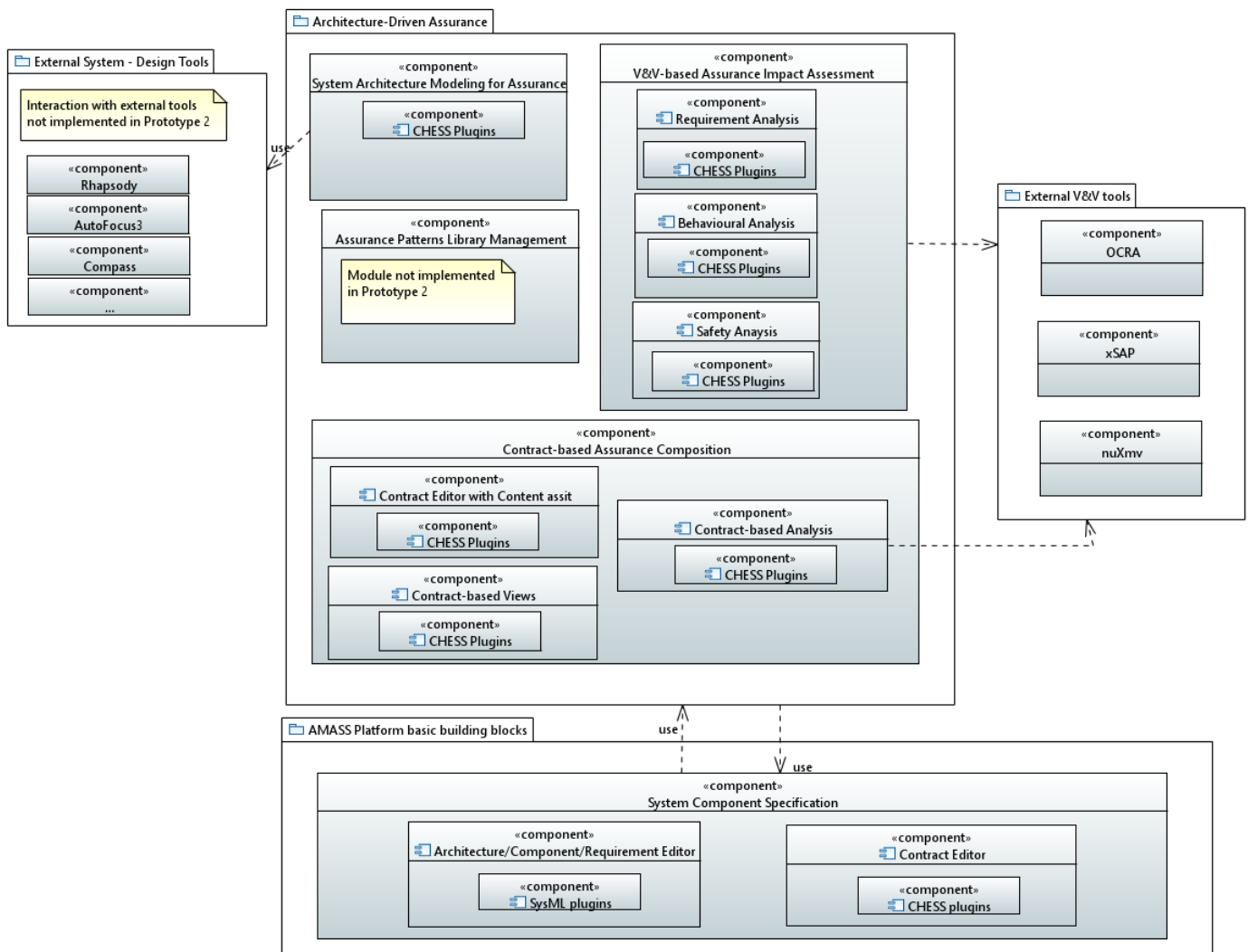
### 2.1 Scope

The scope for the second prototype iteration is the provision of modelling tools for system component specification, including a contract-based approach and the link with the assurance case specification. The main scope is highlighted with a red rectangle on Figure 2, which shows the general layered structure of the AMASS platform (from AMASS deliverable D2.3 [17]).



**Figure 2.** Layered structure of AMASS tool modules

Figure 3 illustrates the component decomposition of these tools based on the design specification documented in deliverable D3.2.



**Figure 3.** Description of main building blocks: System component specification and architecture driven assurance

## 2.2 Implemented Requirements

From the requirements point of view, this second prototype iteration focuses on a set of AMASS requirements as defined in the AMASS deliverable D2.1 “Business cases and high-level requirements” [16]. Each requirement together with the implementation done so far to implement the requirement is shortly outlined in the following sections.

### 2.2.1 System Component Specification

#### 2.2.1.1 System Architecture Edition

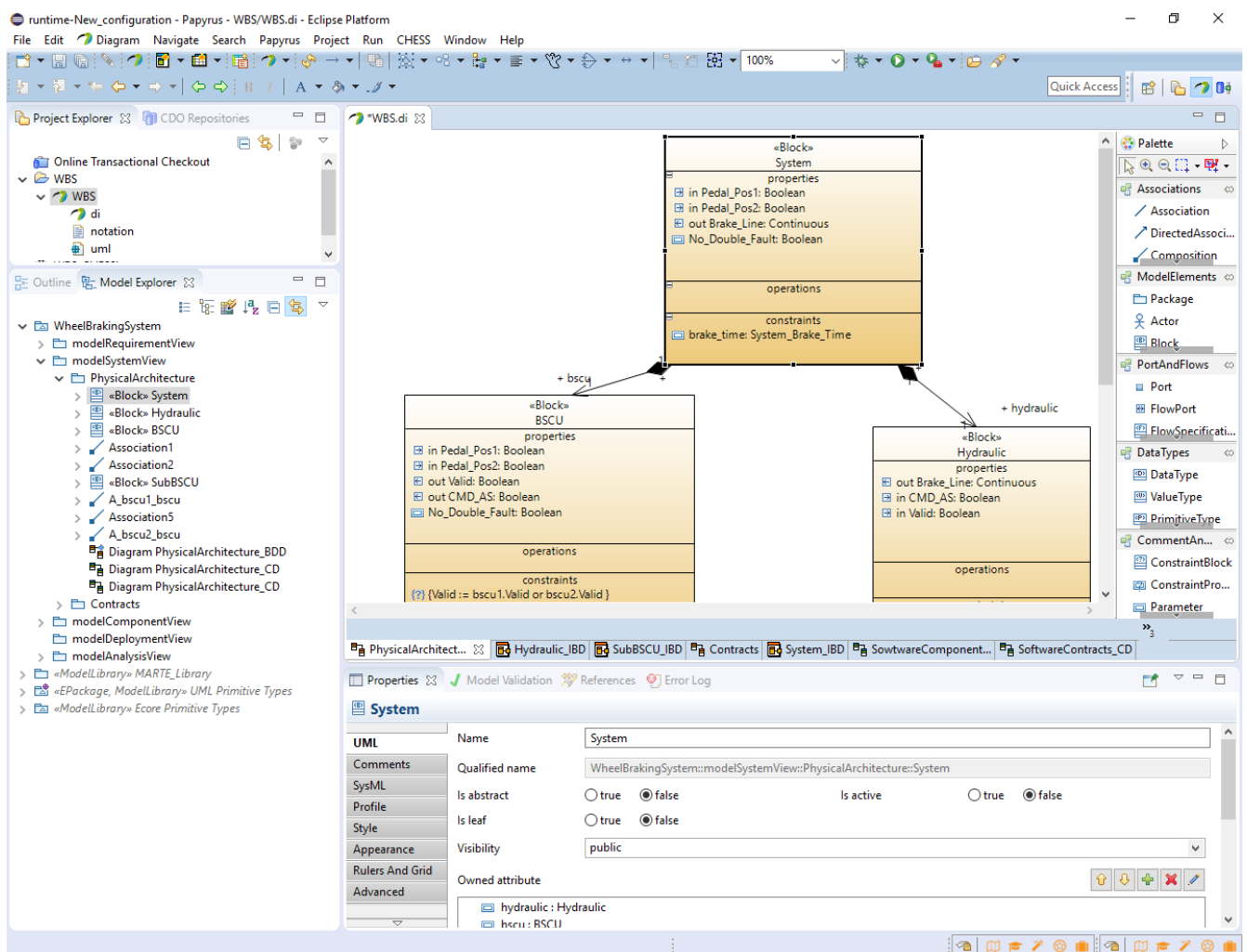
System architecture specification is supported by the Papyrus UML/SysML editor [5]. The selection of UML/SysML has been driven by the wide adoption of these modelling languages in the industry in different domains. Then, the selection of Papyrus UML/SysML editor has been driven by the fact that Papyrus is one of the most appreciated solid open source tools available in the industry for professional modelling; in particular, recently the Papyrus Industry Consortium has been created to support a model-based engineering platform based on the domain specific and modelling capabilities of the Eclipse Papyrus family

of products. It is worth noting that Papyrus has also integration facilities with other tools, such as the commercial IBM UML Rhapsody tool; in addition, it supports the XMI OMG standard [8] for the interchange of UML models between UML tools.

Through Papyrus editor (see Figure 4), SysML Blocks and UML Components can be used to model the architectural entities as required by the AMASS component meta-model definition (see D2.3 [6]). Decomposition of block/components into sub-blocks/sub-components can be modelled by using internal block diagrams or composite structure diagrams. Both Papyrus Editor and other AMASS components are under the same Open Source license, which supports the reuse of these previous results into the AMASS platform.

Information about the functional behaviour of a given component/block can be provided through state machine diagrams.

System architecture UML/SysML models and diagrams are stored in individual files in the Eclipse workspace.



**Figure 4.** Papyrus editor

The Papyrus UML editor supports the definition and application of UML profiles. In AMASS, Papyrus tool is used together with the CHES profile extension [3]; in particular CHES is used here as extension of the UML and SysML modelling languages to allow the modelling of contracts, as explained in the following sections, according to the AMASS component meta-model needs (see D2.3 [6]).

CHESS also provides extension to the Papyrus tool, for instance by adding dedicated diagram palettes to facilitate the creation of the CHESS entities, or by adding a dedicated property tabs view for editing CHESS entities properties (see Section 4).

It is worth noting that the CHESS profile also provides other modelling capabilities, such as the dependability profile [10] for failure modelling and specific support for timing properties (see Section 4 about CHESS features). Moreover, CHESS provides a methodology for the design, verification and implementation of CPS SW systems [1]. The aforementioned features are not currently part of the AMASS basic building blocks; their possible role and integration in AMASS will be studied during the project. The CHESS profile follows the same licenses approach as Papyrus and other AMASS components, which supports the easy integration of the developments from the intellectual property perspective.

#### 2.2.1.2 Formalize Requirements with Formal Properties

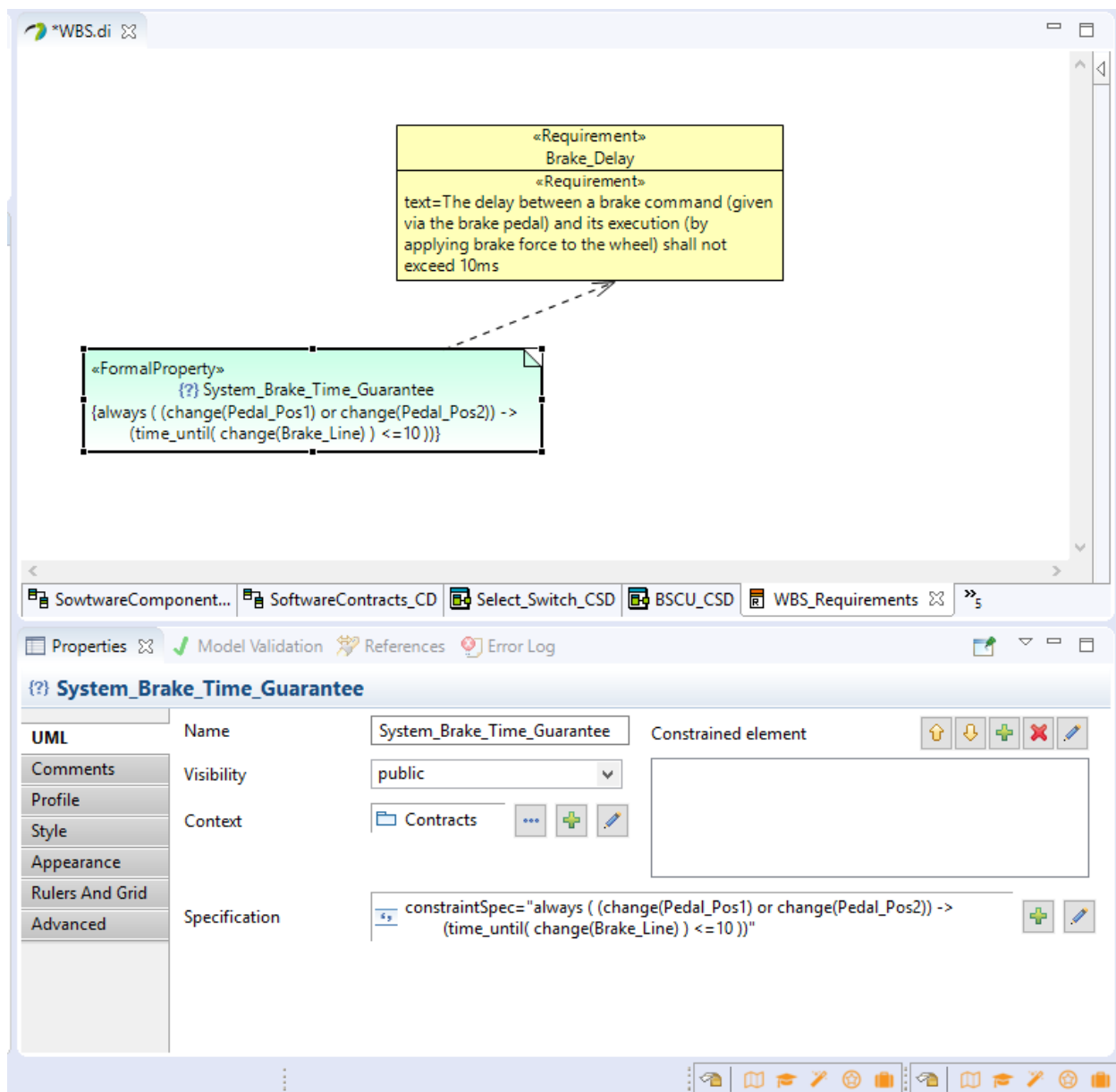
Requirements can be modelled in Papyrus using the SysML profile; indeed, SysML comes with the dedicated *Requirement* stereotype (see Figure 5) which can be managed through Requirement Diagrams. The availability of system requirements represented in the model allows to model their traceability to the different parts of the system model. In particular, by using the SysML profile, requirements can be traced to the entities of the architecture, by using the *Satisfy* link defined by SysML. In this way, model-driven support can be enabled to support requirement traceability (see e.g. [7]), which is an important quality factor to be guaranteed while building systems.

In AMASS, a formal property represents a distinct entity which is used to provide a formal description of a given system requirement, the latter usually described using informal textual language.

To model formal properties, CHESS profile defines a class called **FormalProperty** as an extension of UML Constraint (see Figure 5). A **FormalProperty** can be created first in the model and then it has to be linked to the requirement that it formalizes; the SysML trace link can be created in the SysML Requirement diagram or through the tabular editor provided by Papyrus<sup>5</sup>. Then the formal description of the requirement has to be provided by using the *specification* attribute coming with the **FormalProperty** entity.

---

<sup>5</sup> [https://wiki.eclipse.org/Papyrus\\_User\\_Guide/Table\\_Documentation](https://wiki.eclipse.org/Papyrus_User_Guide/Table_Documentation)



**Figure 5.** Modelling FormalProperty

It is worth noting here that the CHESSE profile does not force the usage of a particular formal language; the choice of the formal language to be adopted for the formalization of requirements is made by the modeller, typically according to the adopted process/methodology. CHESSE currently supports integration with the OCRA contract specification language<sup>6</sup>; in particular, through the CHESSE Contract plugins explained in Section 4 it is possible to verify formal properties specification with respect to OCRA syntax.

### 2.2.1.3 Semi-Formal Contract Definition

As users might not be familiar with formal expressions to define contracts, we offer the possibility to use a set of patterns with which many assertions can be formulated. We thought about different ways to

<sup>6</sup> <https://ocra.fbk.eu>

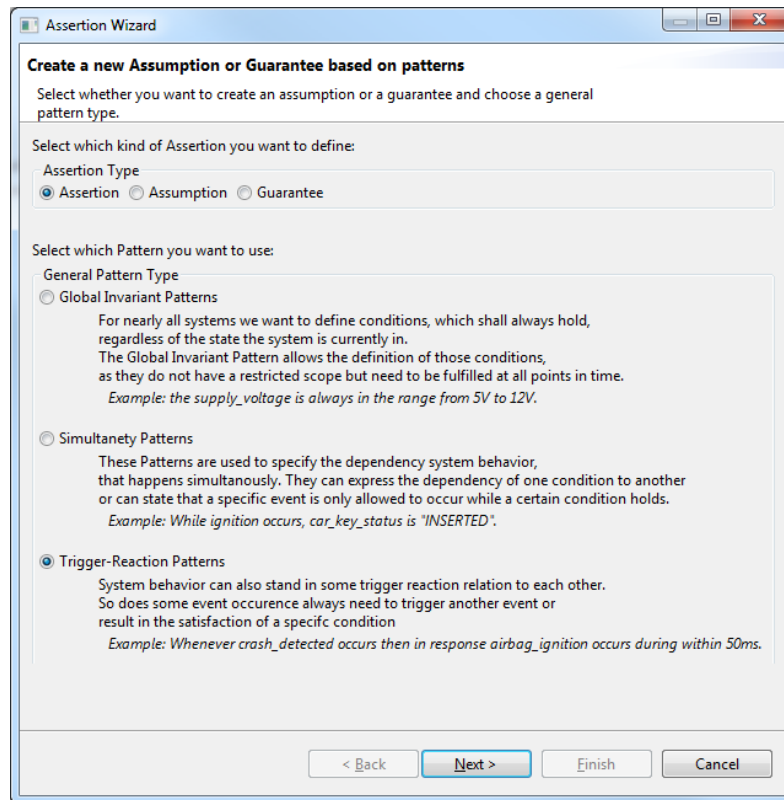
support the user at writing template assertions and ended up providing two ways, which allow the user to specify assertions by using our patterns more easily.

The first option is to use an Assertion Wizard, which guides him or her through a preselected set of available pattern constructs together with examples. If the user has decided on a pattern, he or she just needs to adjust minor details such as variable names or conditional relations until the assertion is completed. The other option for the user is to write assertions in a text editor, which features syntax checks and auto-completion. In the following paragraphs we explain both concepts in detail.

### **Assertion Wizard**

As applying a template language can be quite difficult without any guidelines, we decided to implement a wizard that guides the user through the process of choosing and filling out an appropriate pattern structure for their statement. The first page of the wizard shows the user the three main pattern types of our template language: Global Invariant Pattern, Simultaneity Pattern, and Trigger-Reaction Pattern (see Figure 6). We have added a short description and an example for each one so that it is easier for the user to make a decision.

After selecting the main pattern type, several possible pattern instances of the type will be presented to the user. Each of them features an example to demonstrate a possible application (see Figure 7). 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. That way the user can only choose and use model elements that are in scope (see Figure 8). 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. We also provide a set of time units the user can choose from when specifying timed behavior. Only if no non-terminals remain in the pattern instance and all terminals are replaced by actual interface names, values, units, etc., the assertion can be assigned on a selected component. Otherwise, the wizard will hint the user at the remaining non-terminals or terminals.



**Create a new Assumption or Guarantee based on patterns**

Select whether you want to create an assumption or a guarantee and choose a general pattern type.

Select which kind of Assertion you want to define:

Assertion Type

☒ Assertion ☐ Assumption ☐ Guarantee

Select which Pattern you want to use:

General Pattern Type

☐ Global Invariant Patterns

For nearly all systems we want to define conditions, which shall always hold, regardless of the state the system is currently in.  
The Global Invariant Pattern allows the definition of those conditions, as they do not have a restricted scope but need to be fulfilled at all points in time.  
*Example: the supply\_voltage is always in the range from 5V to 12V.*

☐ Simultaneity Patterns

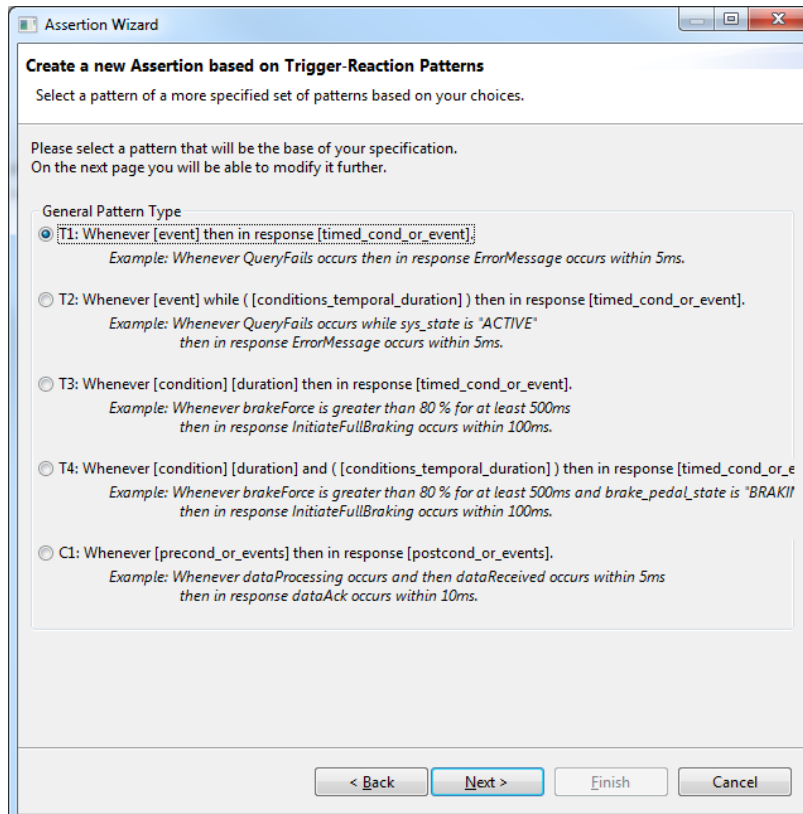
These Patterns are used to specify the dependency system behavior, that happens simultaneously. They can express the dependency of one condition to another or can state that a specific event is only allowed to occur while a certain condition holds.  
*Example: While ignition occurs, car\_key\_status is "INSERTED".*

☒ Trigger-Reaction Patterns

System behavior can also stand in some trigger reaction relation to each other. So does some event occurrence always need to trigger another event or result in the satisfaction of a specific condition  
*Example: Whenever crash\_detected occurs then in response airbag\_ignition occurs during within 50ms.*

< Back Next > Finish Cancel

**Figure 6.** First step in the Assertion-Wizard: Select a General Pattern Type to formulate an assertion. Each selection features a short description and example to offer the user an easy decision.



**Create a new Assertion based on Trigger-Reaction Patterns**

Select a pattern of a more specified set of patterns based on your choices.

Please select a pattern that will be the base of your specification.  
On the next page you will be able to modify it further.

General Pattern Type

☒ T1: Whenever [event] then in response [timed\_cond\_or\_event].  
*Example: Whenever QueryFails occurs then in response ErrorMessage occurs within 5ms.*

☐ T2: Whenever [event] while ( [conditions\_temporal\_duration] ) then in response [timed\_cond\_or\_event].  
*Example: Whenever QueryFails occurs while sys\_state is "ACTIVE" then in response ErrorMessage occurs within 5ms.*

☐ T3: Whenever [condition] [duration] then in response [timed\_cond\_or\_event].  
*Example: Whenever brakeForce is greater than 80 % for at least 500ms then in response InitiateFullBraking occurs within 100ms.*

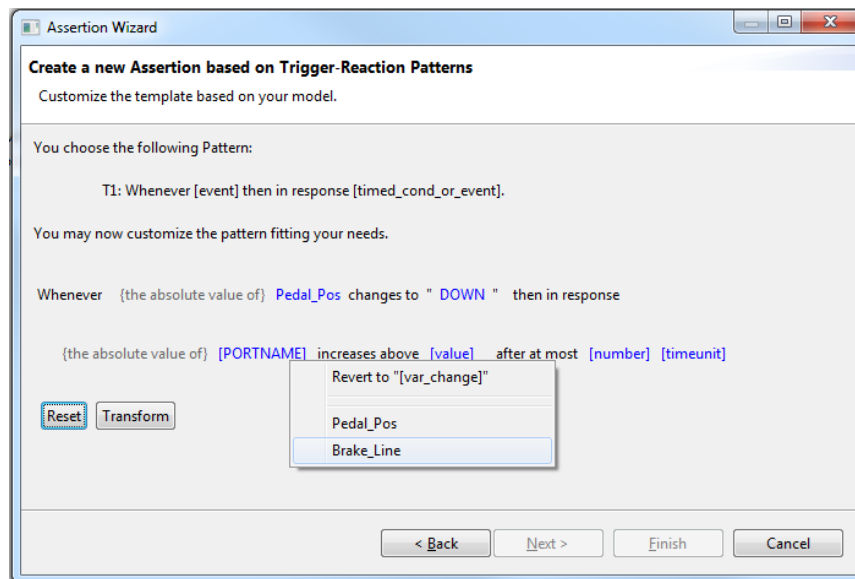
☐ T4: Whenever [condition] [duration] and ( [conditions\_temporal\_duration] ) then in response [timed\_cond\_or\_event].  
*Example: Whenever brakeForce is greater than 80 % for at least 500ms and brake\_pedal\_state is "BRAKE" then in response InitiateFullBraking occurs within 100ms.*

☐ C1: Whenever [precond\_or\_events] then in response [postcond\_or\_events].  
*Example: Whenever dataProcessing occurs and then dataReceived occurs within 5ms then in response dataAck occurs within 10ms.*

< Back Next > Finish Cancel

**Figure 7.** Second step in the Assertion-Wizard: Choose a pattern instantiation of the previously selected general pattern type



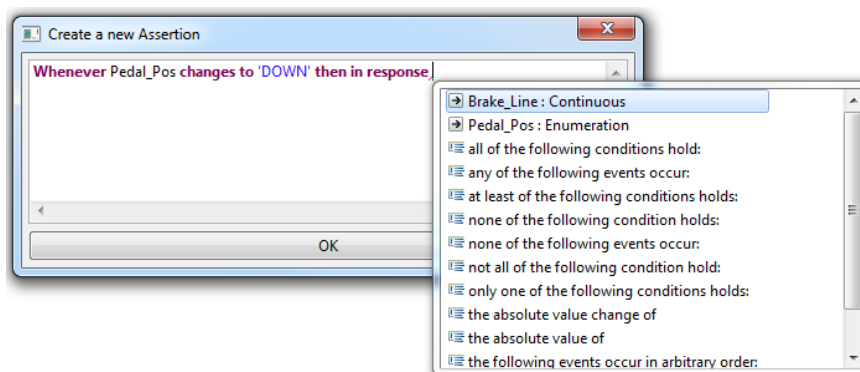


**Figure 8.** Last step of the Assertion-Wizard: Refine the pattern instance with names of available model elements. Only element names which are valid for the corresponding placeholder are allowed to be used

## 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. With the Assertion Editor, we allow the user to directly type in the desired assertion. As writing valid assertions free-hand can be difficult and error-prone, we offer support with an online syntax check and suggestions for auto-completion of the statement, already known from various programming IDEs. Figure 9 shows the Assertion Editor suggesting valid possibilities to continue the current statement.

We chose *Xtext* as the technology to base our text editor on. That allowed us to easily implement the editor merely only by providing the BNF in the *Xtext* grammar format and slightly adjusting the auto-completion suggestions. The rest was done automatically by the code generation feature of *Xtext*. Another important reason why we chose *Xtext* is because it features methods to automatically translate expressions from one language to another. This can be used later to translate our template expressions into a formal language expression.



**Figure 9.** Pattern-suggestion feature of the Assertion Editor

The translation of pattern expressions to LTL or similar temporal logics is planned to be supported in the final prototype.

#### 2.2.1.4 Structure Properties into Contracts

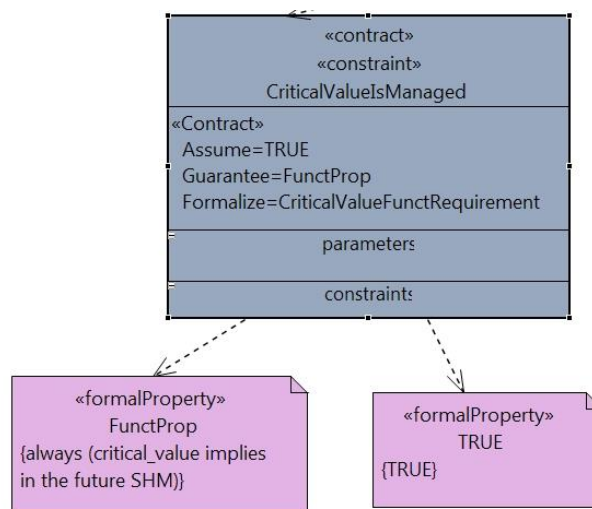
CHES profile supports the modelling of weak and strong contracts to support contract-based design (the reader can refer to AMASS D3.1 [9] for an introduction to weak and strong contracts and contract-based design).

*Contracts* are available in the CHES profile as a special kind of classifiers (i.e. an entity used to describe instance-level entities of the same kind). Contracts can be created in UML class, component, or SysML block diagrams. A Contract comes with two attributes representing the assumption and guarantee formal properties.

By using the CHES Papyrus extension, when a Contract is created in the model, the tool automatically creates a pair of empty *FormalProperties*, the latter playing the role of assumption and guarantee of the Contract itself.

Alternatively, a given *FormalProperty* available in the model before the creation of the Contract can later be assigned to the Contract itself, as assumption or guarantee.

Figure 10 below shows an example of *Contract* and *FormalProperty* modelling; the figure shows the Assume and Guarantee attributes owned by the Contract, which in the example are bounded to the represented *FormalProperty*. A link between the Contract and the *FormalProperty* is also depicted.



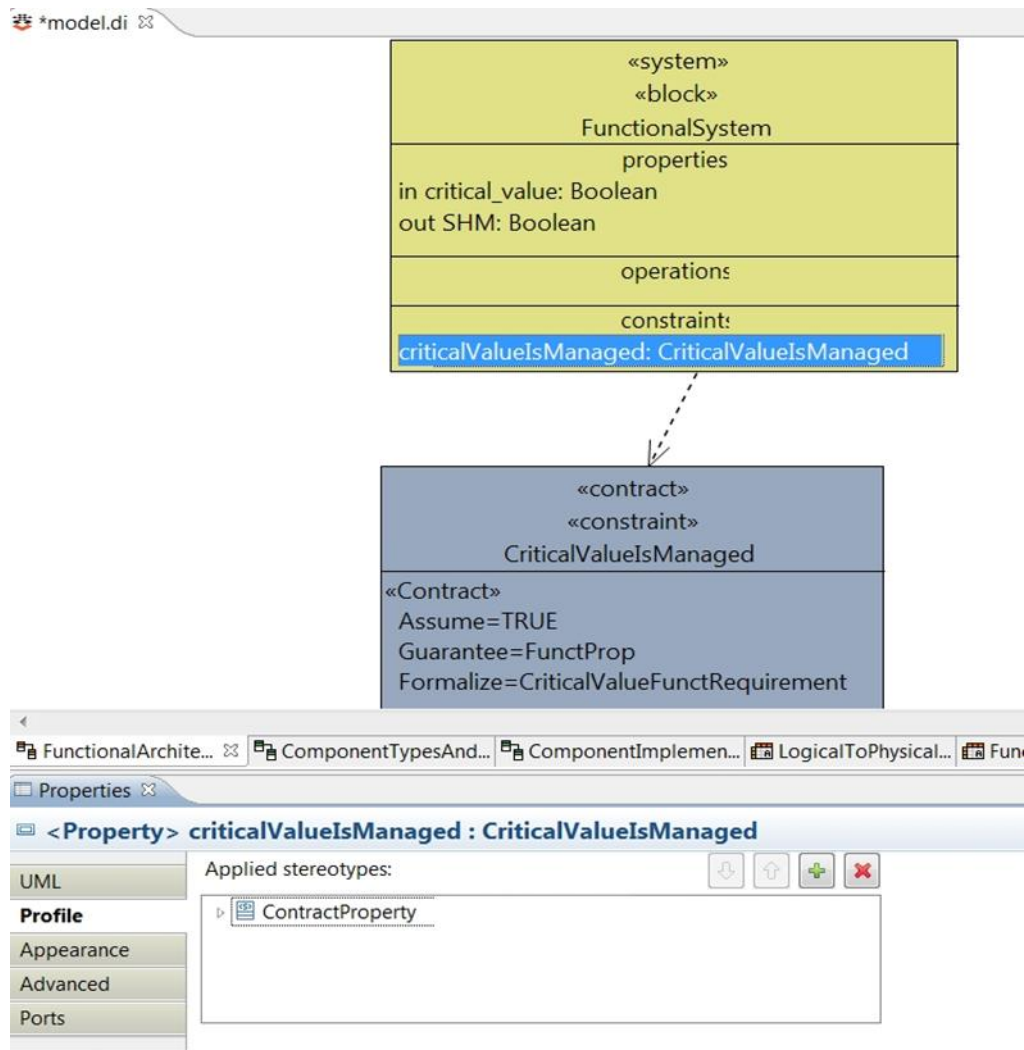
**Figure 10.** Contract and FormalProperty example

#### 2.2.1.5 Assign Contract to Component

In CHES, a Contract is assigned to a given UML Component/SysML Block by instantiating the Contract itself in the Component/Block. In particular, a *ContractProperty* attribute has to be created for the Component/Block first and then the *ContractProperty* type must be set to the particular *Contract*. Therefore, in CHES one important piece of information related to contract-based design is modelled through the *contract instance*, which represents a *Contract* associated to a Component/Block.

*ContractProperty* has also an attribute that allows specifying if the associated Contract has to be applied to the Component/Block according to the weak or strong semantics<sup>7</sup> [9].

As example, Figure 11 shows the *criticalValuesManaged* *ContractProperty* owned by the *FunctionalSystem* Block (the *ContractProperty* is shown in the diagram in the *Constraint* compartment of the Block). The *criticalValuesManaged* property is typed as *CriticalValuesManaged* Contract (*criticalValuesManaged:CriticalValuesManaged*), the latter is also represented in the diagram. The *criticalValuesManaged* property represents the association of the *CriticalValuesManaged* Contract to the *FunctionalSystem* Block.

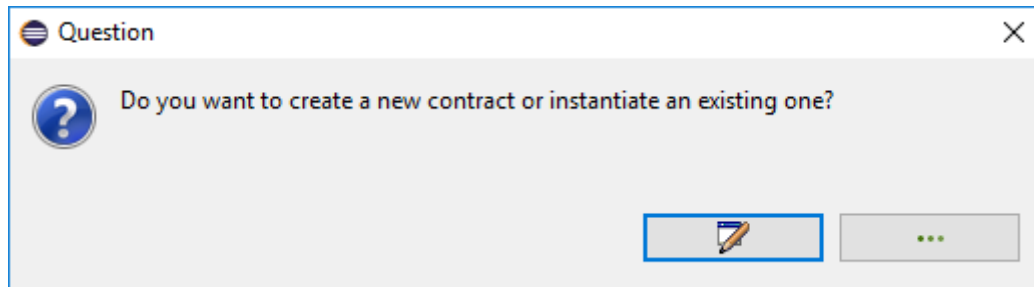


**Figure 11.** Assign Contract to Component

This allows to potentially reusing the same *Contract* in different contexts/systems (as analogous to the practice of sharing requirements across projects, i.e. software/system requirements reuse).

<sup>7</sup> As discussed [9], while strong assumptions define compatible environments in which the component/block can be used, weak assumptions define specific contexts where additional information is available. Hence, a component/block should never be used in a context where some strong assumptions are violated, but if some weak assumptions do not hold, it just means that the corresponding guarantees cannot be relied on.

The second AMASS prototype (Prototype P1) enables also the possibility to automatically create a contract when a *ContractProperty* is created, see Figure 12. In this case, the association contract-component is 1 to 1. The first advantage is that, during the editing of the contract, the content assist supports the user suggesting which are the ports and the attributes name of the component. The second advantage is that, the operation of contract definition is improved in terms of time spent.



**Figure 12.** After the creation of a *ContractProperty*, a Popup appears to decide whether a new contract has to be created or an existing one has to be instantiated

#### 2.2.1.6 Contract Refinement

The CHES profile allows to model contracts refinement/decomposition along the refinement/decomposition of the architectural entities, the latter provided through UML composite structural diagrams or SysML block definition diagrams. In particular, contract instances play a key role during the refinement specification. Indeed, contracts refinement is modelled for contract instances, not for the Contracts entities; this is because the same Contract can be reused in several contexts (i.e. instantiated in several Components/Blocks), and for each context the refinement of the same Contract could be different. So through the CHES profile it is possible to model how a given contract instance is refined by a set of other contract instances.

In practice, given a contract instance  $C$  assigned to a component  $A$ , and given the decomposition of  $A$  into subcomponents  $(A_1, \dots, A_n)$  and the contracts instances assigned to each subcomponent  $(C_{1<1..k>}, \dots, C_{n<1..j>})$ , it is possible to model how  $C$  is decomposed by (a subset of)  $(C_{1<1..k>}, \dots, C_{n<1..j>})$ .

#### 2.2.1.7 Modelling Failure Behavior

For the modelling of failure and security behavior (e.g. an accidental / malicious fault occurring at a given component's input/output port) no specific implementation has been currently identified as official part of the AMASS building block; this is currently an ongoing task in the AMASS project.

Existing support for failure behavior modelling is available from state of the art projects and modelling tools, like the UML/MARTE dependability profile coming with the CHES modelling language [3] (see e.g. section 2.2.3.4).

The investigation of the extension of the CHES dependability profile to support definition of security threats is currently an ongoing task.

### 2.2.2 System Architecture Modelling for Assurance

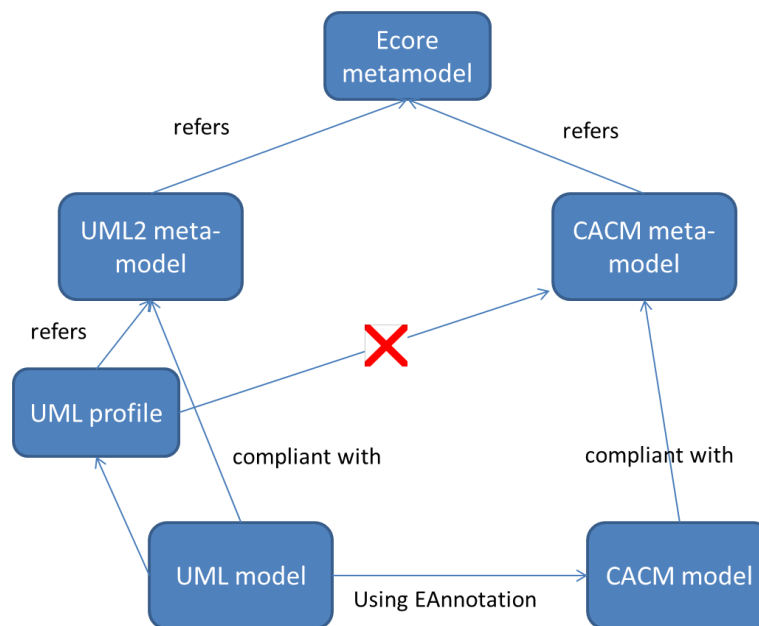
#### 2.2.2.1 Link Architecture-Related Entity to Assurance Case Information

The allowed links between architectural entities and the other parts of the CACM AMASS meta-model (about management of the assurance project as indicated in Figure 2) are currently described in the CONCERTO deliverable D2.2 [4].

As explained in the previous sections, the AMASS component model has been made available as Eclipse plugin as UML/SysML language extended with the CHES profile for contracts, while the other parts of the CACM (argumentation, evidence, compliance management) are currently implemented as Ecore meta-model<sup>8</sup> (not as UML profile).

Within the UML profile definition, it is not possible to refer to an Ecore entity which is not related to the UML language, so the aforementioned links (e.g. from a CHES-Contract to an argumentation-Claim) cannot be expressed through the CHES profile; the links have to be managed with some additional modelling support, as explained below in the text.

Indeed, in the context of Task 3.3 we are currently investigating the best approach to allow the modelling of the links between the component model entities and the other parts of the CACM. One solution could be to use the **EAnnotation** mechanism available in Ecore: EAnnotation allows to attach extra information to any object available in an Ecore model. In our case, EAnnotation could be created for a UML model entity (for instance a Contract)<sup>9</sup>; then EAnnotation could be used to refer to an entity of the CACM defined in some external (to the UML) model (as a Claim in an argumentation model). Figure 13 gives a picture of what has been stated above (CACM model in the figure has to be intended as the model for argumentation, evidence, and compliance management).



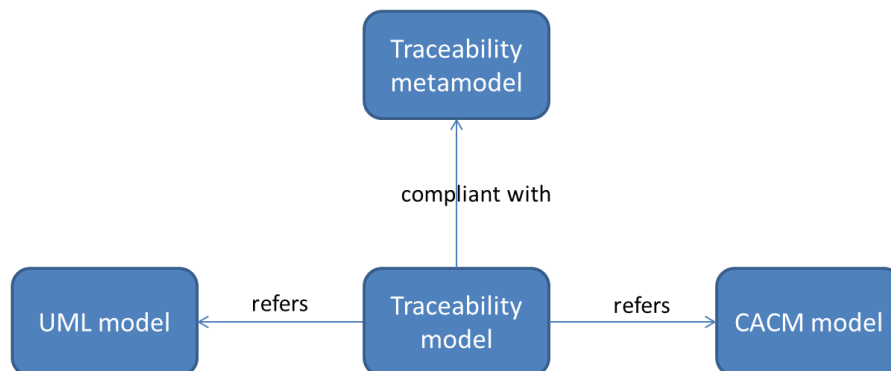
**Figure 13.** Links through EAnnotation

Another possibility is to use a traceability support based upon a dedicated traceability meta-model (see Figure 14). In this way, a link would be created according to the traceability meta-model; each link would own a reference to the UML model entity and a reference to the CACM model entity to be associated. We

<sup>8</sup> Ecore is a model provided by the Eclipse EMF project (<https://www.eclipse.org/modeling/emf/>); Ecore can be used to model the structure of a given domain of data models. Typically, Ecore is referenced as meta-meta-model; the structure of a given domain of data models is referenced as meta-model, where a model is a concrete instance of this meta-model.

<sup>9</sup> It is worth noting that EAnnotation can be added to UML model entities because UML models in Eclipse are implemented as Ecore models.

are also currently evaluating the applicability of this approach to the needs of traceability addressed in WP5.



**Figure 14.** Links through traceability meta-model

It is worth noting that both the aforementioned solutions can also be used to model links between architectural entities and process related information, being the latter defined according to WP6 results (at the time of writing, the specification of the links between architectural entities and process related information has not yet been fully formalized, it will be defined in the context of task T3.2).

We are still investigating the benefits and limits of each of the aforementioned solutions, in particular by using the CAPRA tool which offers an implementation for the solution depicted in Figure 14; CAPRA is also under extension in the context of WP5. What is worth noting is that the usage of a dedicated traceability meta-model could be made generic in order to support traceability between assurance case information and architecture-related entities specified with other non-UML modelling languages. For instance, by assuming the availability of an Architecture Analysis and Design Language (AADL)<sup>10</sup> editor in Eclipse, the same traceability model could be used to create links between AADL entities and argumentation/evidence entities available in the CACM model.

## 2.2.3 V&V-based Assurance Impact Assessment

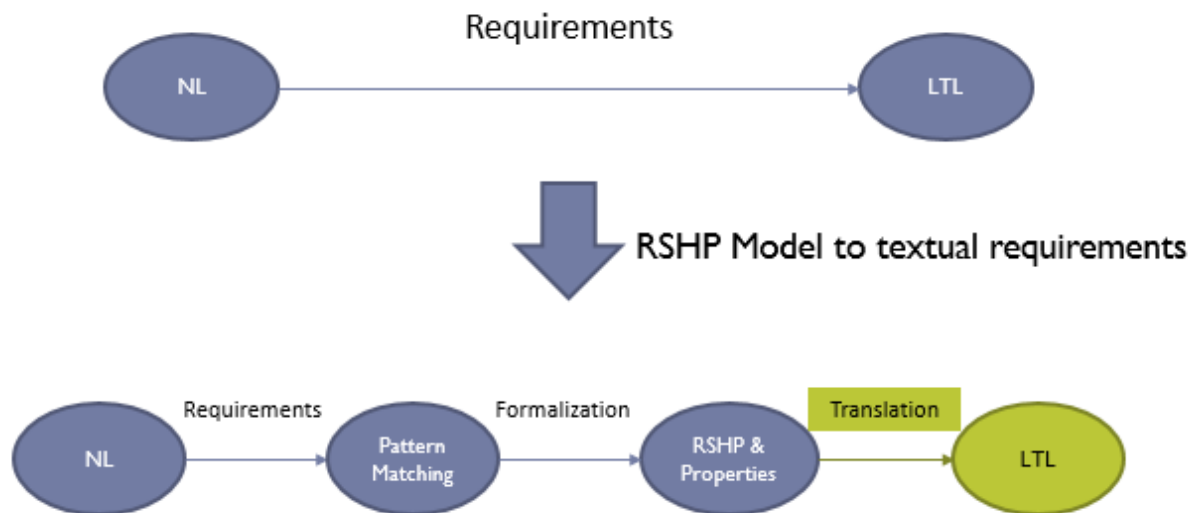
### 2.2.3.1 Requirements Formalization for analysis of Temporal Realizability – Requirement Quality Analyzer approach

Requirement Quality Analyzer (RQA) tools provide the possibility to create custom-coded metrics made by a user. Using this feature, a metric has been created for the detection and evaluation of the temporal elements in a set of requirements.

The implementation consists of a NLP software mechanism applied to textual requirements in order to make a quality assessment, in terms of temporal consistency. The quality assessment is an automatic translation from requirements written in Natural Language to LTL, using the RSHP Model applied to textual requirements.

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, with the objectives of extracting relevant information from them (concepts, relationships and properties), storing and reusing them thanks to RSHP.

<sup>10</sup> <http://www.aadl.info/aadl/currentsite>

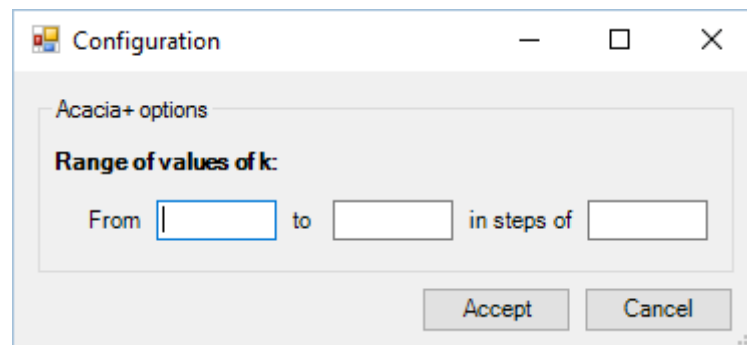


**Figure 15.** Automatic translation general diagram - From NL to LTL

The resultant LTL is being processed by Acacia+, a open source tool with algorithms to check the reliability, synthehis 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).

According to custom-coded development rules, every metric shall follow three steps: configuration, evaluation and results. For the described metric, these steps have been implemented this way:

**Configuration:** Acacia+ tool requests different parameters; however, only one of them is been configurable by the user. Precisely, it is the K number of accepted states (through lower and upper limits and an incremental coefficient), which restricts the number of iterations in case the problem is hard to be computed.



**Figure 16.** Configuration window to setup the K number



On the other hand, the metrics require the pattern or pattern group needed for formalization.

**Evaluation:** the objective is to provide the LTL specification to Acacia+. In this sense, it will:

1. Analyse the entire requirement specification.
2. Check which of the requirements matches the patterns selected in the configuration step.
3. Study the results of the formalization of the requirements.
4. Translate the NL to LTL using the results of the formalization.
5. Execute the resultant LTL specification in Acacia+, which will return at least a winning strategy (according to K number) or a counter-strategy in case the formula is not reliable.

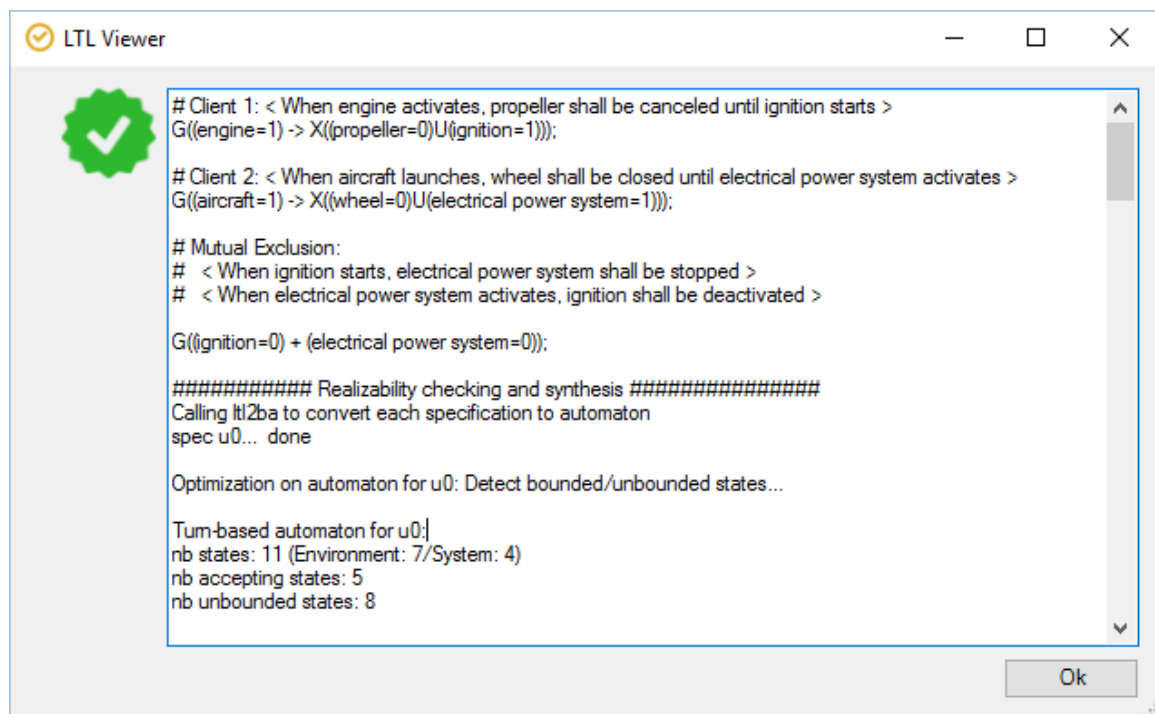


**Results:** there are only two possible final states: temporally reliable specification or not (with the selected configuration). Thus, for the reliable case, the result is high quality (3 stars) or low quality (1 star) for the case it is not possible to demonstrate the reliable. Additionally, there is a result window showing the final translation and Acacia+ result.

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.

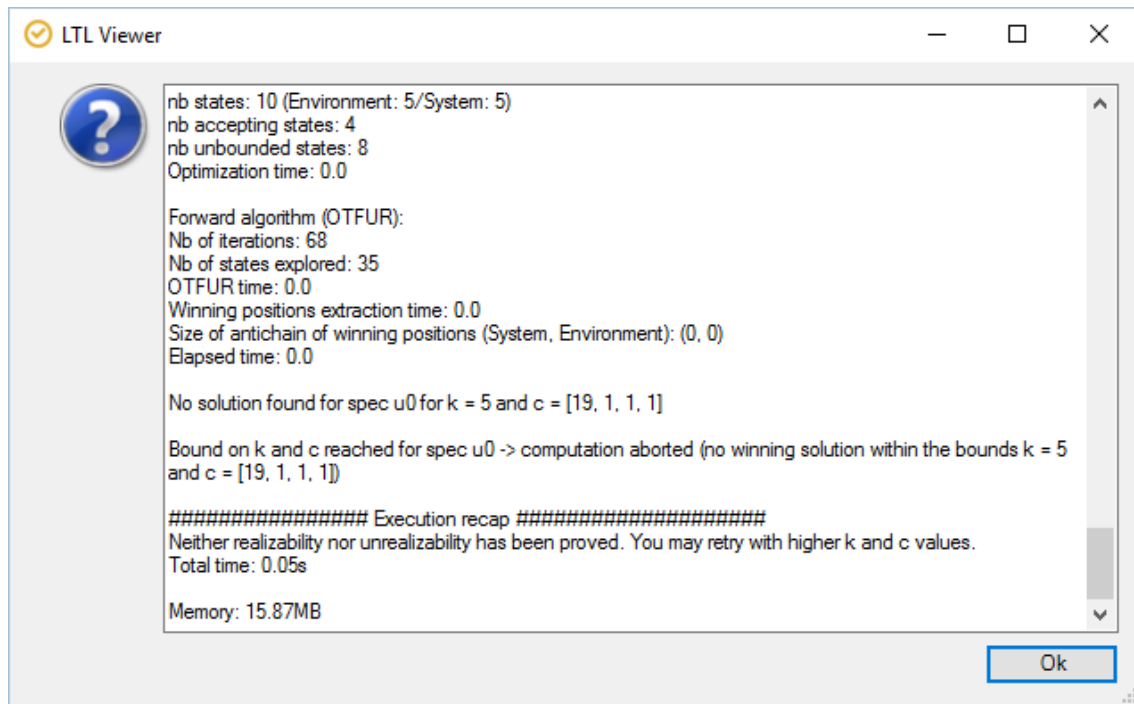
**Figure 17.** Metric results visualization

It is remarkably that, for the cases the problem is not reliable, RQA will indicate to increment the K number in the configuration step, thanks to Acacia+.



**Figure 18.** Result window for reliable experiment





**Figure 19.** Window result for not reliable experiment

### 2.2.3.2 Metrics

This section describes the implementation in the RQA tool related to the metrics presented in the section 2.4.4 Metrics of the D3.2 deliverable [14].

#### 2.2.3.2.1 Metrics for requirements

Following describes the metrics implemented in the RQA tool related with requirements, and presented in the section 2.4.4.1 Metrics of the D3.2 deliverable.

##### 2.2.3.2.1.1 Correctness metrics

The RQA tool implements the follow correctness metrics based on the System Knowledge Base (see Figure 20).

#### In-System Conceptual Model Nouns (SCM Nouns)

The RQA tool allows to create the metric “SCM nouns”, to check if each term in the requirements belongs to the SCM view or any semantic.

#### Out-of-System Conceptual Model Nouns (Out-of-SCM Nouns)

The user can create the metric “Out-of-SCM Nouns”. This metric checks if each term does not belong to any SCM view or any semantic cluster.

#### In-Semantic Clusters Nouns (SCC Nouns)

The RQA tool allows to create the metric “SCC Nouns”. The metric checks if each term of the requirements belongs to one or more semantic clusters.

#### Out-of-Semantic Clusters Nouns (Out-of-SCC Nouns)

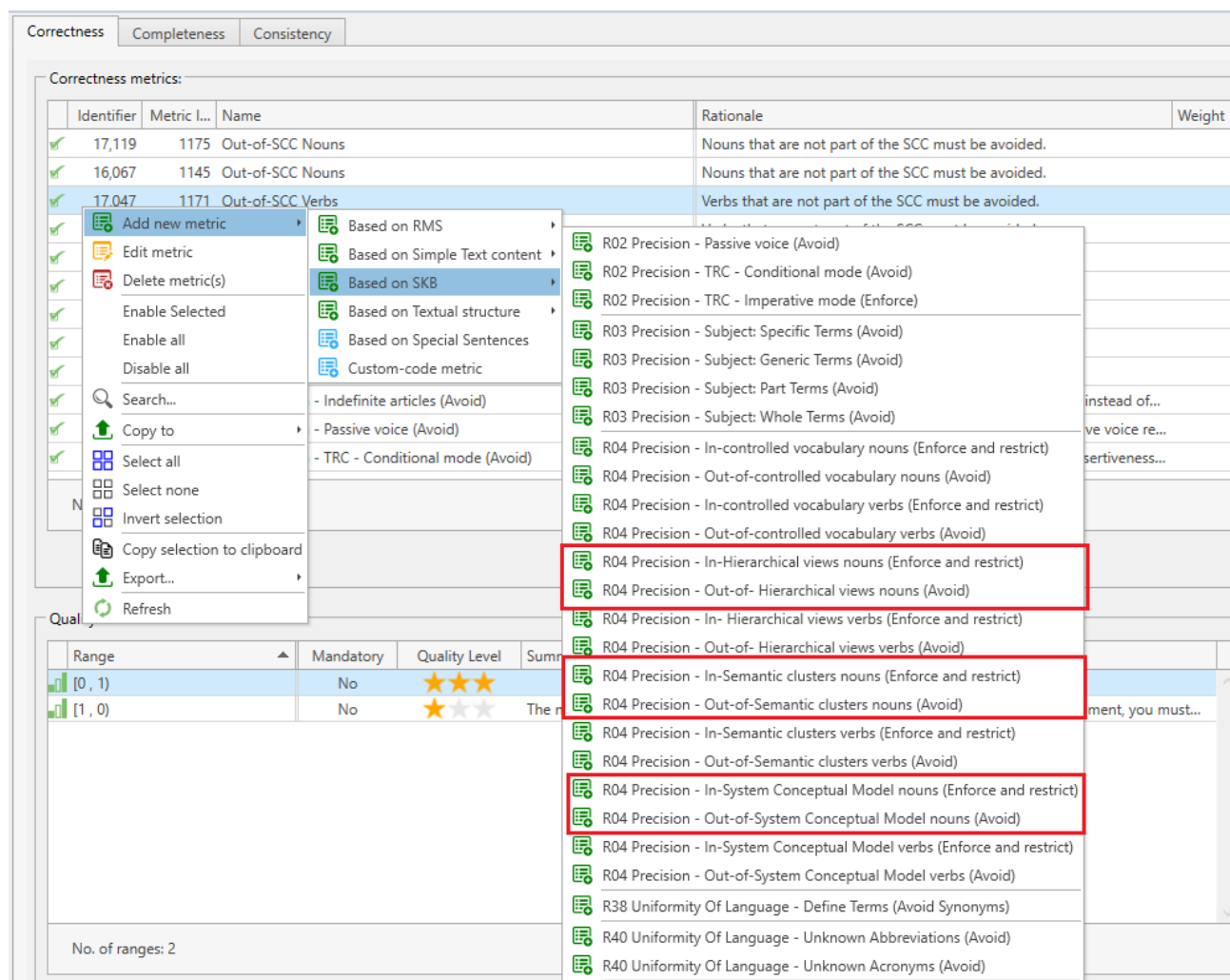
The RQA tool implements the metric “Out-of-SCC Nouns”. The metric checks if each term does not belong to any semantic cluster.

### In-Hierarchical Views Nouns (SCV Nouns)

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

### Out-of-Hierarchical Views Nouns (Out-of-SCV Nouns)

The RQA tool allows to create the metric “Out-of-SCV Nouns” to check that each term in the requirement does not belong to one or more SCM view.



The screenshot displays the RQA tool's 'Correctness' tab. A table lists metrics with columns for Identifier, Metric I..., Name, Rationale, and Weight. The 'Out-of-SCC Verbs' row is selected, and a context menu is open. The menu includes options like 'Add new metric', 'Edit metric', 'Delete metric(s)', and a list of specific metrics. A subset of these metrics is highlighted with red boxes, indicating the focus of the figure.

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 Options:

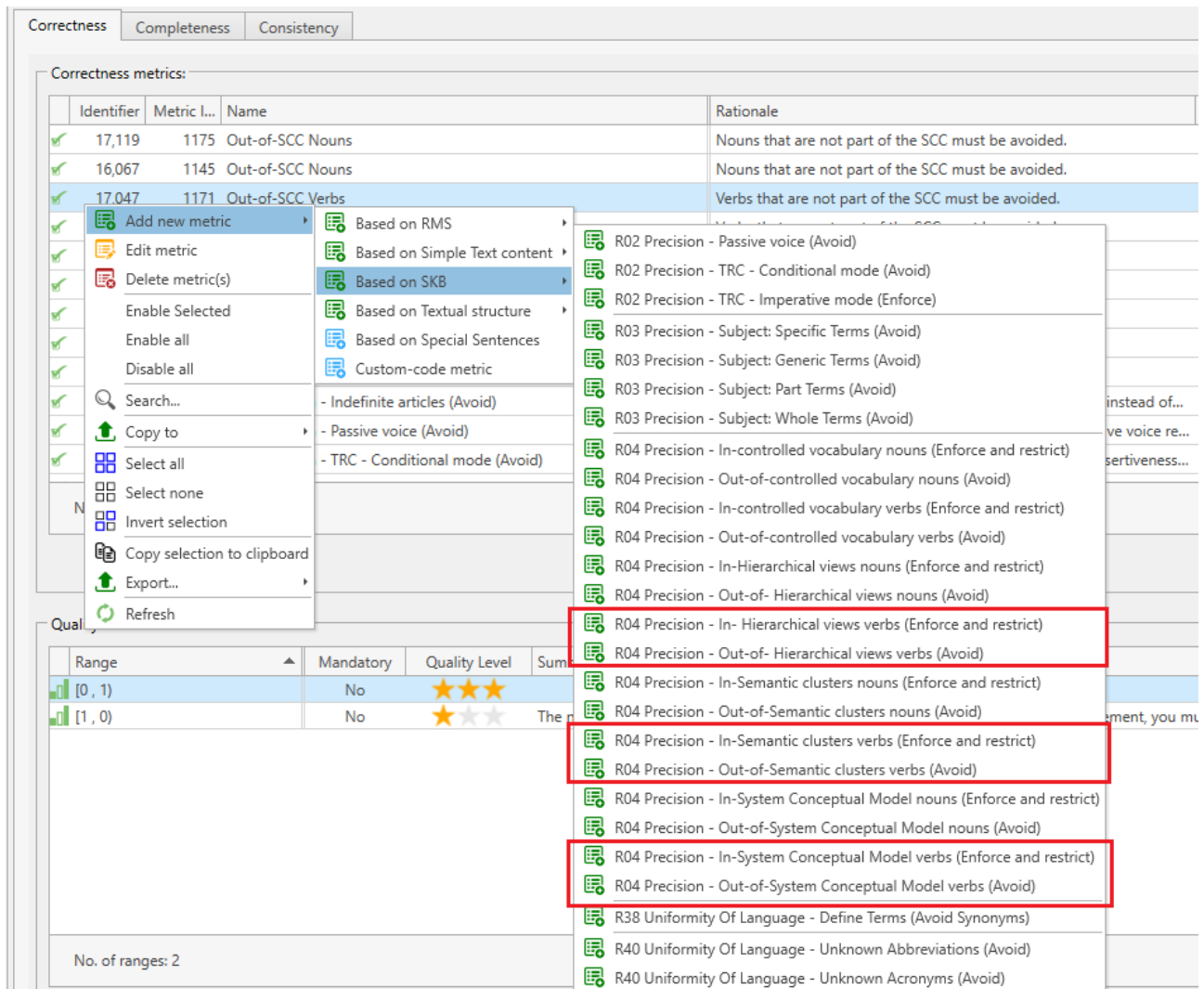
- Add new metric
  - Based on RMS
  - Based on Simple Text content
  - Based on SKB
  - Based on Textual structure
  - Based on Special Sentences
  - Custom-code metric
- Edit metric
- Delete metric(s)
- Enable Selected
- Enable all
- Disable all
- Search...
- Copy to
  - Indefinite articles (Avoid)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

Highlighted Metrics (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 20.** Correctness metrics related to nouns

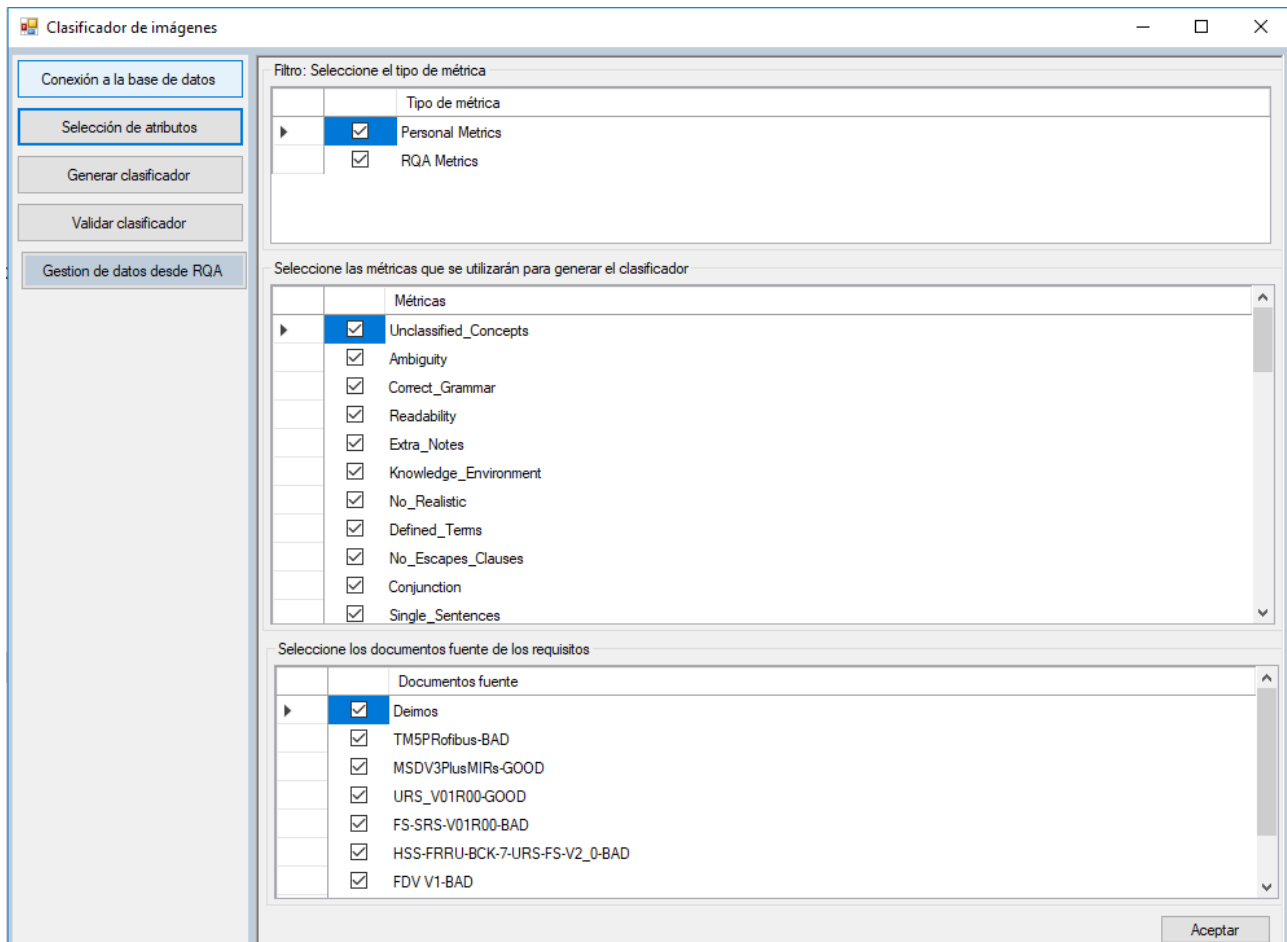
All of these metrics but related to verbs instead of nouns are also implemented in the RQA tool (see Figure 21).



**Figure 21.** Correctness metrics related to verbs

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

Machine learning techniques are used to improve the quality of requirements. The tool showed in Figure 22 is connected to RQA to extract the values of correctness metrics related with a set of requirements.



**Figure 22.** Tool to generate classifiers using machine learning with metrics of RQA

This information is used to generate a classifier that provides the quality of the requirements (see Figure 23).

```
J48 pruned tree
-----
1_Domain_concepts <= 1
| 1_Acronyms <= 0
| | 1_Domain_concepts <= 0: 2 (262.0/9.0)
| | 1_Domain_concepts > 0
| | | 1_Boilerplates_matching <= 0
| | | | 1_Connectors <= 1
| | | | | 1_Connectors <= 0
| | | | | | 1_Unclassified_verbs <= 0
| | | | | | | 1_Implicit_sentences <= 0
| | | | | | | | 1_Text_length(words) <= 10: 1 (5.0)
| | | | | | | | 1_Text_length(words) > 10: 2 (30.0/9.0)
| | | | | | | | 1_Implicit_sentences > 0: 1 (2.0)
| | | | | | | 1_Unclassified_verbs > 0: 2 (52.0/10.0)
| | | | | 1_Connectors > 0
```

**Figure 23.** Example of classifier

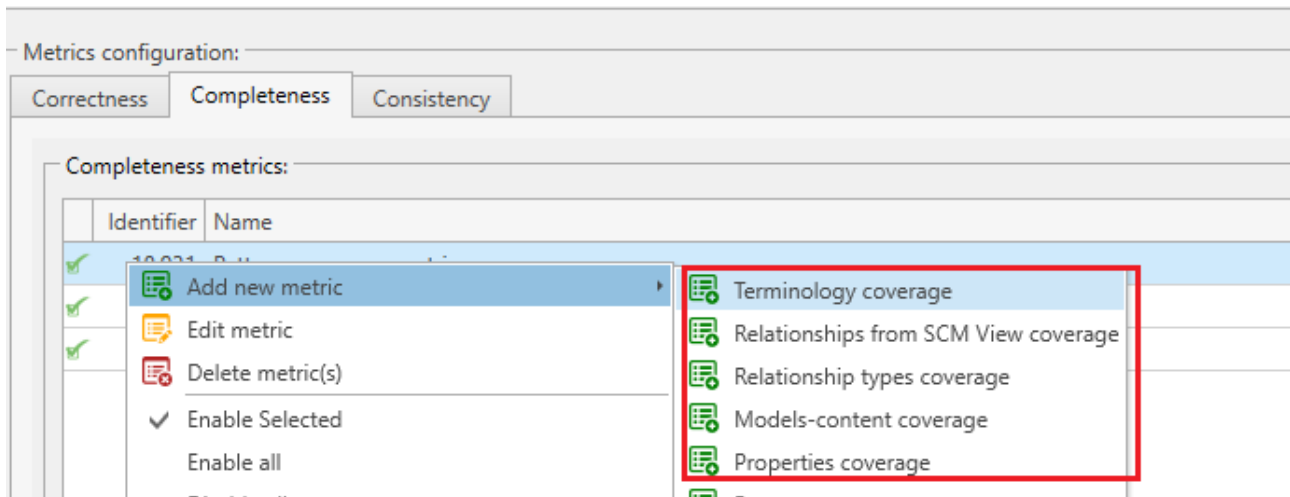
### 2.2.3.2.3 Metrics for models

The RQA tool allows to create metrics to evaluate the completeness and consistency in models.

The completeness metric implemented are (see Figure 24):

- Terminology coverage

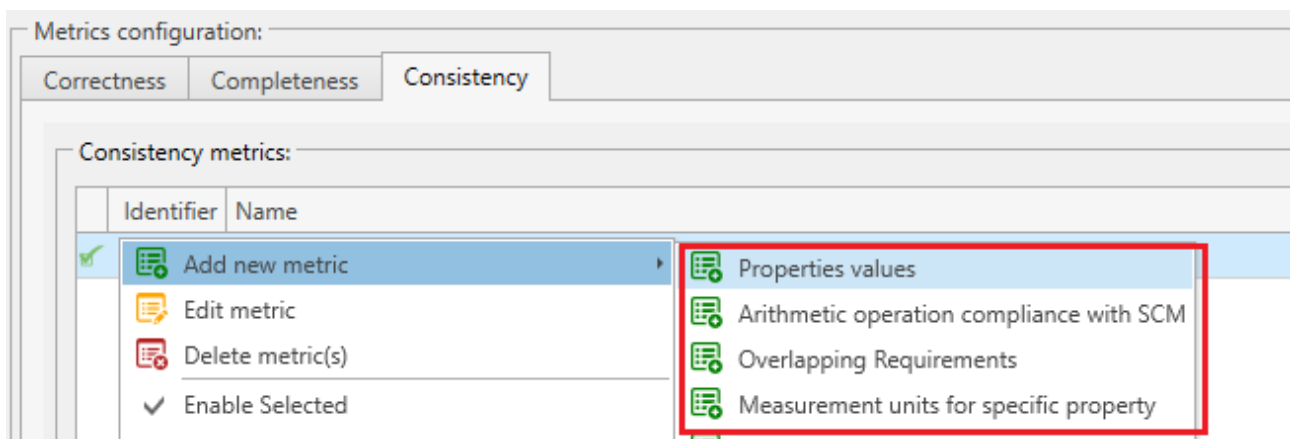
- Relationships from SCM View Coverage
- Relationship types coverage
- Model-content coverage
- Properties coverage



**Figure 24.** Completeness metrics for models

The consistency metric implemented are (see Figure 25):

- Property values
- Arithmetic operation compliance with SCM
- Overlapping requirements
- Measurement units for specific property

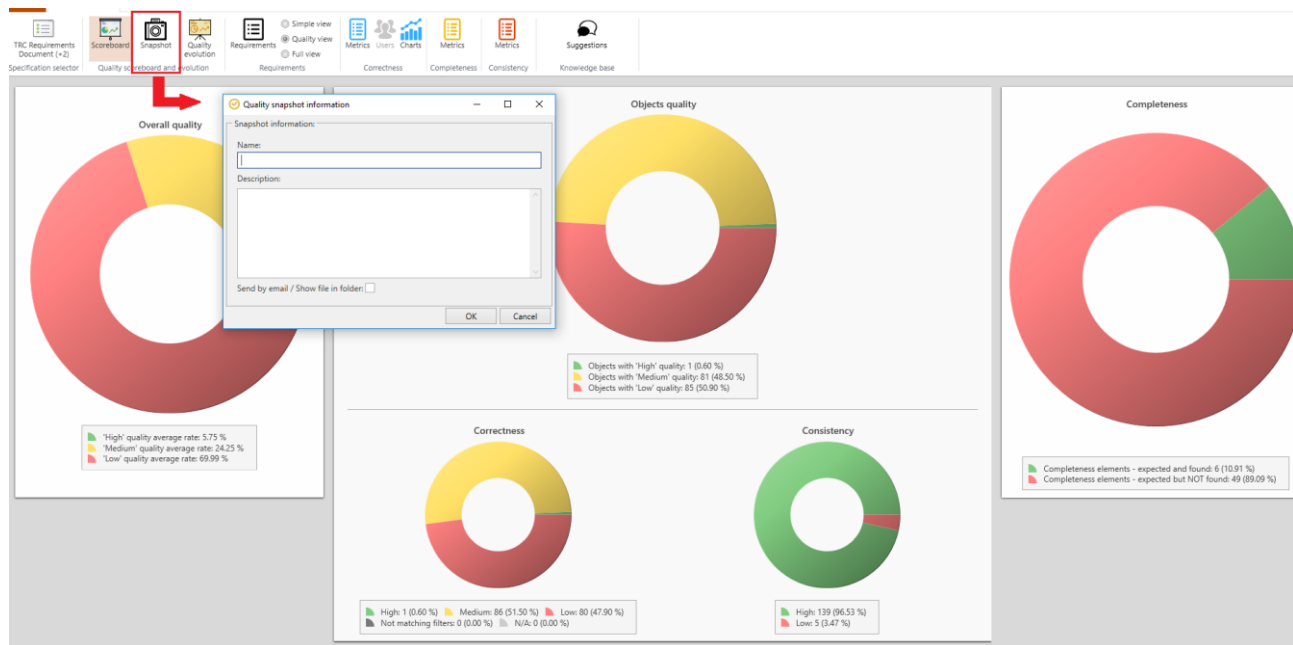


**Figure 25.** Consistency metrics for models

The metric “Arithmetic operation compliance with SCM” implements the update presented in the section “2.4.4.1.2 Consistency metric” in the deliverable D3.2 [14]. With this update, the metric takes into account the possible transformation between measurement unit of the same magnitude even if belongs to different measurement system.

#### 2.2.3.2.4 Quality evolution (with respect to time)

The RQA tool implements the functionality to save the snapshot of quality information of the project over the time (see Figure 26).



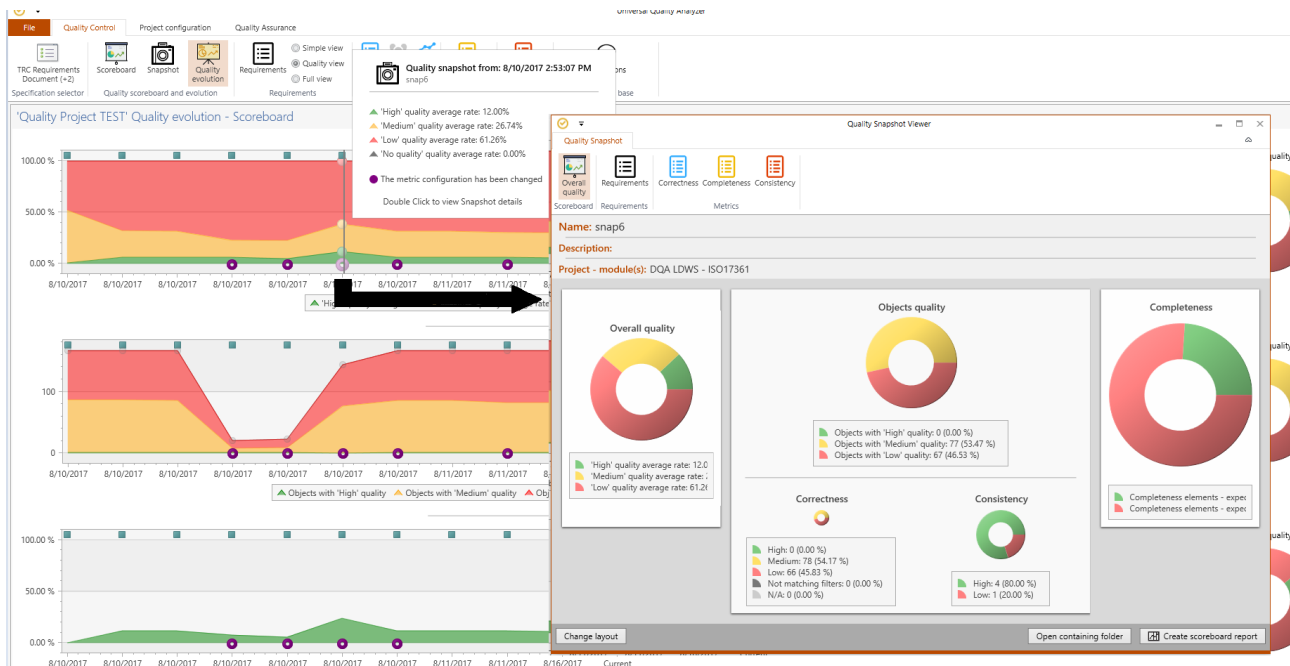
**Figure 26.** Saving snapshot with the quality of the project

The tool allows a graphical representation of the different snapshots saved (see Figure 27).



**Figure 27.** Graphical representation of the quality evolution

It is possible to show the information that composes the snapshots (see Figure 28 ).



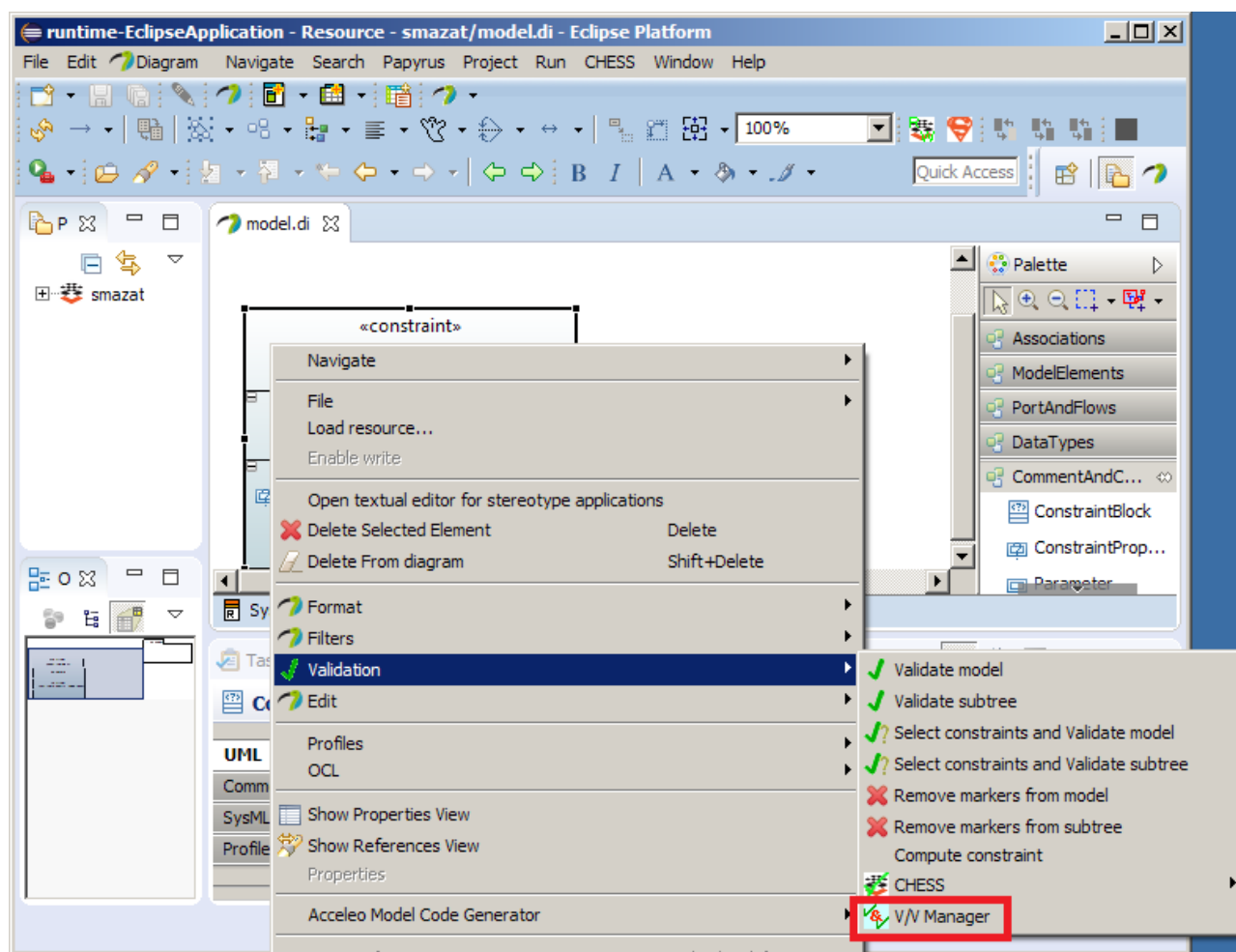
**Figure 28.** Information of the snapshot

### 2.2.3.3 V&V Manager

V&V Manager is an Eclipse plugin under development that enables invocation of multiple verification and validation tools of requirements or contracts directly from the AMASS platform. The V&V Manager for given requirements (and optionally also system architecture or design) calls verification server using OSLC Automation integration to get the V&V Assurance results (whether the requirements are consistent, non-redundant, non-vacuous, realizable and in case of system architecture or design also if requirements comply with given system).

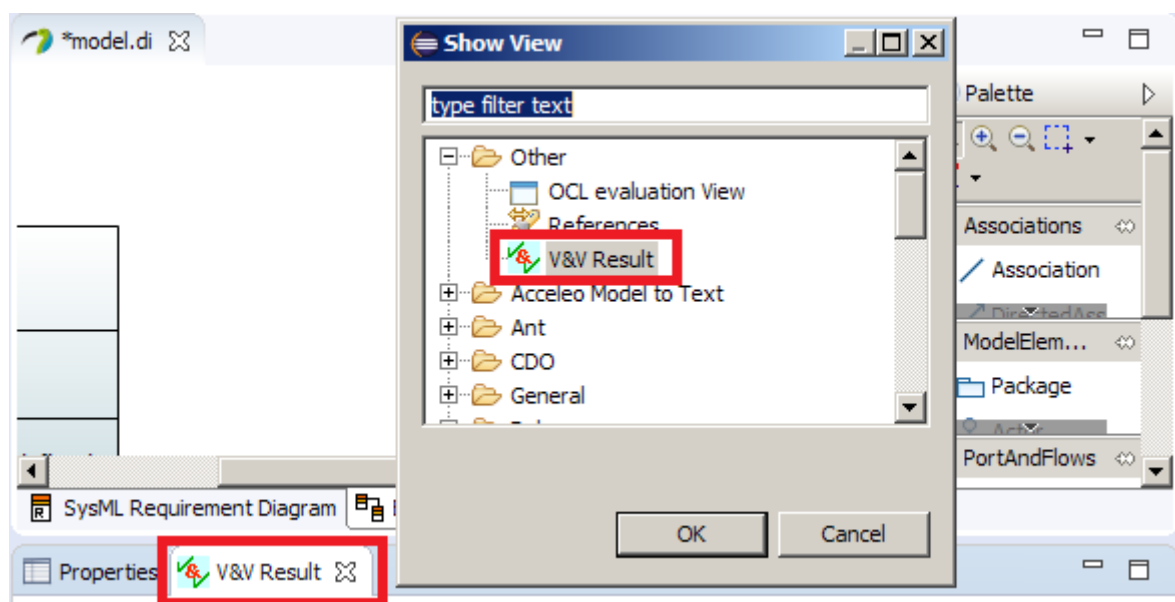
#### Implementation progress

The V&V Manager plugin implementation is in its initial stage. The relevant Eclipse plugin has been created. The command for running the V&V Manager is available, but does not support the whole expected functionality yet.



**Figure 29.** GUI element used to run the V&V Manager

The view for presenting the result of the verification can be shown. This view can display textual documents.



**Figure 30.** Switch in on the V&V Result view



The communication with the Verification Server is being elaborated.

#### **2.2.3.3.1 Verification Servers and Implementation Progress**

The communication between V&V Manager (the Eclipse plugin) and the Verification Server is based on OSLC, i.e. there is a specification describing interaction between these two parts, which confirms to OSLC Performance Monitoring and OSLC Automation specifications. The V&V Manager is an OSLC consumer and the Verification Server is an OSLC provider.

Currently, there is just a one public verification server (hosted at Masaryk University) and it is not fully ready yet. Verification servers at Honeywell are not public since they host also licensed V&V tools and are used for Honeywell confidential data. The V&V tools that are planned to be used for performing the semantic requirement analysis tasks have been already installed. What remains to be done is to finish implementation of the V&V manager; the communication with the Verification Server is both fully specified and tested.

It shall be relatively easy to add another verification servers as the interaction between V&V Manager and verification server will be completely tested. Therefore, everyone shall be able to create his/her own verification server in order to employ his/her own verification tools if needed. Note that although OSLC resources are defined in terms of RDF properties and operations on resources are performed using HTTP, i.e. OSLC provider is usually located on a remote server, it is also possible to use local verification servers running on localhost.

#### **2.2.3.3.2 Semantic Requirement Analysis**

This V&V technique formally proves if a given set formal requirements or contracts are consistent, non-redundant, non-vacuous, realizable or complete. Our approach is to execute multiple V&V tools and its configurations at once on multiple verification servers in parallel since often even V&V tool expert proficient in formal methods cannot determine which V&V tool and configuration will bring the best result fastest. Moreover, especially for model checking this approach distributes the computationally extensive V&V tasks to multiple servers is the fastest way to get the V&V results.

The screenshots below show the example verification results from the Honeywell proprietary tool ForReq. The same results will be visible from AMASS platform after the V&V Manager implementation is finished. The Figure 31 shows requirements that could be realised by trivial system, which suggests that the requirements are incomplete.

The analysis internally calls Acacia+ (same as RQA above) to obtain the realizability witness: a strategy that prescribes what reactions to input signals will lead to requirements satisfaction. In addition to demonstrating the requirements to be realizable, ForReq also interprets this witness to estimate the complexity of the requirements, and thus to some extent their completeness. For each input and output signal we compute the coverage by user requirements. The best requirements can only be satisfied if the system may need to react to a change in the value of each particular input. On the other hand, if a system can completely ignore some (or all) input signals, then we proclaim the requirements to be trivially satisfied. In a similar manner, ForReq assigns a degree of coverage to every signal, ranging from “fully covered” to “not covered” and reports this complexity analysis to the user in a comprehensive manner.

158.138.138.152 Requirements Verification							
Requirement ID	Progress	Text	Server	Consistency	Redundancy	Realisability	
	1/1	All requirements have been proven consistent.					
	1/1	There is no redundancy in the requirements.					
	1/1	The requirements are realisable. But trivially: no input needs to be distinguished. Output gesture could be forever same as or forever different from tap2times. Output gesture could be forever same as or forever different from long_press.					
SW_HL_02_0002	Formal	The Gesture Recognition shall set Gesture Matches to "TAP2TIMES" when all of the following conditions are satisfied: <ul style="list-style-type: none"> <li>o Touch 1 Time To Prev is greater than 0.25</li> <li>o Touch 1 Contact 1 Movement is "STATIC"</li> <li>o Touch 1 Contact 1 Time is lower than 0.25</li> <li>o Touch 2 Time To Prev is lower than 0.3</li> <li>o Touch 2 Contact 1 Movement is "STATIC"</li> <li>o Touch 2 Contact 1 Time is lower than 0.4</li> </ul>	Rak0	0.132414 s.	21.8876 s.	3.00794 s.	
SW_HL_02_0004	Formal	The Gesture Recognition shall set Gesture Matches to "LONG PRESS" when all of the following conditions are satisfied: <ul style="list-style-type: none"> <li>o Touch 1 Time To Prev is greater than 0.25</li> <li>o Touch 1 Contact 1 Movement is "STATIC"</li> <li>o Touch 1 Contact 1 Time is greater than 0.4</li> </ul>	Rak0	0.132414 s.	21.8876 s.	3.00794 s.	

**Figure 31.** Example of requirements from Gesture Recognition system (Case Study 7) that are only trivially realisable

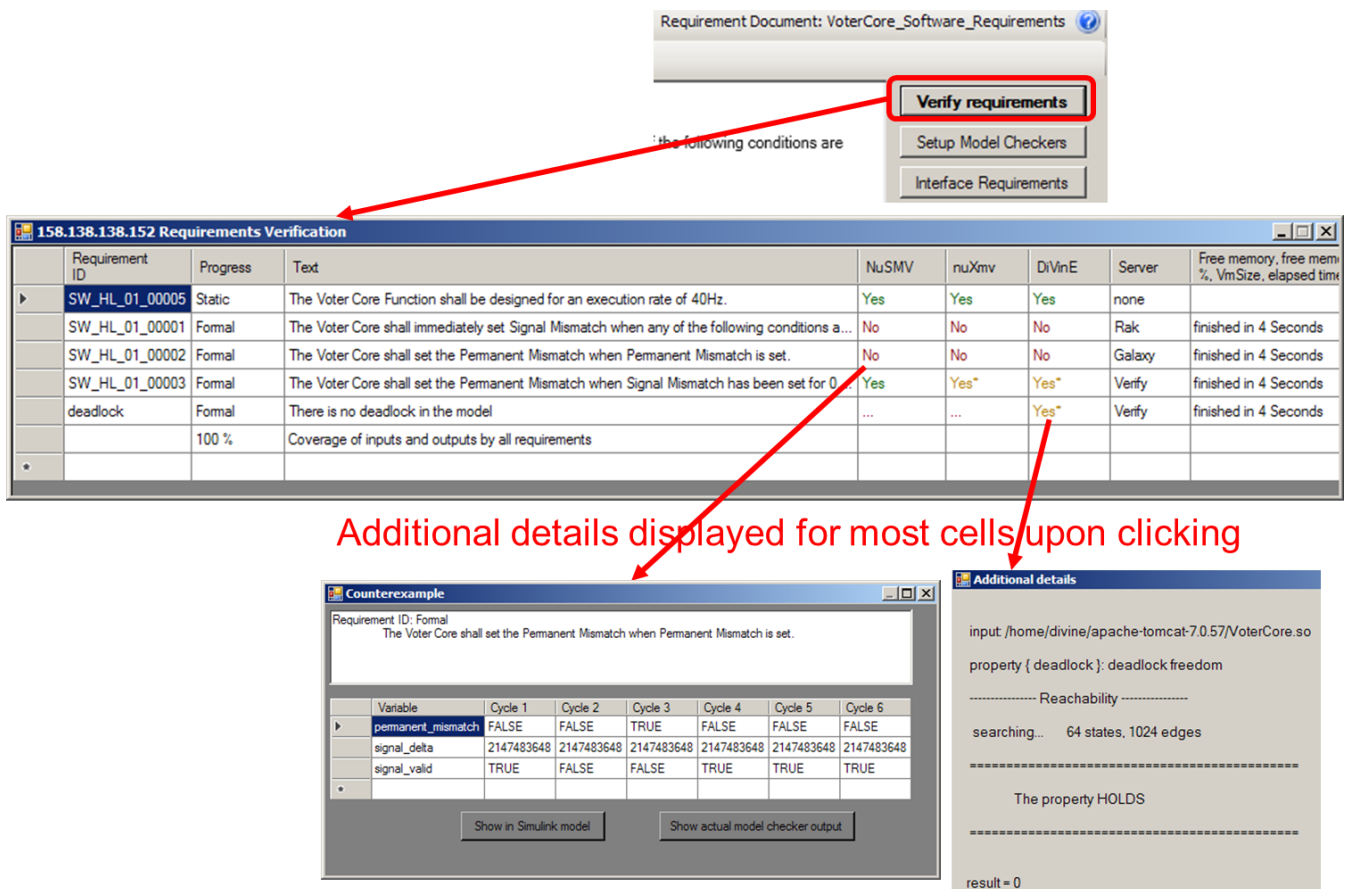
158.138.138.152 Requirements Verification							
Requirement ID	Progress	Text	Server	Consistency	Redundancy	Realisability	
	1/1	All requirements have been proven consistent.					
	1/1	There is no redundancy in the requirements.					
	1/1	The requirements are not realisable.					
SW_HL_01_00005	Static	The Voter Core Function shall be designed for an execution rate of 40Hz.					
SW_HL_01_00001	Formal	The Voter Core shall immediately set Signal Mismatch when any of the following conditions is satisfied, otherwise clear Signal Mismatch: <ul style="list-style-type: none"> <li>o Signal Delta is greater than 1</li> <li>o Signal Valid is invalid.</li> </ul>	Rak0	0.0315789 s.	1.34075 s.	0.313859 s.	
SW_HL_01_00002	Formal	When Permanent Mismatch is set, the Voter Core shall set the Permanent Mismatch and latch forever.	Rak0	0.0315789 s.	1.34075 s.	0.313859 s.	
SW_HL_01_00003	Formal	The Voter Core shall set the Permanent Mismatch when Signal Mismatch has been set for 0.03 seconds, otherwise clear Permanent Mismatch.	Rak0	0.0315789 s.	1.34075 s.	0.313859 s.	
SW_HL_01_00004	Formal	The Voter Core shall initialize the Permanent Mismatch to FALSE during the first frame execution.	Rak0	0.0315789 s.	1.34075 s.	0.313859 s.	

**Figure 32.** Example of requirements that are consistent, non-redundant and not realisable

### 2.2.3.3.3 Formal Verification of Requirements against System Design

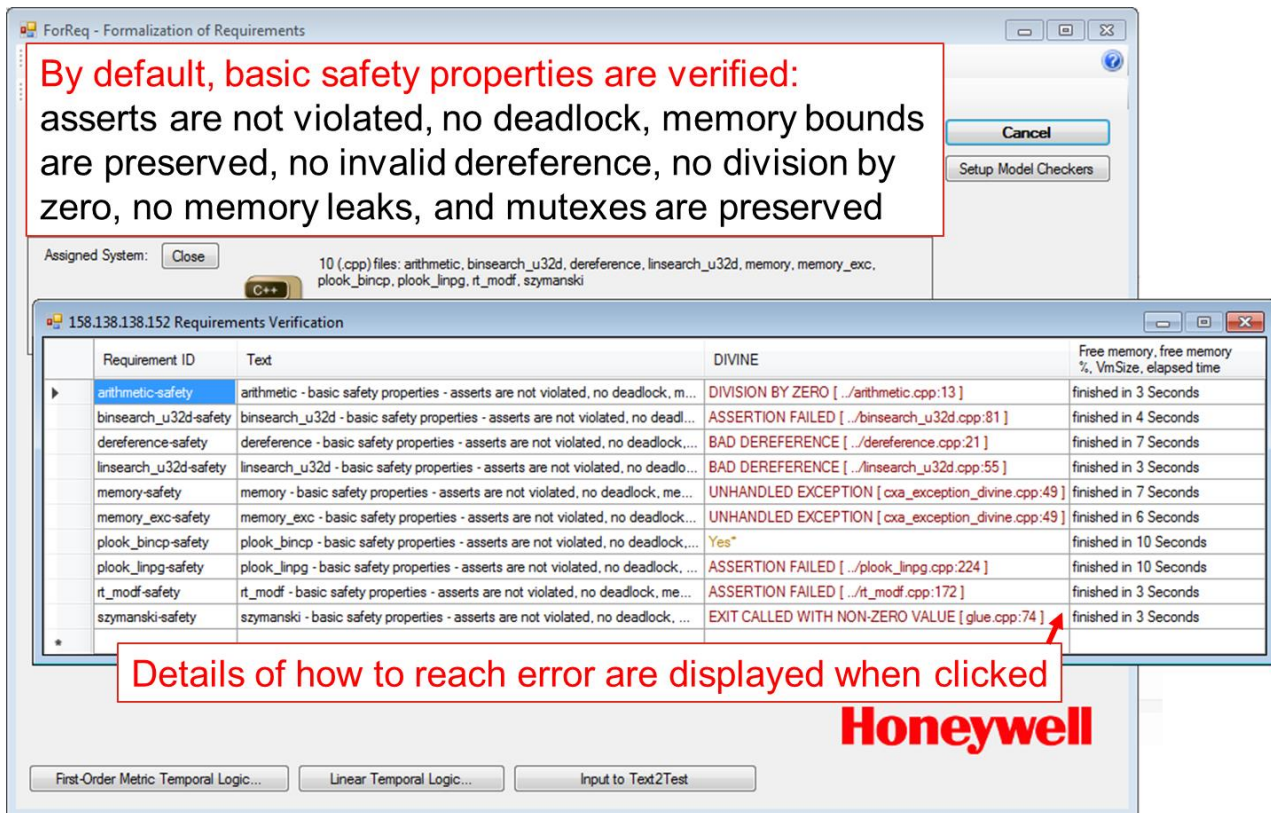
When system architecture or system design is presented each requirement should be verified for compliance with the system. This needs requirements to be formal and mapped to the system. The Figure below shows example of few requirements and its results as provided by 3 model checkers from 3 different verification servers. It should always be the case that the V&V tools agree with the result. However, often only some of the model checkers or its configurations are able to return the complete result.

When the requirement is not satisfied by the given system, the counterexample is provided in the form of table showing relevant input and output values in time that falsify given requirement and also in the case of Simulink system design the special counterexample model could be generated that shows the falsifying behavior.



**Figure 33.** Details for requirements checking

For system design in C or C++, only DIVINE LLVM model checker is currently integrated. Simple requirements could be translated to the form of C asserts and verified by DIVINE model checker jointly with other safety properties or C asserts that are not derived from requirements. As demonstrated on the Figure below.



**Figure 34.** Checking and proposed error handling

#### 2.2.3.4 Generate fault trees from the behavioral model and the fault injection

Generation of fault tree from the behavioural and fault model is supported by xSAP, a tool for safety assessment of synchronous finite-state and infinite-state systems<sup>11</sup>.

CHESS implements a seamless integration with xSAP to allow automatic generation of fault tree starting from the information made available in the CHESS model. In particular, the following information available in CHESS is used for the transformation to xSAP:

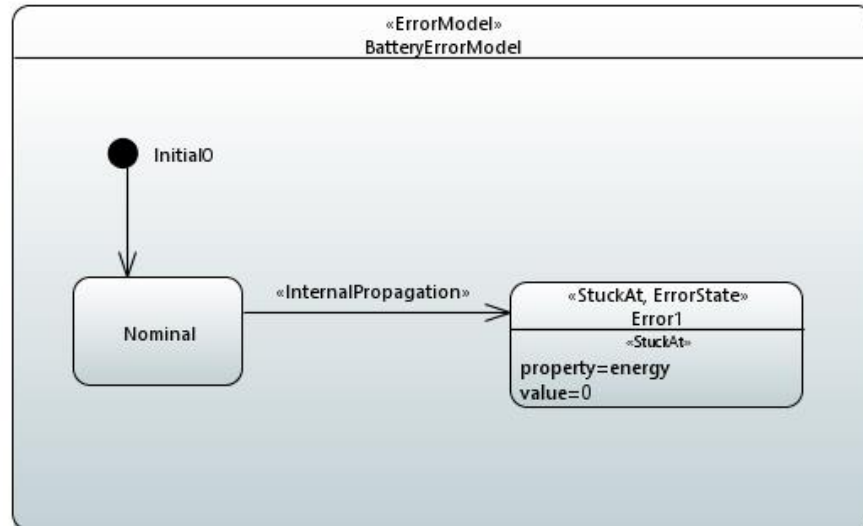
- System components (hierarchical architecture), basically SysML Blocks/UML Component with ports definition and composite relationships
- For each component:
  - The nominal behaviour, modelled by using state machine; the activities in the state machine have to be specified by using the NUSMV language<sup>12</sup>;
  - The error behaviour, modelled by using a state machine stereotyped with <<ErrorModel>> (see Figure 35) stereotype available from the CHESS dependability profile<sup>13</sup>. The CHESS

<sup>11</sup> <https://xsap.fbk.eu/>

<sup>12</sup> The language used by xSAP to represent the nominal model, see [http://nusmv.fbk.eu/NuSMV/papers/sttt\\_j/html/node7.html](http://nusmv.fbk.eu/NuSMV/papers/sttt_j/html/node7.html)

<sup>13</sup> CHESS comes with a dedicated profile for dependability for modelling safety aspects related to the system architecture. The metamodel from which the CHESS dependability profile has been derived is the SafeConcert metamodel; this metamodel is presented in AMASS D3.2 Appendix C.

dependability profile is also used to model error states, error propagation (e.g. InternalPropagation in Figure 35) and failure condition (e.g. stuckAt value, inverted error) upon component properties.



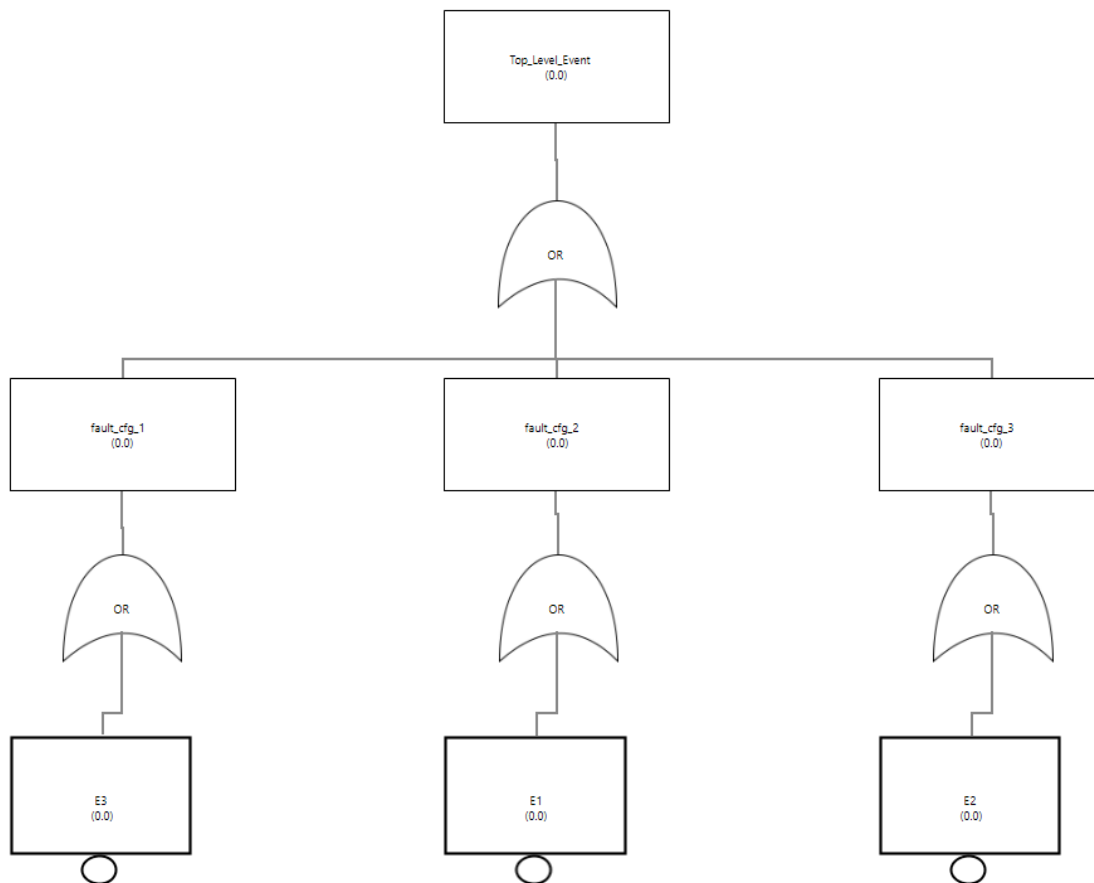
**Figure 35.** CHES error model state machine

An initial integration between CHES and xSAP was originally developed in SafeCer. In AMASS this integration has been reviewed; the model-to-text transformation has been extended and fixed according to the latest modifications of the CHES profile, in particular of the CHES Contract sub-profile. Moreover, some bugs have been discovered and fixed.

In the second prototype, CHES provides a Fault Tree View to graphically represent the result of the analysis as table or tree, see respectively Figure 36 and Figure 37.

	Name	Description
Event fault_cfg_1	fault_cfg_1	
Gate		
Event E3	E3	ps.StateMachine2.mode_is_Error1
Event fault_cfg_2	fault_cfg_2	
Gate		
Event E1	E1	ps.backupBat.BatteryErrorModel.mode_is_Error1
Event fault_cfg_3	fault_cfg_3	
Gate		
Event E2	E2	ps.primaryBat.BatteryErrorModel.mode_is_Error1

**Figure 36.** Example of fault tree represented as a table



**Figure 37.** Example of fault tree represented as tree

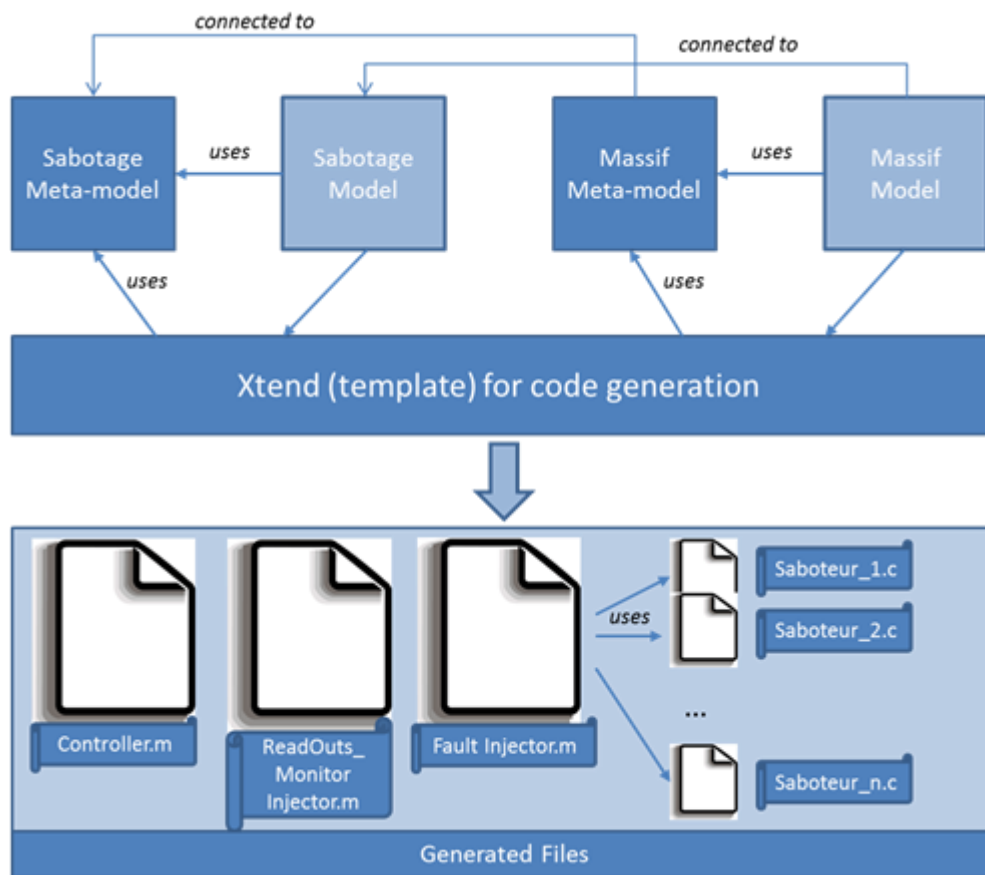
### 2.2.3.5 Simulation-based Fault Injection

Model-based design combined with a simulation-based fault injection technique poses as a promising solution for an early safety assessment of automotive systems. The fault injection functionality is supported by means of the Sabotage simulation fault injection framework. So far, most of the work has been developed as a set of manually coded Matlab scripts and C code. One of the main goals that AMASS promotes is the use and creation of model-based solutions. In that direction, we are currently working on an Eclipse-based fault injection framework, which will be further explained in this section. This framework will provide a model-based approach to configure, create and run the fault injection experiments.

By applying such technique to the fault injection domain, the user does not need to be familiar with the low-level configuration format of the fault injection technology (e.g. Xtend, Matlab or C).

It has to be noted that this is currently an ongoing work and it is thought to be released as part of the Prototype 3. However, some preliminary results and concepts are already covered in this deliverable.

Figure 38 depicts the main technologies tackled in order to build up our Eclipse-based fault injection framework.



**Figure 38.** Sabotage design architecture

After investigating several approaches that allow the modelling of the fault Injection experiments (e.g. UML profile), the one based on Ecore meta-model has been selected. This meta-model is under development and it is used to model the structure of the data models of the fault injection domain. As explained in D3.2 [14], the Sabotage framework creates the faulty system model under test (SMUT) by the fault injector module. Considering the SMUT a Simulink model, the new Eclipse feature Massif is used to support the easy handling of Matlab/Simulink models and import that information to EMF. The complete Massif Ecore description can be found in [13].

On the other hand, Model-to-text transformations are adopted for automation of an implementation step. More specifically, the template language Xtend [12] is applied to generate Matlab and C code. It employs a template-based code generation to export a resulting algorithm into those programming languages. The template system allows readable string concatenation through including a set of tokens, which are replaced by the algorithms computation code during the code generation process. The tokens to be completed are the ones coming from the information specified in Sabotage and Massif meta-models.

In brief, Xtend technology generates the Matlab code through a template toolkit. Some dynamic areas from those templates are completed adding the information coming from Sabotage and Massif (*SMUT.simulink*) meta-model instances.

The following lines explain some of the functionalities in a more accurate way:

- Configuration of the fault injection experiments:

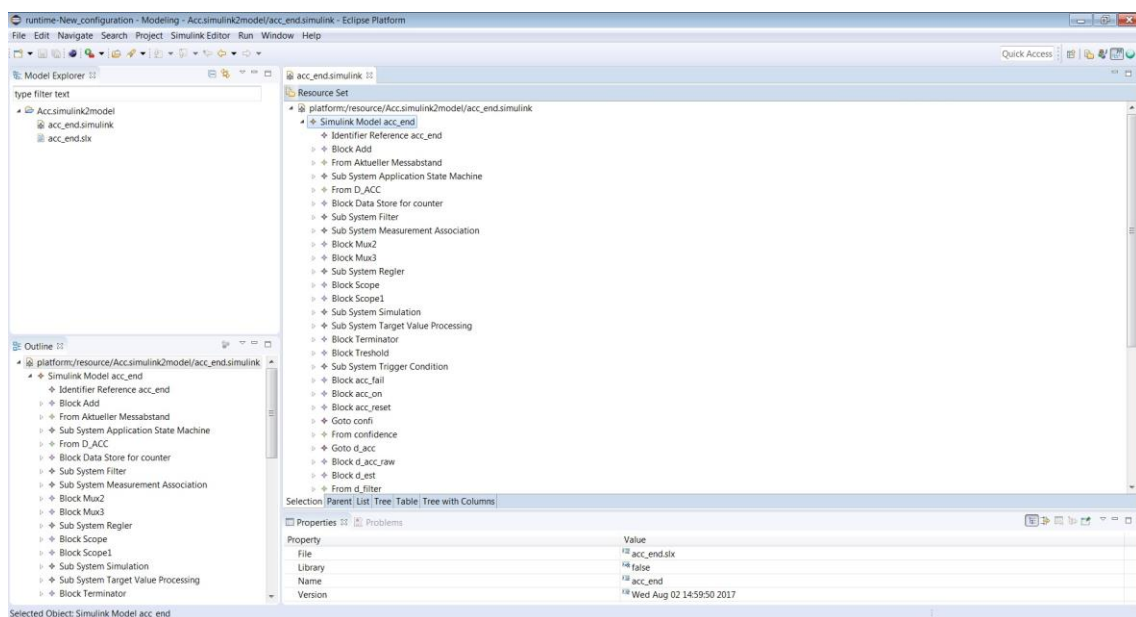


One of the major issues regarding the configuration of the fault injection experiments and the creation of the fault list [14] is to define where to inject the faults. Those faults reproduce a failure behavior of a certain component. In order to extract the necessary information regarding possible injection points, Massif is used. Massif is a new Eclipse feature to support the easy handling of MATLAB Simulink models by providing import and export capabilities to/from EMF [15].

By importing the Simulink model to Massif, the necessary information regarding possible injection points (i.e. fault target) is extracted. This includes information regarding input and output ports or component types that will be connected to our fault target class from the Sabotage meta-model.

The same concept applies to the observation points, monitor or read-outs. This information needs to be specified based on the current model.

An example of the Massif model for an ACC is shown in Figure 39 :

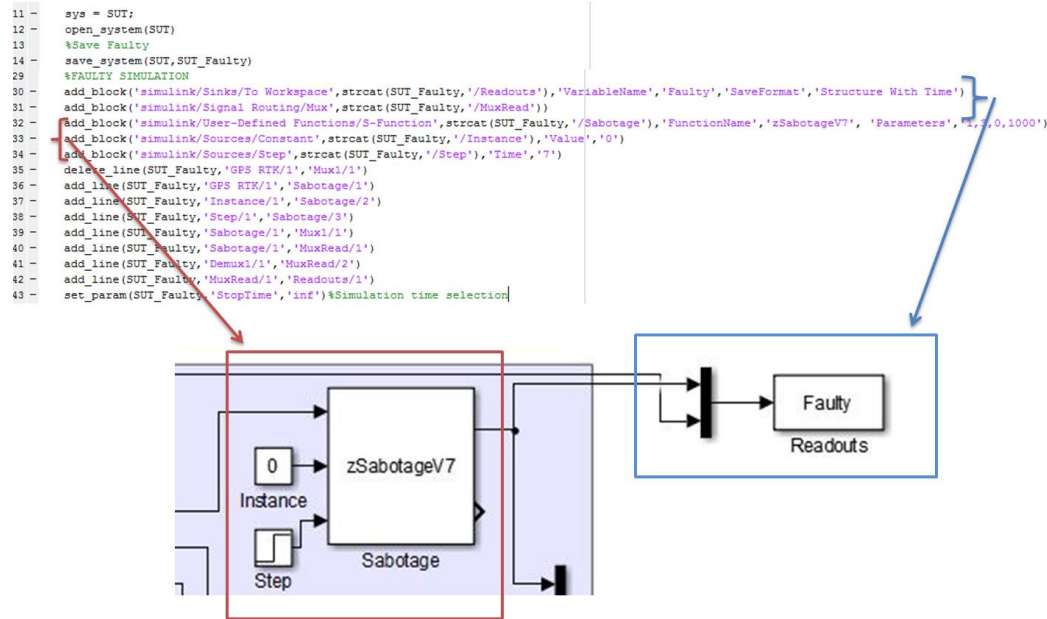


**Figure 39.** Example of a Massif model

- Generation of the fault injection experiments:

One of the main remarkable features is the construction of the faulty Simulink model. After importing Sabotage and Massif meta-models, the Xtend application (under development) creates the *FaultInjector.m* which is the main responsible of constructing the faulty Simulink model.





**Figure 40.** Example of the generated Fault Injector code

The Fault Injector creates and completes the C code of the Saboteurs represented as S-functions. Other alternatives can be explored as AMASS project evolves and depends on the project needs. This is related to fault model representativeness and Simulink block solutions will be evaluated versus purely C-based ones (S-functions).

```

double outputsignalx = *inSignalx[0];
double outputsignaly = *inSignaly[0];
if(NoiseEn >= 1){
    outputsignalx = outputsignalx + NoiseValue*(rand()%200 - 100 + 1)/100;
    outputsignaly = outputsignaly + NoiseValue*(rand()%200 - 100 + 1)/100;
}

if((*inRunStop[0] >= 1 && StuckatEn >= 1) || StuckatLatch == 1){
    if(StuckatType <= 0){outx = outk; outy = outy;}
    else {outx = StuckatValue; outy = StuckatValue;}
    StuckatLatch = 1;
}else{
    if((*inRunStop[0] >= 1 && TransEn >= 1){
        tri_iRamp.iTrigger = (int)*inRunStop[0];
        tri_iRamp.iDelay = TransTime;
        tri_iRamp.iClock = *inClock[0];
        TRI_Ramp(tri_iRamp);
        if(tri_iRamp.oActivated >= 1){
            if(TransType <= 0){
                outx = TransValue;
                outy = TransValue;
            }else{
                if(TransType == 1){
                    outx = transrandom;
                    outy = transrandom;
                }else{
                    outx = outk;
                    outy = outy;
                }
            }
        }
    }else{
        outx = outputsignalx;
        outy = outputsignaly;
    }
}else{
    outx = outputsignalx;
    outy = outputsignaly;
}
}
}
  
```

**Figure 41.** Example of a saboteur code

As part of the future work initiative, we plan to enhance our approach in order to investigate and add the following features as follows. The possible role and integration of other architecture-driven assurance functionalities with Sabotage will be studied during the project, especially establishing relations to contracts-based approach and model-based safety analysis. This means that the information regarding the fault type to be introduced can be linked as information contained in the system architecture (failure mode). In order to complete the information describing the faulty behaviour of a certain component and reproduce that behaviour in a form of a saboteur. As specified in Section 2.2.1.1, the failure modelling feature is currently supported via the CHES profile but not as part of the AMASS building block.

To sum up, as future work, connections to other AMASS meta-models will be considered. This will allow relating information such as the aforementioned failure mode of a certain component defined as part of

the system architecture (Papyrus/CHESS) to the Sabotage framework. Furthermore, connections to model safety analysis solutions will be analysed.

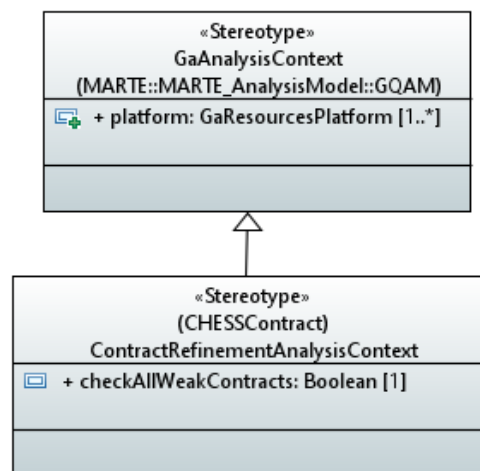
These functionalities will be provided in the next release of this deliverable (D3.6).

### 2.2.3.6 Traceability between different kinds of V&V evidence

This requirement is about traceability of different artefacts produced during the model based design and implementation process. For instance, the requirement cites “a contract-based, component-based specification should be traced with the corresponding analysis-results”.

Support for this requirement has been implemented in CHESS modelling language, in order to be able to trace analysis results with the set of model entities and assumptions used to perform that particular analysis. The adopted approach has been inherited by the MARTE modelling language, which comes with the concept of *analysis context* allowing to represent the set of model information needed to run a given analysis.

CHESS modelling language has been extended with AMASS specific analysis context; for instance the new stereotype named *ContractRefinementAnalysisContext* (see Figure 42) allows to collect the information available in the CHESS model that has to be used to run a given contract refinement analysis. The aforementioned information is basically the set of components with associated contracts that has to be analysed; in fact the CHESS model can comprise different views (e.g. functional, logical, physical) and different analysis can be run on each of the different views, or even different parts of the same view.



**Figure 42.** Analysis Context

The *ContractRefinementAnalysisContext* stereotype comes also with a Boolean attribute *checkAllWeakContracts* which can be used to specify which weak contracts have to be considered for the analysis; if the value is true all weak contracts available in the current components set identified by the analysis context are considered, otherwise only the weak contracts marked by the modeller as valid are given in input to the analysis.

According to the new modelling language support, CHESS tool has been modified to allow the user to invoke contract refinement analysis, the latter performed thanks to the integration with the OCRA tool (see section 2.2.3.3), by selecting an existing *ContractRefinementAnalysisContext*. Once the analysis has finished, analysis results can then be linked to the analysis context, and so to the target analysed set of components and associated contracts; this last step is not currently automatized and has to be made by the user by using the traceability capabilities discussed in section 2.2.2.1.

### 2.2.3.7 Generation of product-based assurance arguments from CHESS model

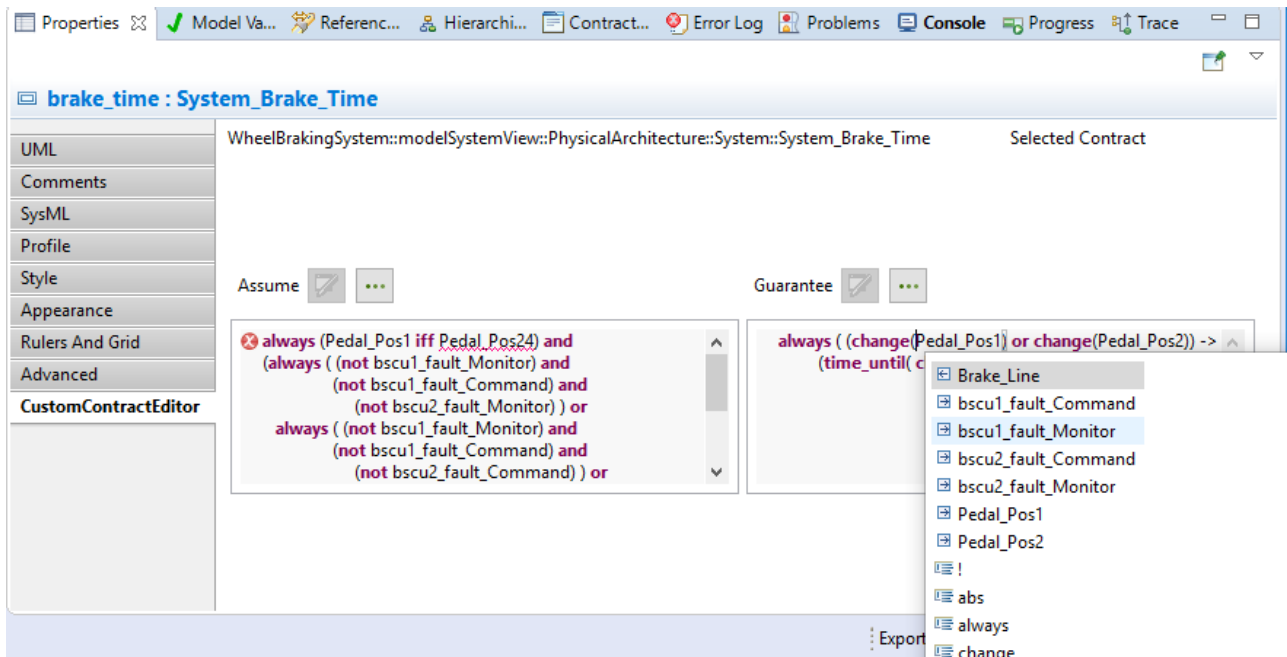
The generation of product-based assurance arguments is based on the assurance information associated with the strong and weak contracts. To include only the relevant weak contracts in generation we need to first know which of those hold in the current system. To achieve that, we have extended the CHESS tool by using the *checkAllWeakContracts* attribute when performing contract refinement analysis to transform all the weak contracts in OCRA format such that all strong contracts  $C=(A,G)$  are transformed into normal OCRA contracts  $C=(A,G)$ , while the weak contracts  $C_s=(B,H)$  are transformed into guaranteed implications in OCRA as  $C_w=(TRUE,B\Rightarrow H)$ . The refinement connection of  $C_w$  is inherited from the corresponding weak contracts. To check consistency of the weak assumptions in the given context and identify which weak contracts should be used in argument generation, we have extended CHESS tool to allow for property validation in OCRA of the weak contract assumptions. The result of both OCRA commands are saved in a file and previewed to the user. The results are used to update the status of the contracts. To perform the contract refinement analysis with all weak contracts, the user sets the Boolean attribute *checkAllWeakContracts* of the *ContractRefinementAnalysisContext* stereotype to TRUE and selects the Check Contract Refinement functionality. Then, to validate the weak contract assumptions, the user makes sure the *checkAllWeakContract* attribute is set to TRUE, and selects Validate Weak Contracts functionality.

Based on the contract status we create a set of argument-fragments in the corresponding assurance case project where they can be viewed in the assurance case editor. The generator uses a pre-existing argument pattern for the generation and the information from the traceability editor of the contracts and the assurance evidence. The generated argument-fragments include only assurance evidence of those contracts relevant in the given context, which is determined by the status attribute of the contracts. The argument-fragments generation can be performed once refinement analysis is successfully completed and contract validity check is done. The generation is performed from the *ContractRefinementAnalysisContext* argument generator property tab.

## 2.2.4 Contract-based Assurance Composition

### 2.2.4.1 Contract Editor with content assist

In the second AMASS prototype (Prototype P1), the contract definition and the property definition can be edited using an editor with content assist, see Figure 43. The latter provides two utilities: (1) it notifies whether a word does not belong to the language used or whether it is not a port or an attribute of the component of the editing contract/property. (2) It suggests the keyword of the language used and the ports and attributes of the component.



**Figure 43.** Contract Editor with content assist

In this example, in the editing area of the assume property, a wrong port name is notified. In the editing area of the guarantee property, it is suggested which are the compatible keywords to insert.

#### 2.2.4.2 Contract-based Views

In the second AMASS prototype (Prototype P1), CHES provides a hierarchical view that shows the decomposition of the system component into sub-components. It shows also the contracts assigned for each component. The system is graphically represented as the top element of the view (see Figure 44).

System Architectures		Number of Subcomponents and Contracts	
System		3	
bscu:BSCU		5	
bscu1:SubBSCU		2	
SubBSCU_CMD_Time			
SubBSCU_Safety			
bscu2:SubBSCU		2	
SubBSCU_CMD_Time			
SubBSCU_Safety			
switch:Select_Switch_Impl		2	
Select_Switch_Sel0_Time			
Select_Switch_Sel1_Time			
BSCU_CMD_Time			
BSCU_Safety			
hydraulic:Hydraulic		1	
Hydraulic_Brake_Time			
System_Brake_Time			

**Figure 44.** Hierarchical view of the system decomposed into sub-components and contracts

CHESS also provides a hierarchical view that shows the contracts with their refining contracts, see Figure 45. The weak contracts are graphically represented as a document with a “W” on top.

<div> <div>Properties</div> <div>Model Validation</div> <div>References</div> <div>Hierarchical Model View</div> <div>Contract Refinement View</div> </div>		
Refined Contracts	Number of sub-contracts	
System.brake_time	3	
bscu.cmd_time	6	
bscu1.cmd_time	0	
bscu1.safety	0	
bscu2.cmd_time	0	
bscu2.safety	0	
switch.sel0_time	0	
switch.sel1_time	0	
bscu.safety	2	
bscu1.safety	0	
bscu2.safety	0	
hydraulic.brake_time	0	

**Figure 45.** Contract Refinement View

#### 2.2.4.3 Contract refinement analysis

Contract refinement analysis is supported by the OCRA tool. CHESS comes with a seamless integration with OCRA which allows to invoke the analysis starting from the components and associated contracts available in the CHESS model. When the analysis is invoked through the CHESS tool the following steps are performed:

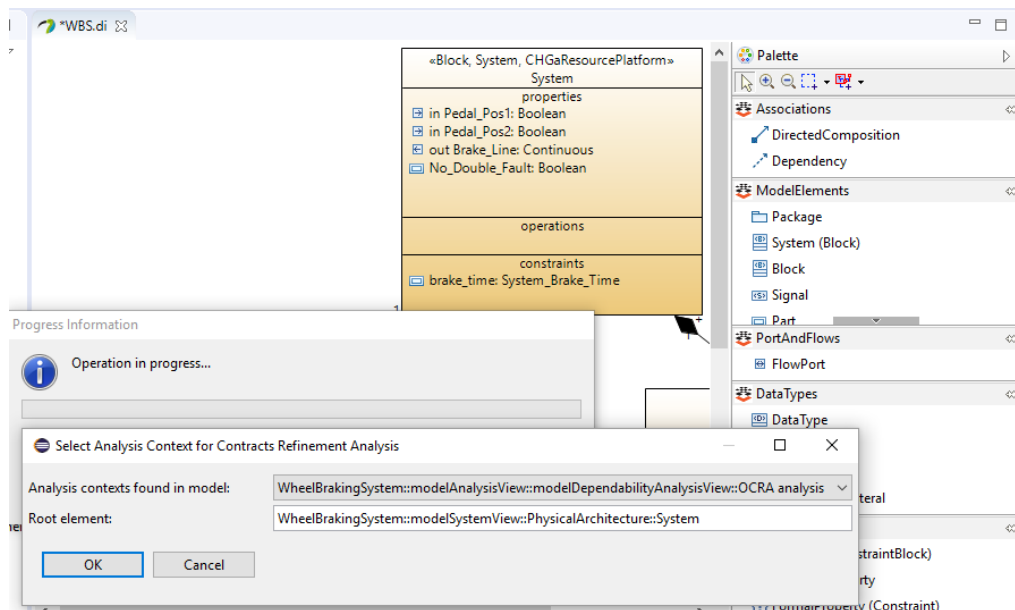
1. a validation is performed on the CHESS model to check that the modelled information is available and syntactically correct with respect to what is required by OCRA;
2. the user selects the analysis context that has to be taken into account;
3. model-to-text transformation from CHESS model to OCRA language is executed (.oss artefact derivation, see Figure 46);
4. the OCRA tool is invoked with the produced .oss and with the appropriate command option;
5. the results from the OCRA analysis are showed to the modeller in a dedicate window, and saved as output artefacts in a specific folder under the current CHESS project.

```

COMPONENT system
INTERFACE
INPUT Pedal_Pos1 : boolean ;
INPUT Pedal_Pos2 : boolean ;
INPUT bscu1_fault_Monitor : boolean ;
INPUT bscu2_fault_Monitor : boolean ;
INPUT bscu1_fault_Command : boolean ;
INPUT bscu2_fault_Command : boolean ;
OUTPUT Brake_Line : continuous ;
CONTRACT System_Brake_Time assume : always ( Pedal_Pos1 iff Pedal_Pos2 )
and ( always ( ( not bscu1_fault_Monitor ) and ( not bscu1_fault_Command ) and ( not bscu2_fault_Monitor ) ) or
always ( ( not bscu1_fault_Monitor ) and ( not bscu1_fault_Command ) and ( not bscu2_fault_Command ) ) or
always ( ( not bscu1_fault_Monitor ) and ( not bscu2_fault_Command ) and ( not bscu2_fault_Monitor ) ) or
always ( ( not bscu1_fault_Command ) and ( not bscu2_fault_Command ) and ( not bscu2_fault_Monitor ) ) ) ;
guarantee : always ( ( change ( Pedal_Pos1 ) or change ( Pedal_Pos2 ) ) -> ( time_until ( change ( Brake_Line ) ) <= 10 ) ) ;
REFINEMENT
SUB hydraulic : Hydraulic ;
SUB bscu : BSCU ;
CONNECTION bscu.Pedal_Pos1 := Pedal_Pos1 ;
CONNECTION bscu.Pedal_Pos2 := Pedal_Pos2 ;
CONNECTION Brake_Line := hydraulic.Brake_Line ;
CONNECTION hydraulic.CMD_AS := bscu.CMD_AS ;
CONNECTION hydraulic.Valid := bscu.Valid ;
CONNECTION bscu.bscu1_fault_Monitor := bscu1_fault_Monitor ;
CONNECTION bscu.bscu2_fault_Monitor := bscu2_fault_Monitor ;
CONNECTION bscu.bscu1_fault_Command := bscu1_fault_Command ;
CONNECTION bscu.bscu2_fault_Command := bscu2_fault_Command ;
CONTRACT System_Brake_Time REFINEDBY bscu.BSCU_CMD_Time , bscu.BSCU_Safety , hydraulic.Hydraulic_Brake_Time ;
COMPONENT Hydraulic
INTERFACE
INPUT CMD_AS : boolean ;
INPUT Valid : boolean ;
OUTPUT Brake_Line : continuous ;
CONTRACT Hydraulic_Brake_Time assume : TRUE ;
guarantee : always ( change ( CMD_AS ) -> ( time_until ( change ( Brake_Line ) ) <= 5 ) ) ;
COMPONENT BSCU
INTERFACE

```

**Figure 46.** Part of the OCRA input file, also called OSS (OCRA System Specification). It describes the system architecture represented by a tree of components (given by the decomposition into sub-components)



**Figure 47.** Selecting analysis context for contract refinement

The integration between CHES and OCRA was originally developed in SafeCer. In AMASS, this integration has been improved, in particular by introducing the analysis context support; moreover the model-to-text transformation has been reviewed according to the latest modifications of the CHES profile, in particular of the CHES Contract sub-profile.

#### 2.2.4.4 Contract-based Safety Analysis

The contract-based safety analysis detects the component failures identified 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. This dependency can be automatically computed based on the contract refinement. CHES interacts with OCRA to produce a fault tree in which each intermediate event represents the failure of a component or its environment.

#### 2.2.4.5 Contract-based verification of the behavioural model

The Contract-based verification of the behavioural model is supported by the OCRA tool. This functionality verifies if the finite state machines defined in the CHES model verify the contracts. The state machines are translated in the SMV language, where the behaviour is described by means of logical formulas that describe the initial states and the state transitions, see Figure 48. Meanwhile, the contracts, as already mentioned in Section 2.2.3.3, are translated to OCRA language in a .oss file. CHES sends such information as input to OCRA, and then in the Trace View, for each contract the result of the check is shown, see Figure 49.

```

-- =====
MODULE AlternateCommandCalculator(power, as_cmd_in_1, as_cmd_in_2)
  VAR
    as_cmd_out : real;

  ASSIGN
    as_cmd_out := case
      (power & as_cmd_in_1 >= as_cmd_in_2) : as_cmd_in_1;
      (power & !(as_cmd_in_1 >= as_cmd_in_2)) : as_cmd_in_2;
      !power : 0;
      TRUE : 0;
    esac;
  LTLSPEC NAME alternate_command_computation_norm_guarantee := (TRUE -> (( G ((power & as_c

-- =====
--                               End of module
-- =====

MODULE NormalCommandCalculator(power, brake_cmd, as_cmd)
  VAR
    brake_as_cmd : real;

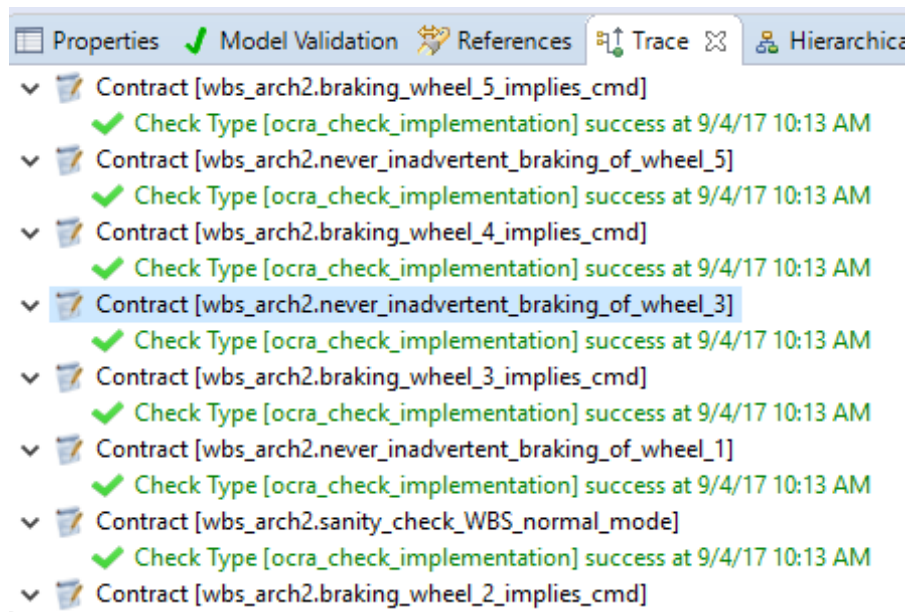
  ASSIGN
    brake_as_cmd := case
      (power & as_cmd = 1) : 0;
      (power & as_cmd = 0) : brake_cmd;
      TRUE : 0;
    esac;
  LTLSPEC NAME normal_command_computation_norm_guarantee := (TRUE -> (( G ((power & as_cmd

-- =====
--                               End of module

```

**Figure 48.** Part of an .SMV file representing the behaviour of the leaf components of the model





**Figure 49.** In this example, for each contract the results of the Contract-based verification are listed in the Trace View

### 3. Installation and User Manuals

The steps necessary to install the second prototype are going to be exhaustively described in the AMASS User Manual (currently in progress) and will not be repeated here. That document will contain all required steps and document references to set up the tools. There is currently no pre-packaged distribution.

In summary, this document is a user manual of the second AMASS tool prototype implementation (Prototype P1). The users can find the installation instructions, the tool environment description, and the functionalities for the creation of Standards and Process models (models representing Standards, Regulations, or Company-specific Processes), Assurance Projects and the associated Evidence models (Artefacts), Compliance Maps (so far, compliance maps from Reference Artefacts to Artefacts), and Argumentation models, in addition to Architecture models.



## 4. Implementation Description

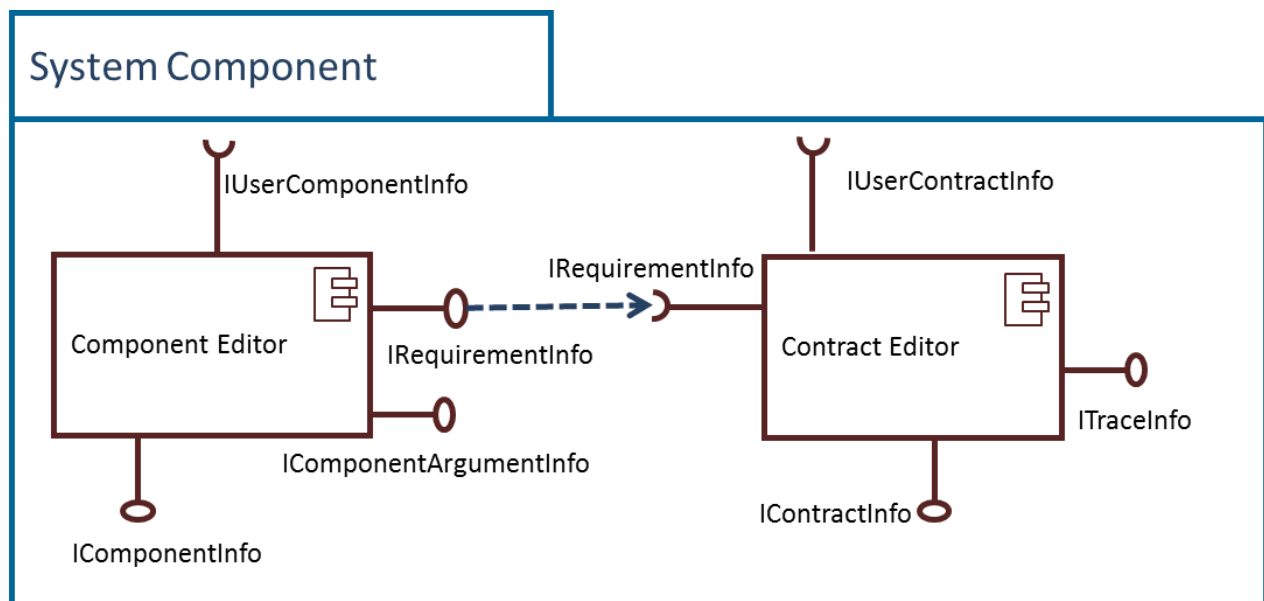
### 4.1 Implemented Modules

#### 4.1.1 System Component Specification Block

As documented in AMASS deliverable D2.3 [6], the System Component Specification logical building block decomposes into two sub-blocks (see Figure 50): the Component Editor and the Contract Editor. The purpose of the first tool module is to provide services for architecture specification; the second tool module provides services to store and instantiate contracts and to associate them to the architectural entities.

The two aforementioned blocks and associated services are made available in the AMASS platform through the usage of the Eclipse-Based Papyrus UML/SysML Editor extended with the CHESSE plugins. In particular, Papyrus contains plugins for edition of architectural/component-based models, together with the possibility to model requirements (by using the SysML profile support). CHESSE provides plugins for management of formal properties and contracts specification and their association to the architectural components.

The CHESSE profile for Contract (see D3.1) is implemented as a UML/SysML profile; the profile has been designed using the Papyrus editor facilities.



**Figure 50.** Tool modules for System Component Specification

#### 4.1.2 Architecture-Driven Assurance Block

As documented in AMASS deliverable D2.3 [6], the Architecture-Driven Assurance allows for explicit integration of assurance and certification activities with the CPS development activities, including specification and design. It decomposes into four sub-blocks: system architecture modelling for assurance, V&V-based Assurance Impact Assessment, Contract-Based Assurance Composition, and Assurance Patterns Library Management. The latter will be implemented in the third AMASS Prototype (Prototype P2).

## 4.2 Source Code Description

### 4.2.1 System Component Specification Block

Papyrus<sup>14</sup> is an Eclipse project and its source code is freely available through the Eclipse GIT server<sup>15</sup>.

The source code of the CHESSE contract editor is available through the Polarsys CHESSE project<sup>16</sup>.

Extensions to the Polarsys CHESSE project are foreseen during the context of AMASS project; the extensions will be developed by working on an AMASS dedicated code repository ([https://services.medini.eu/svn/AMASS\\_source](https://services.medini.eu/svn/AMASS_source)). Then, once the extensions are sufficiently mature, they will be pushed to the Polarsys CHESSE repository as AMASS contribution.

The additional CHESSE plugins that need to be installed on top of Papyrus environment to enable the CHESSE-based AMASS Contract Editor features are the following (see also Figure 51):

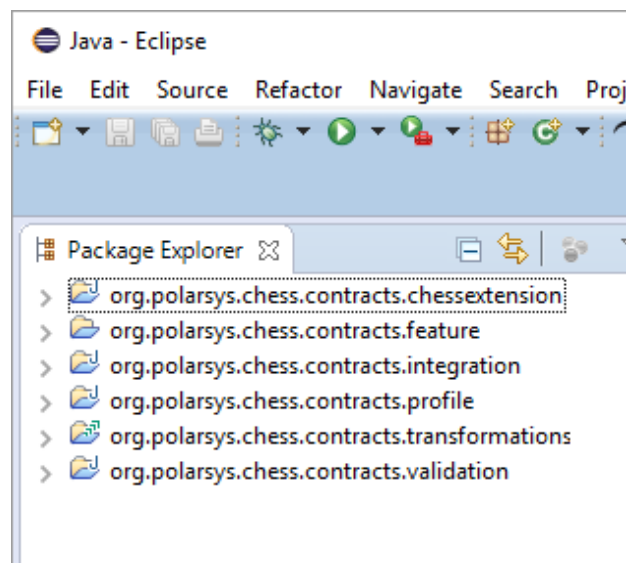
- `org.polarsys.chess.contracts.chessexension`: provides the Papyrus extension to easily work with the CHESSE Contract profile, for instance to facilitate the creation of CHESSE stereotypes.
- `org.polarsys.chess.contract.integration`: implements the integration with the OCRA and XSAP tools; in particular, it allows automatically invoking the aforementioned tools and getting back the obtained results within the Eclipse environment.
- `org.polarsys.chess.contracts.profile`: implements the CHESSE profile for contracts.
- `org.polarsys.chess.contracts.transformations`: implements the model of text transformation for the integration with the OCRA and XSAP tools; in particular a corresponding OCRA model can be generated starting from the components and contracts modelled in UML/SysML and CHESSE profile. The plugin adds dedicated command to the CHESSE Eclipse menu to invoke the transformations.
- `org.polarsys.chess.contracts.validation`: implements the validation of the constraints that the CHESSE model has to satisfy in order to allow the mapping to the OCRA language and then the integration with the OCRA tool.
- `org.polarsys.chess.contracts.feature`: allows to deploy/undeploy the CHESSE plugins related to contract-based design support.
- `org.polarsys.chess.contracts.contractPropertyManager`: allows the automatic generation of the contract component when a contractInstance is associated to a component.
- `org.polarsys.chess.utils`: contains methods related to CHESSE elements, contracts, and elements selected by the user via the GUI.

---

<sup>14</sup> <https://eclipse.org/papyrus/>

<sup>15</sup> <https://git.eclipse.org/c/papyrus/org.eclipse.papyrus.git/>

<sup>16</sup> <https://git.polarsys.org/c/chess/chess.git?h=develop>



**Figure 51.** CHES plugins supporting Contract Based Design

One important point to mention is that, in addition to the aforementioned support for contract design, the Polarsys CHES project provides additional features.

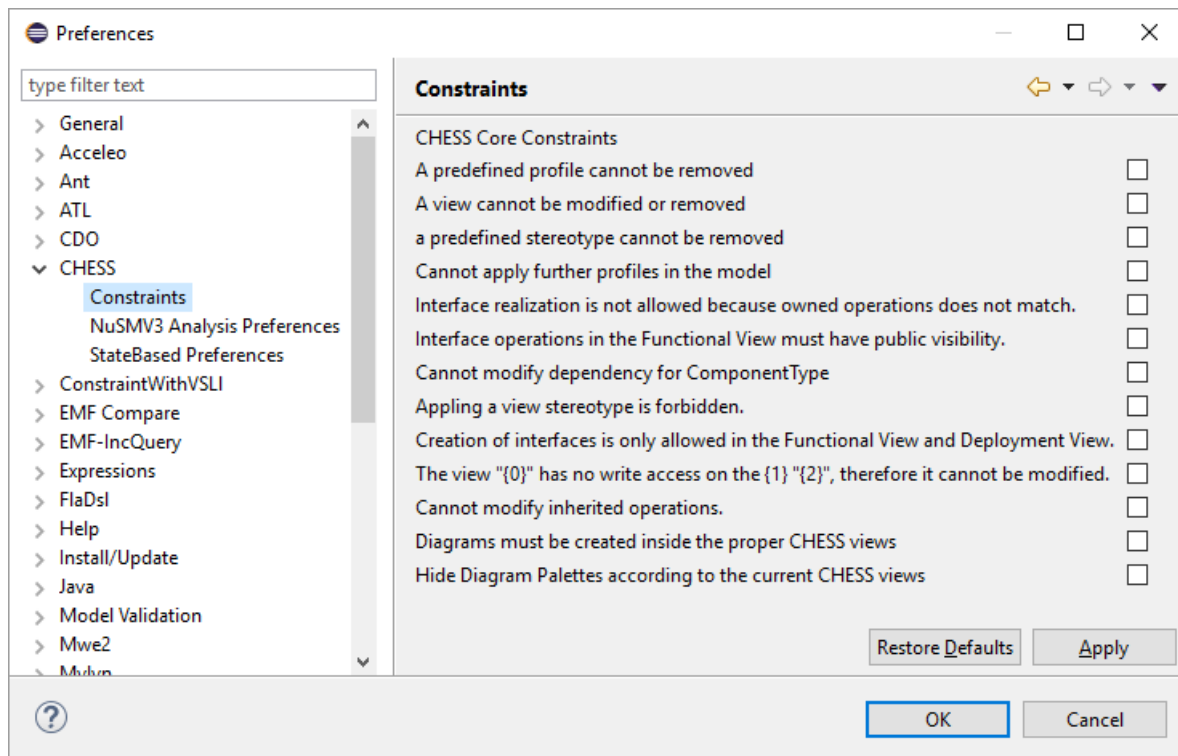
In particular, the Polarsys CHES project provides a set of *core* plugins that allow the application of the CHES methodology ([1][2]). This is the base upon which the AMASS methodology will build. The actual CHES methodology allows the design, verification and implementation of cyber physical software systems; CHES adopts a dedicated component model language [4] and ad-hoc model transformations to enable timing/dependability analysis and code generation. Moreover, the CHES methodology defines a multi-view approach for modelling the different aspects/concerns of the system; for each view, the diagrams and entities that can be created/viewed/modified are fixed and formalized in the view definition. The CHES plugins extend the Papyrus editor to support the CHES modelling language and design-by-view approach; so, by using the CHES Papyrus extension, the constraints imposed by the CHES methodology are enforced in a live-manner, at modelling time, to avoid late discovery of modelling activities which can violate the correctness-by-construction approach implemented by CHES.

The CHES-based AMASS Contract plugins use some utilities provided by other core CHES plugins; in detail, the core CHES plugins used are:

- `org.polarsys.chess.core`: provides some facilities regarding selections and diagram status.
- `org.polarsys.chess.services`: provides functionalities about the CHES editor (as extension of the Papyrus one).
- `org.polarsys.chess.validation`: provides functionalities about model validation.
- `org.polarsys.chessmlprofile`: provides the SysML/UML/MARTE profile implementation of the CHES modelling language [3]. Moreover, it provides dedicated diagram palettes extending the Papyrus ones to easily manage the creation of CHES stereotypes in a given diagram. Therefore, CHES core plugins are required in order to use the CHES Contract feature.

In order to allow the AMASS platform's stakeholders to use the CHES-based AMASS Contract features on top of the Papyrus editor without having to use the CHES methodology for SW development, an extension has been made to the CHES core plugins. In particular, the user can decide to disable the live-check of the constraints associated to the CHES multi-views support; in this way, the modeller can use the full Papyrus and UML features, together with the CHES extension for contract based design.

Figure 52 below provides a snapshot of the CHES methodology constraints that can be enabled/disabled through the Eclipse preferences page.



**Figure 52.** CHES methodology constraint

## 4.2.2 Architecture-Driven Assurance Block

### 4.2.2.1.1 Requirements Formalization with Temporal Logics – RQA approach

To create the custom-coded metric needed to detect linear temporal logic consistency issues in the requirements is necessary to add the following information to RQA:

- Assembly: the .DLL file generated after building the project. It is necessary to include the entire path of the .DLL file or add it into the RQA installation path.
- Class: the name of the class in the project. RQA should help you choosing this field using the provided assembly.
- Method: the name of the method containing the code of the metric. RQA should help you choosing this field using the provided class.

### 4.2.2.1.2 Simulation-based Fault Injection

As specified in Section 2.2.3.5, MASSIF Simulink Integration Framework for Eclipse is used for accessing Simulink model information. It has to be mentioned that importing is done using the command line interface of Matlab and not directly parsing mdl or slx files. This is the API recommended by Mathworks for accessing Simulink model information.

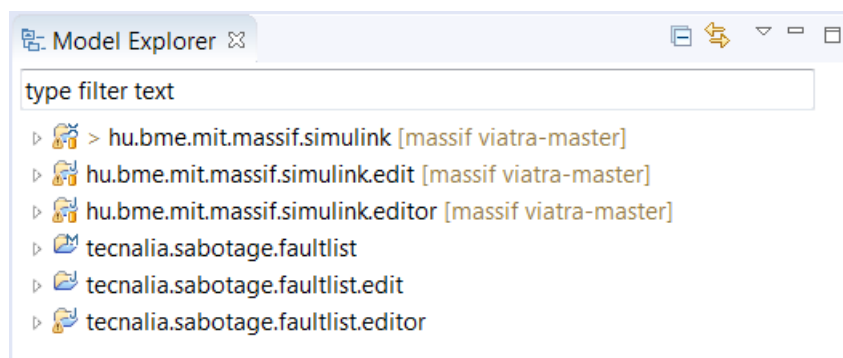
Following the procedure explained in [<https://github.com/viatra/massif>], the user needs to install MASSIF on its Eclipse Neon environment. The most remarkable prerequisites are the following:

1. Clone the Massif "Master" branch from <https://github.com/viatra/massif> (Massif 0.6.0).

2. Install VIATRA Query and Transformation SDK 1.5.0 from <http://download.eclipse.org/viatra/updates/release/>. It is important to remark that there is a dependency between VIATRA and EMF. To avoid any incompatibilities with the VIATRA version, EMF 2.12 must be installed.
3. Install Xtext Complete SDK 2.10.

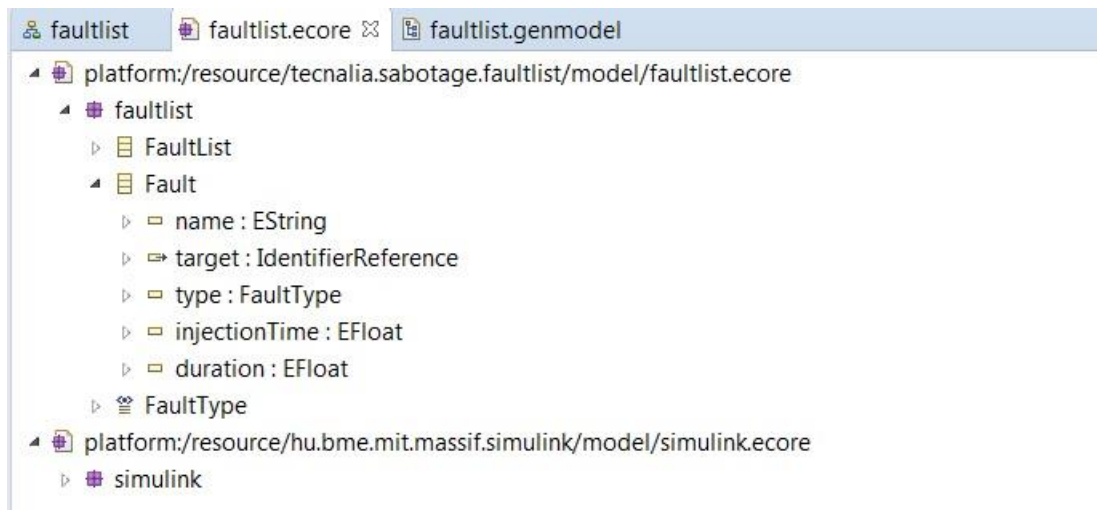
Another important point to have into account is the one referring to association between different meta-models. Especially between Sabotage and Massif meta-models.

- `tecnalia.sabotage.ecore` → Sabotage meta-model defines all the faults injected in the Simulink model. This meta-model is currently under development.
- `hu.bme.mit.massif.simulink` → Massif meta-model is designed to store all information for each MATLAB block.



**Figure 53.** Massif and Sabotage meta-models

Load resource functionality needs to be carried out at both meta-model and model level in order to establish the connection.



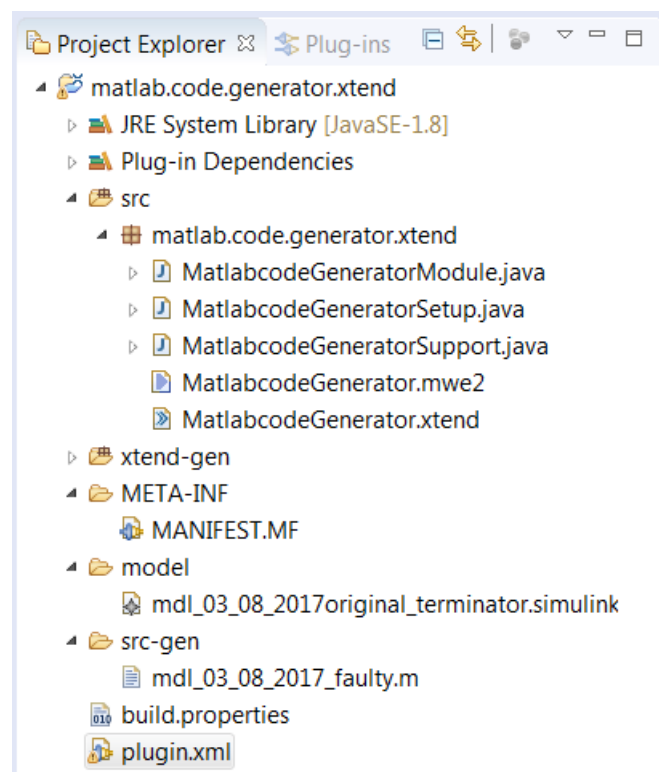
**Figure 54.** Connection between Sabotage and Massif meta-models

In order to perform the code generation activity by means of Xtend (cf. Section), those meta-models are included as dependency plugins. Together with these plugins, other ones are required as well:

- `hu.meb.mit.massif.simulink`: provides the generated java files from Massif EMF meta-model. Massif meta-model is designed to store all information for each MATLAB block.

- `tecnalia.sabotage.faultlist`: provides the generated java files from Sabotage EMF meta-model. This meta-model defines all the faults injected in the Simulink model.
- `org.eclipse.core.runtime`: provides support for the runtime platform, core utility methods and the extension registry.
- `org.eclipse.xtext.generator`: provides Generator facilities for Xtext.
- `org.eclipse.emf.mwe2.launch`: MWE2 (Modeling Workflow Engine) allows to compose object graphs declaratively in a very compact manner.
- `org.eclipse.emf.mwe2.language.ui`: provides user interface for MWE2 facilities.
- `org.apache.log4j`: provides most of the logging operations, except configurations.
- `org.apache.commons.logging`: provides a Log interface that is intended to be both lightweight and an independent abstraction of other logging toolkits. It provides the middleware/tooling developer with a simple logging abstraction allowing the user (application developer) to plug in a specific logging implementation.

All of them are included in the manifest file.



**Figure 55.** Code Generation workspace

#### 4.2.2.1.3 Fault Trees generation

The plugin to visualize fault tree is named `eu.fbk.eclipse.standardtools.faultTreeView` and does not depend on CHES, it is an Eclipse plugin located in the git repository: <https://gitlab.fbk.eu/adebiasi/EST>. It

is derived from the open source tool EMFTA<sup>17</sup>, to embed the fault tree viewer inside the CHES platform. The plugin requires the Sirius Eclipse tool<sup>18</sup> to work correctly.

#### **4.2.2.2 Metrics**

##### **4.2.2.2.1 Metrics for requirements**

###### **4.2.2.2.1.1 Correctness metrics**

Following, the information relative to the functions of the source code is presented. For each metric two function are specified: one relative to numerical value and other to feature information.

#### ***Metric to nouns***

##### **In-System Conceptual Model Nouns (SCM Nouns)**

ScmNounCount: return a double with the result of the metric.

ScmNounFeatures: return a list of strings with the resultant features of the metric.

##### **Out-of-System Conceptual Model Nouns (Out-of-SCM Nouns)**

OutOfScmNounCount: return a double with the result of the metric.

OutOfScmNounFeatures: return a list of strings with the resultant features of the metric.

##### **In-Semantic Clusters Nouns (SCC Nouns)**

SccNounCount: return a double with the result of the metric.

SccNounFeatures: return a list of strings with the resultant features of the metric.

##### **Out-of-Semantic Clusters Nouns (Out-of-SCC Nouns)**

OutOfSccNounCount: return a double with the result of the metric.

OutOfSccNounFeatures: return a list of string with the resultant features of the metric.

##### **In-Hierarchical Views Nouns (SCV Nouns)**

ScvNounCount: return a double with the result of the metric.

ScvNounFeatures: return a list of strings with the resultant features of the metric.

##### **Out-of-Hierarchical Views Nouns (Out-of-SCV Nouns)**

OutOfScvNounCount: return a double with the result of the metric.

OutOfScvNounFeatures: return a list of strings with the resultant features of the metric.

#### ***Metric to verbs***

##### **In-System Conceptual Model Verbs (SCM Verbs)**

---

<sup>17</sup> <https://github.com/juli1/emfta>

<sup>18</sup> <https://eclipse.org/sirius/>



ScmVerbCount: return a double with the result of the metric.

ScmVerbFeatures: return a list of string with the resultant features of the metric.

#### **Out-of-System Conceptual Model Verbs (Out-of-SCM Verbs)**

OutOfScmVerbCount: return a double with the result of the metric.

OutOfScmVerbFeatures: return a list of strings with the resultant features of the metric.

#### **In-Semantic Clusters Verbs (SCC Verbs)**

ScvVerbCount: return a double with the result of the metric.

ScvVerbFeatures: return a list of strings with the resultant features of the metric.

#### **Out-of-Semantic Clusters Verbs (Out-of-SCC Verbs)**

OutOfScvVerbCount: return a double with the result of the metric.

OutOfScvVerbFeatures: return a list of strings with the resultant features of the metric.

#### **In-Hierarchical Views Verbs (SCV Verbs)**

ScvVerbCount: return a double with the result of the metric.

ScvVerbFeatures: return a list of strings with the resultant features of the metric.

#### **Out-of-Hierarchical Views Verbs (Out-of-SCV Verbs)**

OutOfScvVerbCount: return a double with the result of the metric.

OutOfScvVerbFeatures: return a list of strings with the resultant features of the metric.

#### **4.2.2.2.2 Applying machine learning to improve the quality of requirements**

This functionality is implemented in an external tool and uses libraries of RQA.

#### **4.2.2.2.3 Metrics for models**

The structure of the source code previously implemented to generate the completeness and consistency metric relative to requirements has been adapted to evaluate quality of the models. The information extracted of the models is processed by the metrics to evaluate the quality. Following is showed the function of each metric.

Completeness

- Terminology coverage:  
TerminologyCoverageMetric\_Evaluation:
- Relationships from SCM View Coverage  
SCMCoverageMetric\_Evaluation
- Relationship types coverage  
RelationshipTypeCoverageMetric\_Evaluation
- Model-content coverage  
ModelContentCoverageMetric\_Evaluation
- Properties coverage



#### PropertiesCoverageMetric\_Evaluation

##### Consistency

- Property values  
PropertiesConsistencyMetric\_Evaluation
- Arithmetic operation compliance with SCM  
ArithmeticOperationConsistencyMetric\_Evaluation
- Overlapping requirements  
OverlappingConsistencyMetric\_Evaluation
- Measurement units for specific property  
MeasurementUnitsSpecificPropertyConsistencyMetric\_Evaluation

#### **4.2.2.2.4 Quality evolution (with respect to time)**

The quality evolution is represented with three general functions: save snapshot, show graphical quality evolution, and open snapshot information.

CreateAndSaveSnapshot: this function creates one snapshot with the quality information of the project.

LoadQualityEvolutionView: this function recovers the quality value of the snapshot saved in the project and a graphical area chart is showed.

LoadQualityEvolutionSnapshot: this function shows the information contained in one snapshot.

#### **4.2.2.3 Contract-Based Assurance Composition**

The plugins that need to be installed on top of the CHESSE environment to enable the editor with content assist are the following:

- org.polarsys.chess.contracts.contractEditor: it provides a contract editor with content assist. It enables also the possibility to create a new contract directly from the editor view.
- org.polarsys.chess.constraints.constraintEditor: it provides a constraint editor with content assist.
- org.polarsys.chess.properties.propertyEditor: it provides a property editor with content assist.

The CHESSE plugins to enable different hierarchical views based on contracts are the following:

- org.polarsys.chess.contracts.hierarchicalContractView: it provides a view that shows the decomposition of the system component into sub-components. It also shows the contracts assigned for each component.
- org.polarsys.chess.contracts.refinementView: it provides a view that shows the contracts with their refining contracts.

The complete set of contract-based analysis is provided by the following plugin:

- org.polarsys.chess.ocraService: it provides different analysis command invoking the OCRA tool.

They depend on a set of Eclipse plugins that are available at the following source code repository: <https://gitlab.fbk.eu/adebiasi/EST>. The plugins are:

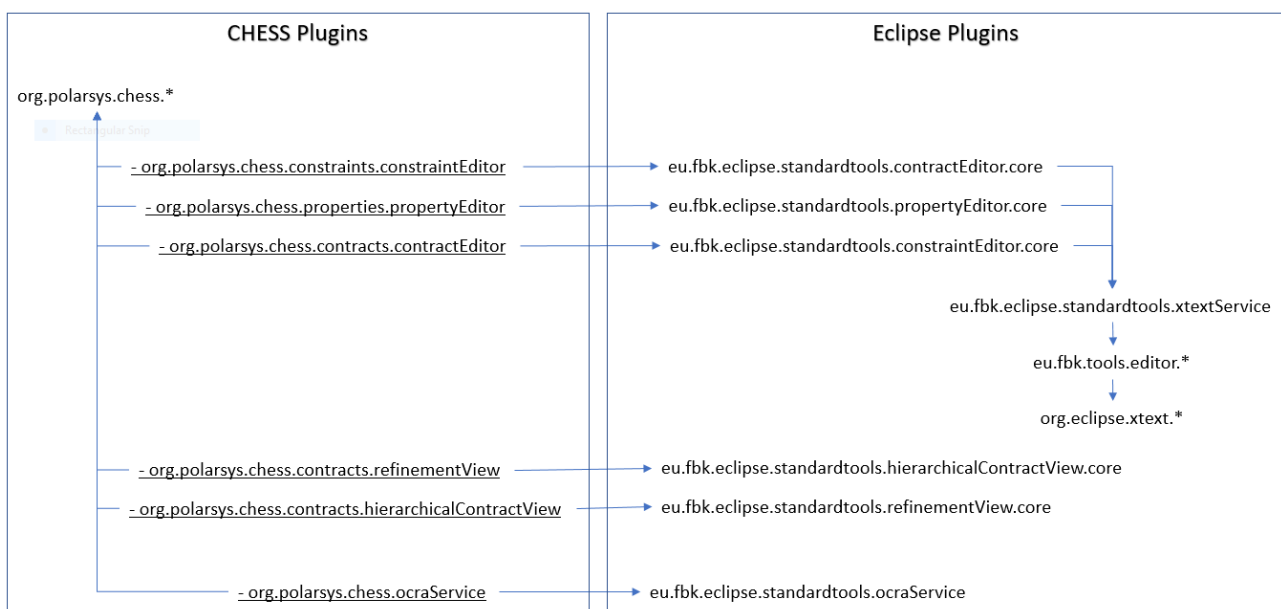
- eu.fbk.eclipse.standardtools.contractEditor.core: it contains the core functionalities used in the CHESSE plugin contractEditor.

- `eu.fbk.eclipse.standardtools.constraintEditor.core`: it contains the core functionalities used in the CHESSE plugin `constraintEditor`.
- `eu.fbk.eclipse.standardtools.propertyEditor.core`: it contains the core functionalities used in the CHESSE plugin `propertyEditor`.
- `eu.fbk.eclipse.standardtools.hierarchicalContractView.core`: it contains the core functionalities used in the CHESSE plugin `hierarchicalContractView`.
- `eu.fbk.eclipse.standardtools.refinementView.core`: it contains the core functionalities used in the CHESSE plugin `refinementView`.
- `eu.fbk.eclipse.standardtools.xtextService`: it contains the core functionalities used in the three CHESSE plugins editors.
- `eu.fbk.eclipse.standardtools.ocraService`: it contains the core functionalities used in the CHESSE plugin `ocraService`.

There are some Eclipse plugins used by the Eclipse plugins as external libraries but not implemented in the project. They can be installed from the following Eclipse update site [http://es-static.fbk.eu/tools/amass\\_sde](http://es-static.fbk.eu/tools/amass_sde). The available plugins are:

- `eu.fbk.tools.editor.*`: plugins provided by FBK that enrich a text area with content assist for an LTL grammar.
- `org.eclipse.xtext.*`: xText library is needed for of the editor plugins.

Figure 56 shows more in details the dependences among the plugins. The set `org.polarsys.chess.*` are the plugins described in Section 4.2.1.



**Figure 56.** Diagram showing the dependences among the plugins. The direction of the arrow means that the origin plugin depends on the target plugin

## 5. Conclusions

This deliverable D3.5 “Prototype for Architecture-Driven Assurance (b)” is the second output of the AMASS task T3.3 Implementation for Architecture-driven Assurance, whose objective is the development of a tooling framework to support architecture-driven assurance.

With three planned prototype iterations for the framework; this deliverable reports the status for the second prototype release (Prototype P1), in particular for the system component specification and the tooling framework supporting architecture-driven assurance, by describing the supported functionalities and the details about implementation.

Insofar, all partners made significant progress with the implementation of their individual features and functionalities.

The remaining deliverable D3.6 “Prototype for architecture-driven assurance (c)” will strongly focus in the integration of different approaches and ideas into one unified AMASS tooling framework supporting architecture-driven assurance.

## Abbreviations

<i>Abbreviation</i>	<i>Explanation</i>
AADL	Architecture Analysis and Design Language
API	Application Programming Interface
BNF	Backus-Naur Form
CACM	Common Assurance and Certification Meta-model
CHESSML	CHESS Modelling Language
CPS	Cyber Physical System
ECSEL	Electronic Components and Systems for European Leadership
EMF	Eclipse Modeling Framework
GUI	Graphical User Interface
IDE	Integrated Development Environment
JU	Joint Undertaking
LTL	Linear Temporal Logic
NLP	Natural Language Processing
MARTE	Modeling and Analysis of Real Time and Embedded systems
OCRA	Othello Contracts Refinement Analysis
OMG	Object Management Group
OSLC	Open Services for Lifecycle Collaboration
OSS	OCRA System Specification
RQA	Requirement Quality Analyzer
RSHP	RelationSHiP
SCC	Semantic Clusters
SCM	System Conceptual Model
SCV	Hierarchical Views
SMUT	System Model Under Test
SMV	
SW	Software
SysML	System Modelling Language
TRL	Technology Readiness Level
UML	Unified Modelling Language
V&V	Verification and Validation



WP	Work Package
XMI	XML Metadata Interchange
XSAP	eXtended Safety Assessment Platform

## References

- [1] Mazzini S., J. Favaro, S. Puri, L. Baracchi., “CHESS: an open source methodology and toolset for the development of critical systems”, 2nd International Workshop on Open Source Software for Model Driven Engineering (OSS4MDE), Saint-Malo, October 2016
- [2] L.Baracchi, S.Mazzini, S.Puri, T.Vardanega: “Lessons Learned in a Journey Toward Correct-by-Construction Model-Based Development”, Reliable Software Technologies – Ada-Europe 2016 Volume 9695 of the series Lecture Notes in Computer Science pp 113-128, 31 May 2016
- [3] <https://www.polarsys.org/chess/publis/CHESSMLprofile.pdf>
- [4] CONCERTO ARTEMIS JU project, D2.2 The CONCERTO Component Model, 9 May 2014, available at <http://www.concerto-project.org/results>
- [5] Papyrus Eclipse project: <https://eclipse.org/papyrus/>
- [6] AMASS D2.3 AMASS Reference Architecture (b) deliverable, 29 September 2017
- [7] M. dos Santos Soares, J. Vrancken: “Model-Driven User Requirements Specification using SysML”, JOURNAL OF SOFTWARE, VOL. 3, No. 6, June 2008
- [8] XML Metadata Interchange, [www.omg.org/spec/XMI/](http://www.omg.org/spec/XMI/)
- [9] [AMASS D3.1 Baseline and requirements for architecture-driven assurance](#), 30 September 2016
- [10] CONCERTO D3.3 – Design and implementation of analysis methods for non-functional properties - Final version, 18 November 2015, Public Distribution, <http://www.concerto-project.org/results>
- [11] Acacia+, <http://lit2.ulb.ac.be/acaciaplus/>
- [12] Xtend, <https://eclipse.org/xtend/documentation/2.7.0/Xtend%20User%20Guide.pdf>
- [13] Massif Ecore description, <https://github.com/viatra/massif/tree/master/plugins/hu.bme.mit.massif.simulink/model>
- [14] AMASS D3.2 Design of the AMASS tools and methods for architecture-driven assurance (a) deliverable, 30 June 2017
- [15] Massif: MATLAB Simulink Integration Framework for Eclipse, <https://github.com/viatra/massif>
- [16] AMASS D2.1 Business cases and high-level requirements, 28 February 2017
- [17] AMASS D2.2 AMASS Reference Architecture (a) deliverable, 30 November 2016