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 (c)
D3.6

Work Package:	WP3: Architecture-Driven Assurance
Dissemination level:	PU = Public
Status:	Final
Date:	31st August 2018
Responsible partner:	B&M
Contact information:	Peter M. Kruse <peter.kruse@berner-mattner.com>
Document reference:	AMASS_D3.6_WP3_B&M_V1.0

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, Markus Grabowski	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)
Stefano Tonetta, Alberto Debiasi	Fondazione Bruno Kessler (FBK)
Garazi Juez, Estibaliz Amparan	Tecnalia Research & Innovation (TEC)
Tomáš Kratochvíla, Vít Koksa	Honeywell (HON)
Jaroslav Bendík	Masaryk University (UOM)

Reviewers

Names	Organisation
Marc Sango (Peer review)	ALL4TEC (A4T)
Zoë Stephenson (Peer review)	Rapita Systems (RPT)
Alejandra Ruiz (TC review)	Tecnalia Research & Innovation (TEC)
Cristina Martinez (Quality Manager)	Tecnalia Research & Innovation (TEC)

TABLE OF CONTENTS

List of Figures

Executive Summary

The deliverable D3.6 “Prototype for Architecture-Driven Assurance (c)” is the last 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. D3.6 is the evolution of D3.5, which described the second prototype, the sections modified with respect to D3.5 have been marked with (*) in the headlines.

AMASS task T3.3 has three prototype iterations, described in D3.4 [20] , D3.5 [21] and D3.6 (this document). This deliverable reports the status of the aforementioned tooling framework for the final prototype release (Prototype P2), 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 has been selected from the AMASS deliverable D2.1 “Business cases and high-level requirements” [18]. D3.6 describes the technologies that allow the implementation of all selected functionality also covering requirements which have not been implemented in previous prototype iterations.

The logical structural view of the AMASS reference tool architecture elaborated in the “AMASS Reference Architecture” deliverables, D2.3 [6] and D2.4 [7], have also been considered in this deliverable; in particular physical components like CHESS and its contract editing functionality have been successfully mapped to the logical tool components *Component Editor* and *Contract Editor*.

WP4 and WP5 results have been particularly useful in guiding the argumentation and evidence metamodel specification; importantly, system architecture-related information can now be traced to the argumentation and evidence models.

The deliverable D3.6 “Prototype for architecture-driven assurance (c)” is the final evolution of this deliverable; in particular, D3.6 documents the final state of the tooling framework’s implementation supporting architecture-driven assurance using contract based design.

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 OPENCOSS[1] and SafeCer[2] 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 OPENCOSS 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)[3] 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[4] is a strong candidate to host AMASS (See D7.3 [27], D7.5 [28] and D7.6 [29] 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.

[1] www.opencoss-project.eu

[2] https://artemis-ia.eu/project/40-nsafecer.html

[3] https://open-services.net

[4] www.polarsys.org

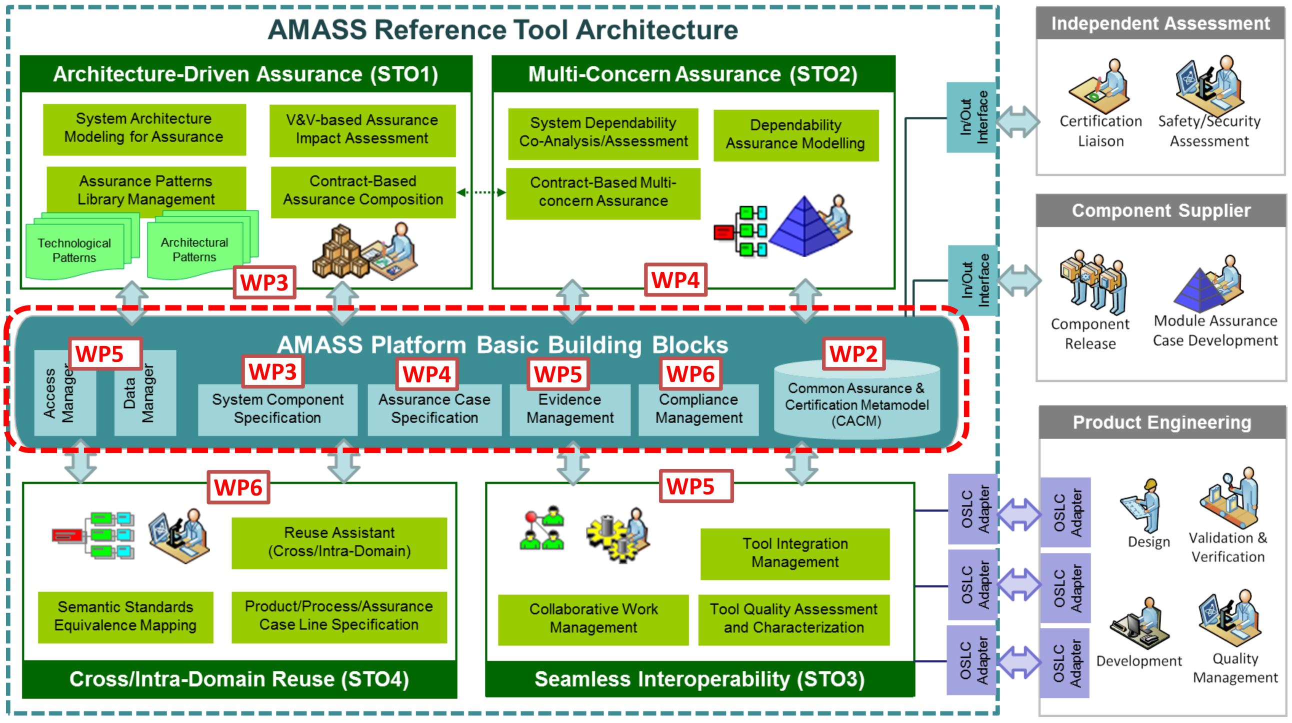

AMASS Reference Tool Architecture
Architecture-Driven Assurance (STO1)
System Architecture Modeling for Assurance
V&V-based Assurance Impact Assessment
Assurance Patterns Library Management
Contract-Based Assurance Composition
Technological Patterns
Architectural Patterns
WP3
Multi-Concern Assurance (STO2)
System Dependability Co-Analysis/Assessment
Dependability Assurance Modelling
Contract-Based Multi-concern Assurance
WP4
AMASS Platform Basic Building Blocks
Access Manager
Data Manager
WP5
WP3
System Component Specification
WP4
Assurance Case Specification
WP5
Evidence Management
WP6
Compliance Management
WP2
Common Assurance & Certification Metamodel (CACM)
WP6
Reuse Assistant (Cross/Intra-Domain)
Semantic Standards Equivalence Mapping
Product/Process/Assurance Case Line Specification
Cross/Intra-Domain Reuse (STO4)
WP5
Tool Integration Management
Collaborative Work Management
Tool Quality Assessment and Characterization
Seamless Interoperability (STO3)
In/Out Interface
In/Out Interface
OSLC Adapter
Independent Assessment
Certification Liaison
Safety/Security Assessment
Component Supplier
Component Release
Module Assurance Case Development
Product Engineering
Design
Validation & Verification
Development
Quality Management

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 prototype 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 P2, WP3 is responsible for driving the 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 [10] Section 3.1.1).

This deliverable follows the outcomes of D3.5 [21], which comprised a thorough report on 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. In that framework, D3.6 strongly focuses on the integration of different approaches and ideas into one unified AMASS tooling framework supporting architecture-driven assurance.

Other important parts of the D3.6 document are:

- Description of the AMASS Platform tools for the final prototype
- Finalized User Manuals and installation Instructions
- Source code description

2. Implemented Functionality

2.1 Scope (*)

The scope for the third 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 red rectangles on Figure 2, which shows the general layered structure of the AMASS platform (from AMASS deliverable D2.3 [19]).

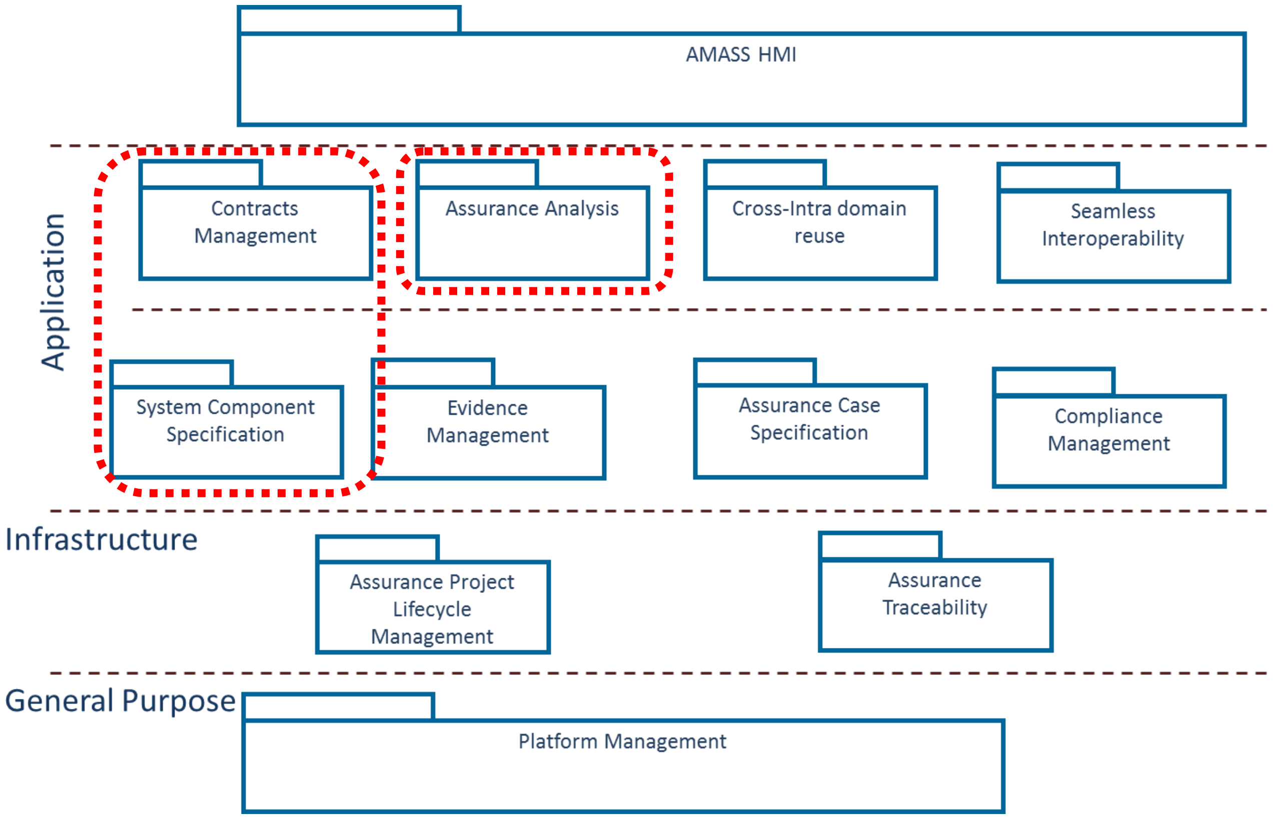


Figure 2. Layered structure of AMASS tool modules

Figure 3 illustrates the component decomposition of these tools based on the design specification documented in deliverable D3.3 [16].

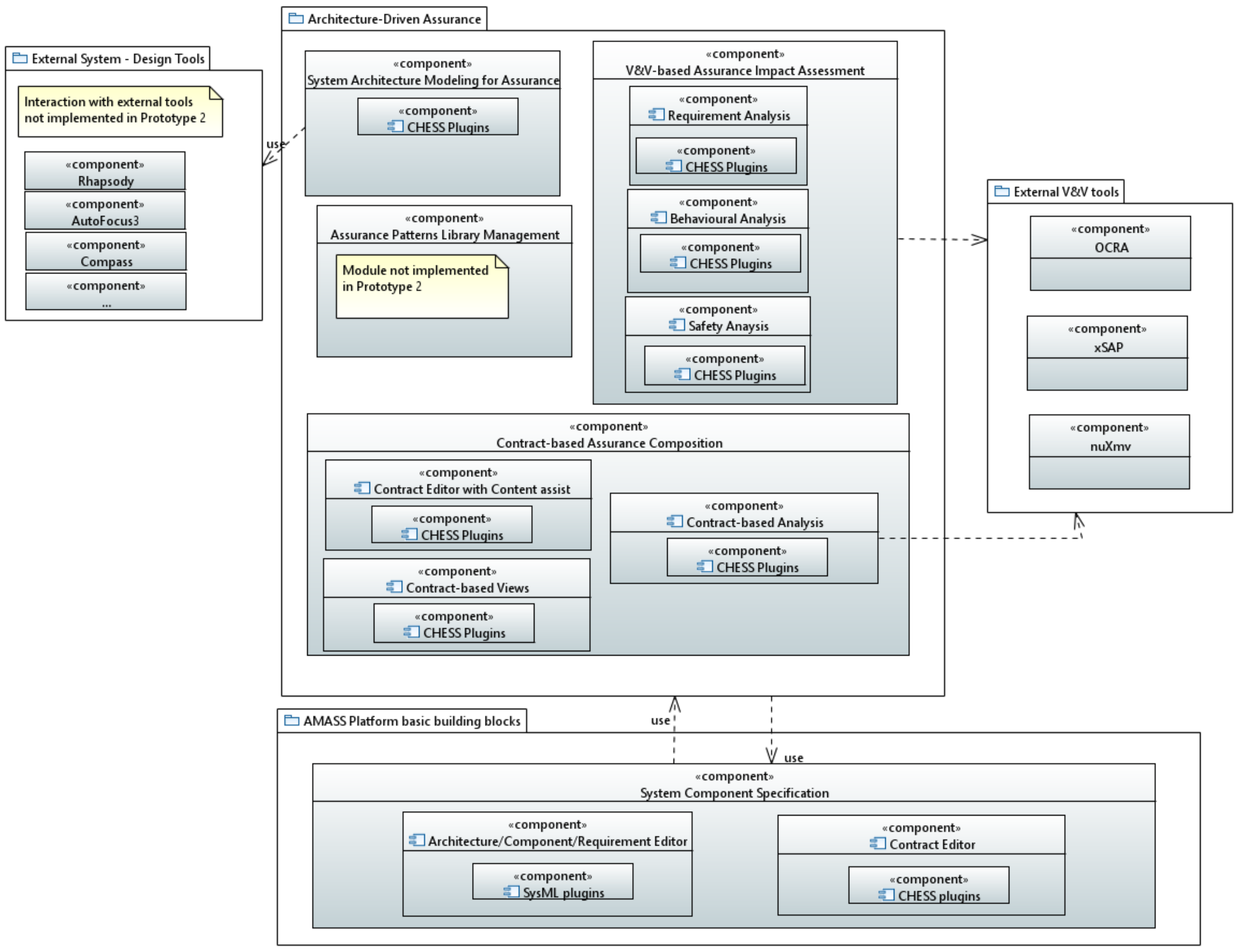

Architecture-Driven Assurance
External System - Design Tools
Interaction with external tools not implemented in Prototype 2
«component» Rhapsody
«component» AutoFocus3
«component» Compass
«component» ...
use
«component» System Architecture Modeling for Assurance
«component» CHESS Plugins
«component» Assurance Patterns Library Management
Module not implemented in Prototype 2
«component» V&V-based Assurance Impact Assessment
«component» Requirement Analysis
«component» CHESS Plugins
«component» Behavioural Analysis
«component» CHESS Plugins
«component» Safety Anaysis
«component» CHESS Plugins
External V&V tools
«component» OCRA
«component» xSAP
«component» nuXmv
«component» Contract-based Assurance Composition
«component» Contract Editor with Content assist
«component» CHESS Plugins
«component» Contract-based Analysis
«component» CHESS Plugins
«component» Contract-based Views
«component» CHESS Plugins
AMASS Platform basic building blocks
use
use
«component» System Component Specification
«component» Architecture/Component/Requirement Editor
«component» SysML plugins
«component» Contract Editor
«component» CHESS plugins

Figure 3. Description of main building blocks

The System component specification enables the design of: the overall architecture, each single component and the requirements. Moreover, it provides features of contract editing. The architecture driven assurance is decomposed in different modules; the System Architecture Modelling for Assurance module that interacts with the External Design Tools, the V&V-based Assurance Impact Assessment module that provides V&V analysis invoking external V&V tools, the Contract-based Assurance Composition that provides contact-based features, and the Assurance Patterns Library Management module that implements the concept of assurance pattern.

2.2 Implemented Requirements (*)

From the requirements point of view, this last prototype iteration focuses on a set of AMASS requirements as defined in the AMASS deliverable D2.1 "Business cases and high-level requirements" [18]. Table 1 shows all relevant requirements which the final prototype shall implement. Even though some of the requirements are still pending or in development at the release of this document, the final prototype will cover all of them.

Table 1. Requirements implemented in the final prototype of the AMASS platform (P2)

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_APL_001	Drag and drop an architectural pattern	The system shall be able to instantiate in the component model and architectural pattern selected from the list of patterns stored	Solved	Papyrus	INT, TEC, CEA
WP3_APL_002	Edit an architectural pattern	The system should be able to edit, store and retrieve architectural patterns	Solved	Papyrus	INT, TEC, CEA
WP3_APL_003	Use of architectural patterns at different levels	The system shall be able to apply to the component model architectural patterns at different levels: AUTOSAR, IMA, Safety/Security Mechanisms (security controls)	Solved	Papyrus	INT, TEC, CEA (B&M)
WP3_APL_005	Generation of argumentation fragments from architectural patterns/decisions	The system shall be able to generate arguments fragments based on the usage of specific architectural patterns in the component model	Pending	OpenCert	TEC, CEA
WP3_CAC_001	Validate composition of components by validating their contracts	The system shall be able to validate the composition of components by supporting the validation of their contracts, analysing the relationship among assumptions and guarantees	Solved	CHESS, OCRA	FBK
WP3_CAC_002	Assign contract to component	The system shall allow to associate a contract to a component. Then, the system shall allow to drop a contract from a component	Solved	CHESS, SAVONA	MDH, FBK, B&M
WP3_CAC_003	Structure properties into contracts (assumptions/guarantees)	The system shall be able to support the extraction of assumptions and guarantees to be used in component contracts based on component properties	Solved	CHESS/SAVONA	FBK, B&M
WP3_CAC_004	Specify contract refinement	The system shall enable users to specify the refinement of the contract along the hierarchical component's architecture	Solved	CHESS/SAVONA	FBK, B&M
WP3_CAC_005	General management of contract-component assignments	The system should enable users to have a view of the association between contracts and components for the entire system architecture (thus, not only a view on the single contract assignment for each component)	Solved	CHESS	INT, FBK
WP3_CAC_006	Refinement-based overview	The system should enable users to have a hierarchical view of the contract refinements along the system architecture	Solved	CHESS, SAVONA	FBK, B&M

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_CAC_007	Overview of check refinements results	The system should enable users to have an overview in terms of status of check refinement of all the defined contracts.	Solved	CHESS	FBK
WP3_CAC_008	Contract-based validation and verification	The system must provide support for contract-based system validation and verification, including refinement checking, compositional verification of behavioural models, contract-based fault-tree generation	Solved	CHESS	FBK
WP3_CAC_009	Improvement of Contract definition process	The operation of contract definition should be improved in terms of time spent.	Solved	CHESS, SAVONA	FBK, B&M
WP3_CAC_011	Overview of contract-based validation for behavioural models	The system could enable users to have an overview of the validation of a contract over a state-machine. In case of failure, the system could enable users to have information about the trace that does not fulfil the contract.	Solved	CHESS	FBK
WP3_CAC_012	Browse Contract status	The user shall be able to browse the contracts associated within a component and their status (fulfilled or not)	Solved	CHESS	INT
WP3_CAC_013	Specify contracts defining the assumption and the guarantee elements	The system shall provide the capability to create a contract defining two new properties (assumptions/guarantees) implicitly associated to that contract.	Solved	CHESS	INT
WP3_CAC_014	Drop contract from component	The system shall allow to drop a contract from a component	Solved	CHESS, SAVONA	INT, B&M
WP3_CAC_015	Reassign contract to component	The system shall allow to substitute the already assigned contract to a component with another contract	Solved	CHESS	INT
WP3_SAM_001	Trace component with assurance assets	The supplier of a component shall be able to trace all the assurance information with the specific component	Solved	CAPRA	INT
WP3_SAM_002	Impact assessment if the component changes	The system shall provide the capability for a component change impact analysis	Pending	CAPRA	B&M, INT
WP3_SAM_003	Compare different architectures according to different concerns which haven't been specified before	The system shall be able to compare different system architectures based on predefined criteria, like dependability or timing concerns	Solved	CHESS	FBK

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_SAM_004	Integration with external modelling tools	The system could interact with external tools for system design and development (e.g., Rhapsody, AutoFocus, Compass) to get the system architecture.	Solved	CHESS, Papyrus	INT, UC3, TRC, FBK, B&M
WP3_SC_001	System abstraction levels	The user shall be able to browse along the different abstractions levels (system, subsystem, component)	Solved	CHESS, SAVONA	INT, B&M
WP3_SC_002	System abstraction levels	The user shall be able to move and edit along the different abstractions levels (system, subsystem, component)	Solved	CHESS, SAVONA	INT, B&M
WP3_SC_003	Modelling languages for component model	The system shall be able to support different modelling languages to model the component/Subsystem/system	Solved	CHESS, OCRA, SAVONA, Papyrus	FBK, B&M
WP3_SC_004	Formalize requirements with formal properties	The system shall be able to specify requirements about a component in a formal way	Solved	CHESS, SAVONA	INT, B&M
WP3_SC_005	Requirements allocation	The system shall provide the capability for allocating requirements to parts of the component model. More in general, requirements traceability shall be enabled.	Solved	CHESS, Papyrus, CAPRA	INT, KMT
WP3_SC_006	Specify component behavioural model (state machines)	The system shall be able to specify the component behavioural model	Solved	CHESS	FBK
WP3_SC_007	Fault injection (include faulty behaviour of a component)	The system shall have fault injection capabilities	Solved	CHESS, SABOTAGE	INT, TEC
WP3_VVA_001	Traceability between different kinds of V&V evidence	The system shall provide the ability to trace immediate evidence (obtained during the execution of the left-hand side of the V-model) with direct evidence (obtained during the execution of the right-hand side of the V-model). For instance: a contract-based, component-based specification should be traced with the corresponding analysis-results.	Solved	CAPRA	INT
WP3_VVA_002	Trace model-to-model transformation	The system shall be able to trace all component model transformations executed during V&V model-based analysis	Pending	CAPRA	INT
WP3_VVA_003	Validate requirements checking consistency, redundancy, … on formal properties	The system shall be able to validate formal requirements/properties	Solved	CHESS, OCRA, V&V Manager	FBK, HON, UOM
WP3_VVA_004	Trace requirements validation checks	The system shall be able to trace requirements validations	Solved	Papyrus, CAPRA	INT

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_VVA_005	Verify (model checking) state machines	The system shall be able to verify the component behavioural model match with the specification	Solved	CHESS, NuXmv, V&VManager	FBK, HON, UOM
WP3_VVA_006	Automatic provision of HARA/TARA-artifacts	The system shall provide the capability for automating HARA (Hazard Analysis Risk Assessment)/TARA (Threat Assessment & Remediation Analysis)-related artefacts (e.g., FTA, FMEA, attack trees.).	Solved	MediniAnalyze, SafetyArchitect, CHESS	B&M, KMT, A4T
WP3_VVA_007	Generation of reports about system description/ verification results ….	The system shall generate reports about system/subsystem/component verification results	Pending	CHESS, V&VManager	FBK, HON
WP3_VVA_010	Model-based safety analysis	The system shall allow the user to generate fault trees and FMEA tables from the behavioural model and the fault injection	Pending	CHESS, XSAP	INT, FBK
WP3_VVA_011	Simulation-based Fault Injection	The system should allow the user to generate fault injection simulations from the fault trees and FMEA tables	Pending	SABOTAGE	TEC, AIT, B&M
WP3_VVA_012	Design Space Exploration	The system could support the design space exploration of a system for a certain safety/security criticality level	Pending	CHESS	FBK

Each requirement together with the implementation completed so far to implement the requirement is briefly outlined in the following sections.

2.2.1 System Component Specification

2.2.1.1 System Architecture Editor (*)

Table 2. Requirements implemented in the System Architecture Editor

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_SC_001	System abstraction levels	The user shall be able to browse along the different abstractions levels (system, subsystem, component)	Solved	CHESS, SAVONA	INT, B&M
WP3_SC_002	System abstraction levels	The user shall be able to move and edit along the different abstractions levels (system, subsystem, component)	Solved	CHESS, SAVONA	INT, B&M
WP3_SC_003	Modelling languages for component model	The system shall be able to support different modelling languages to model the component/Subsystem/system	Solved	CHESS, OCRA, SAVONA, Papyrus	FBK, B&M

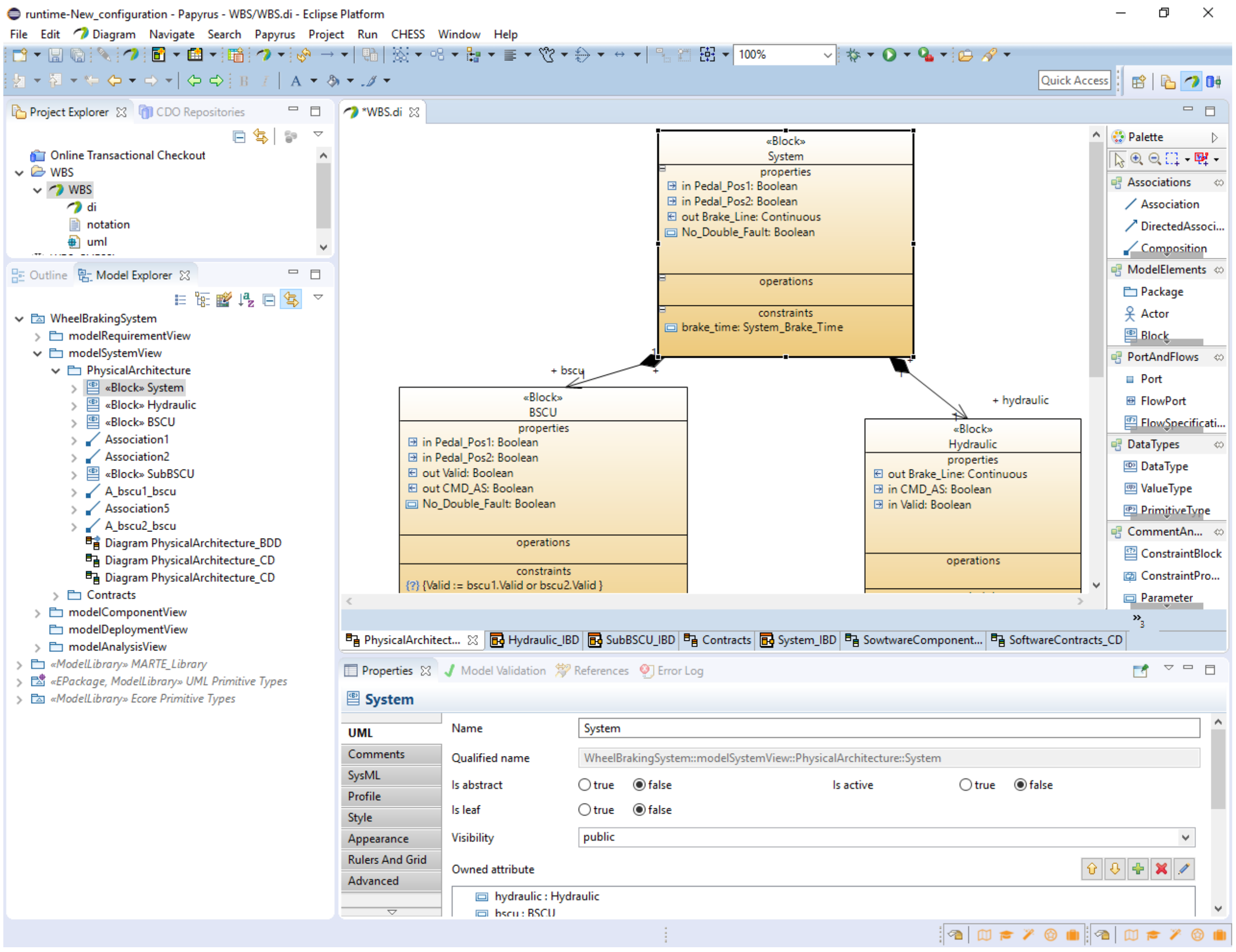
System architecture specification is supported by the Papyrus UML/SysML editor [5]. The selection of UML/SysML has been driven by the wide adoption of these modelling languages in the industry in different domains. Then, the selection of the Papyrus UML/SysML editor has been driven by the fact that Papyrus is an open source tool with very strict adherence to the OMG standards definition and very good support for customization (i.e. profiling), with also different successfully use case stories in the industry already available[5]. 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 also has integration facilities with other tools, such as the commercial IBM UML Rhapsody tool; in addition, it supports the XMI OMG standard [9] for the interchange of UML models between UML tools.

Through the 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 D2.4 [7]). Decomposition of blocks/components into sub-blocks/sub-components can be modelled by using internal block diagrams or composite structure diagrams. Both the Papyrus Editor and other AMASS components are under the same Open Source license, which supports the reuse of these previous results within the AMASS platform.

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

The resulting UML/SysML models and diagrams are stored in individual files in the Eclipse workspace.

[5] https://www.eclipse.org/papyrus/testimonials.html



System

Name	System
Qualified name	WheelBrakingSystem::modelSystemView::PhysicalArchitecture::System
Is abstract	true [✓ false]
Is active	true [✓ false]
Is leaf	true [✓ false]
Visibility	public
Owned attribute	hydraulic : Hydraulic; bscu : BSCU

Figure 4. Papyrus editor

The Papyrus UML editor supports the definition and application of UML profiles. In AMASS, the Papyrus tool is used together with the CHESS profile extension [3]; in particular CHESS 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 D2.4 [7]).

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 0).

For the GUI perspective, the CHESS theme enriches UML and SysML diagrams with useful information for the user, such as multiplicity attributes for ports and components, and guard expressions in state machines. The user may also hide graphical elements that cause visual clutter such as the stereotypes applied to the CHESS entities. CHESS enables the automatic generation of SysML diagrams from the CHESS model and provides a layout facility for arranging the diagram elements based on the Eclipse Layout Kernel (ELK)[12].

It is worth noting that the CHESS profile also provides other modelling capabilities, such as the dependability profile [11] for failure modelling and specific support for timing properties (see Section 0 about CHESS features). Moreover, CHESS provides a methodology for the design, verification and implementation of CPS SW systems [1]. The CHESS profile follows the same licensing approach as Papyrus and other AMASS components, which supports the easy integration of the developments from the intellectual property perspective.

2.2.1.1.1 Easy System Architecture Modelling with SAVONA (*)

As Papyrus and CHESS can be used for various different modelling activities their options and possibilities might overcome an average system engineer without an extensive background on applying SysML. It has been found that a restricted user interface, which only allows meaningful actions would result in a higher user acceptance. SAVONA has been developed to support system engineers in creating static system architectures in SysML.

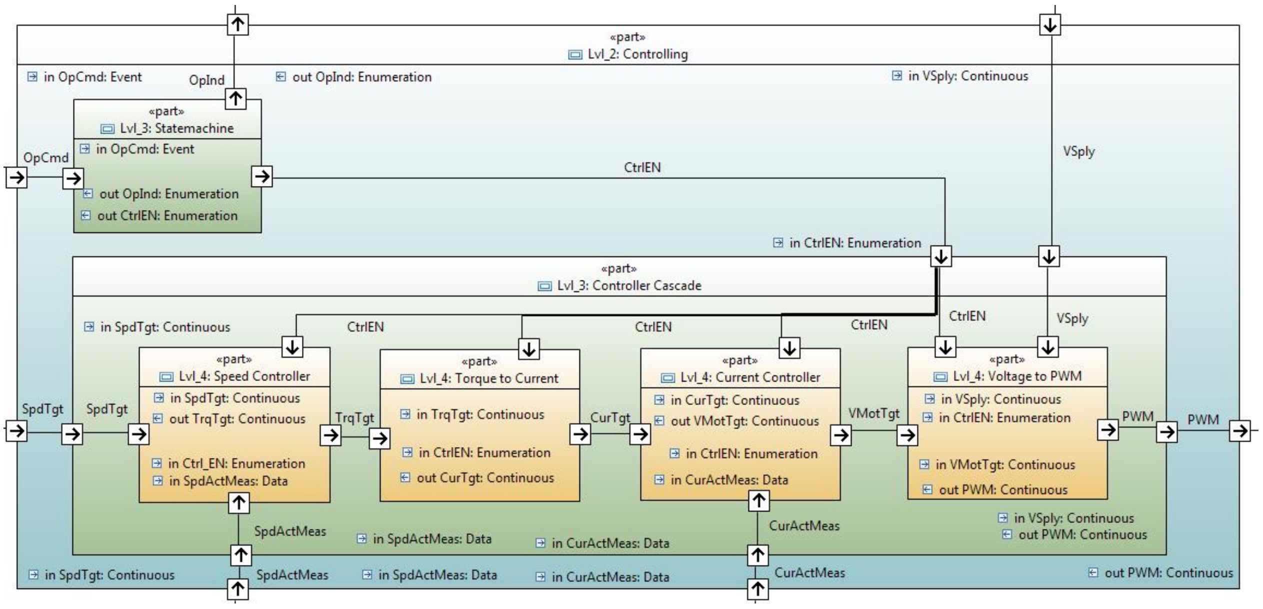


Figure 5. SysML IBD showing multiple system layers

The SysML Internal Block Diagram (IBD) has been chosen as the main diagram type for designing the systems architecture in SAVONA, as it allows an intuitive understanding of the model though multiple system layers. Mechanisms have been added that generate whitebox diagrams of selected system blocks, which can automatically be synchronized on model changes. This enables an effortless display of the model's current design status.

The Model Explorer is customized to only show relevant model elements such as blocks, parts, ports, signals and diagrams. Its context menu (see Figure 6) has been revised to only allow applicable copy/paste operations and to create new model elements and diagrams.

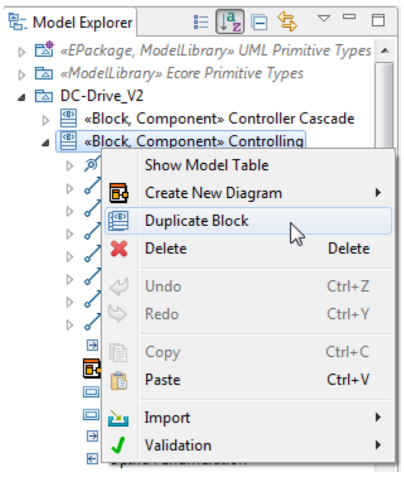
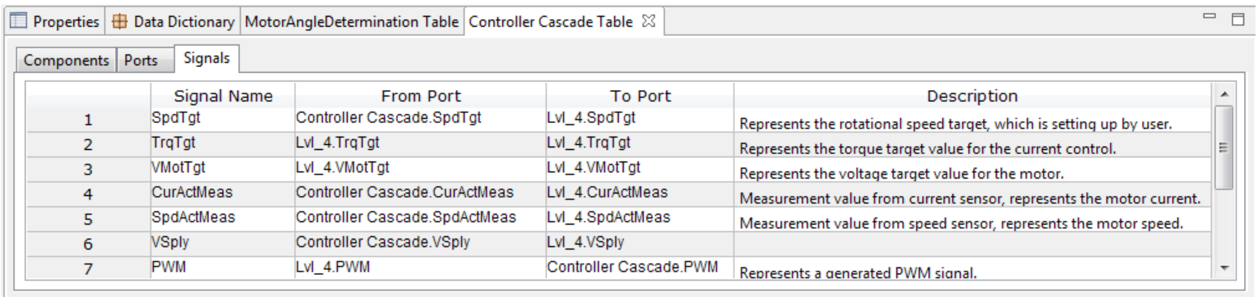


Figure 6. Revised Context Menu of the Model Explorer

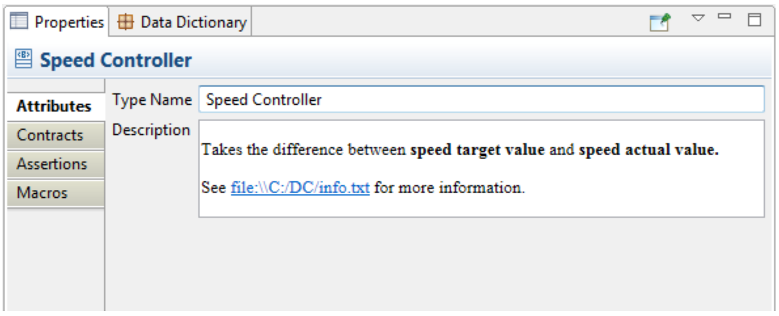
SAVONA also allows a tabular view of model elements. Such *Model Tables* can be viewed for each SysML block, containing the parts, ports and signals of the block. Figure 7 shows the Signals-tab containing all signals of the selected block. Model element names and descriptions can be edited directly inside the model table.



	Signal Name	From Port	To Port	Description
1	SpdTgt	Controller Cascade.SpdTgt	Lvl_4.SpdTgt	Represents the rotational speed target, which is setting up by user.
2	TrqTgt	Lvl_4.TrqTgt	Lvl_4.TrqTgt	Represents the torque target value for the current control.
3	VMotTgt	Lvl_4.VMotTgt	Lvl_4.VMotTgt	Represents the voltage target value for the motor.
4	CurActMeas	Controller Cascade.CurActMeas	Lvl_4.CurActMeas	Measurement value from current sensor, represents the motor current.
5	SpdActMeas	Controller Cascade.SpdActMeas	Lvl_4.SpdActMeas	Measurement value from speed sensor, represents the motor speed.
6	VSply	Controller Cascade.VSply	Lvl_4.VSply	
7	PWM	Lvl_4.PWM	Controller Cascade.PWM	Represents a generated PWM signal.

Figure 7. Model Table showing all signals of a SysML Block

The custom Properties View in SAVONA combines all relevant information for each model element. Element descriptions can be formatted and may contain hyperlinks as shown in Figure 8. The view also contains sections for specifying contracts, which is further described in section 2.2.1.5.1.



Type Name: Speed Controller

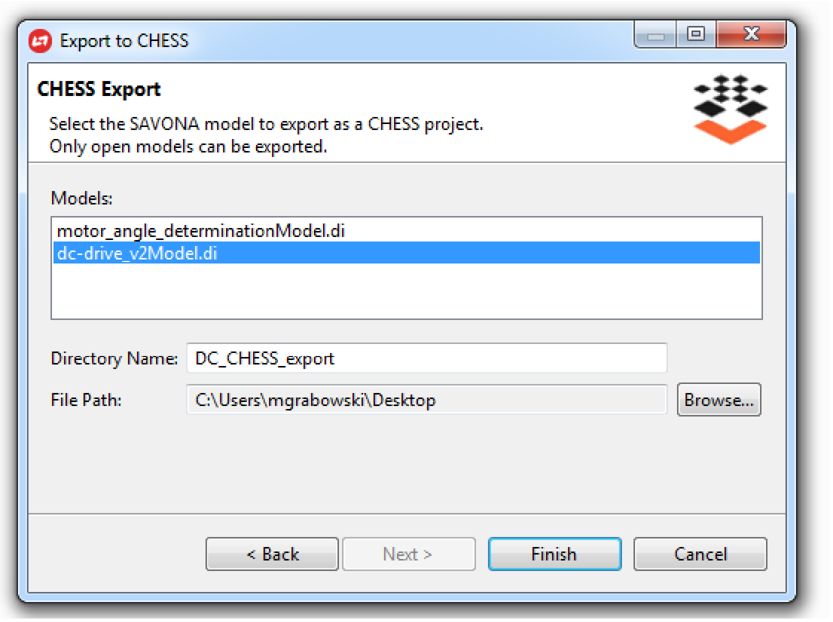
Description: Takes the difference between **speed target value** and **speed actual value.**

See file:\\C:/DC/info.txt for more information.

Figure 8. Properties View in SAVONA

Modelling a system's architecture often results in large diagrams that are difficult to lay out by hand. Based on the Eclipse Layout Kernel (ELK)[12] SAVONA offers the automatic layout of SysML IBD, which simplifies the laying out of new diagrams or parts of it.

Since SAVONA is also based on Papyrus it offers possibilities for interoperability to the AMASS Platform / CHESS. A CHESS export function allows conversion of the SAVONA model into a CHESS model (see Figure 9). That way, the initial architecture design can be performed in SAVONA and later be reopened in CHESS to perform various V&V activities on the model without any loss of information.


Export to CHESS

CHESS Export

Select the SAVONA model to export as a CHESS project.
Only open models can be exported.

Models:
motor_angle_determinationModel.di
dc-drive_v2Model.di

Directory Name: DC_CHESS_export

File Path: C:\Users\mgrabowski\Desktop

Figure 9. CHESS Export function of SAVONA

2.2.1.2 Formalize Requirements with Formal Properties (*)

Table 3. Requirements regarding the formalization of system requirements

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_SC_004	Formalize requirements with formal properties	The system shall be able to specify requirements about a component in a formal way	Solved	CHESS, SAVONA	INT, B&M
WP3_SC_005	Requirements allocation	The system shall provide the capability for allocating requirements to parts of the component model. More in general, requirements traceability shall be enabled.	Solved	CHESS, Papyrus, CAPRA	INT, KMT

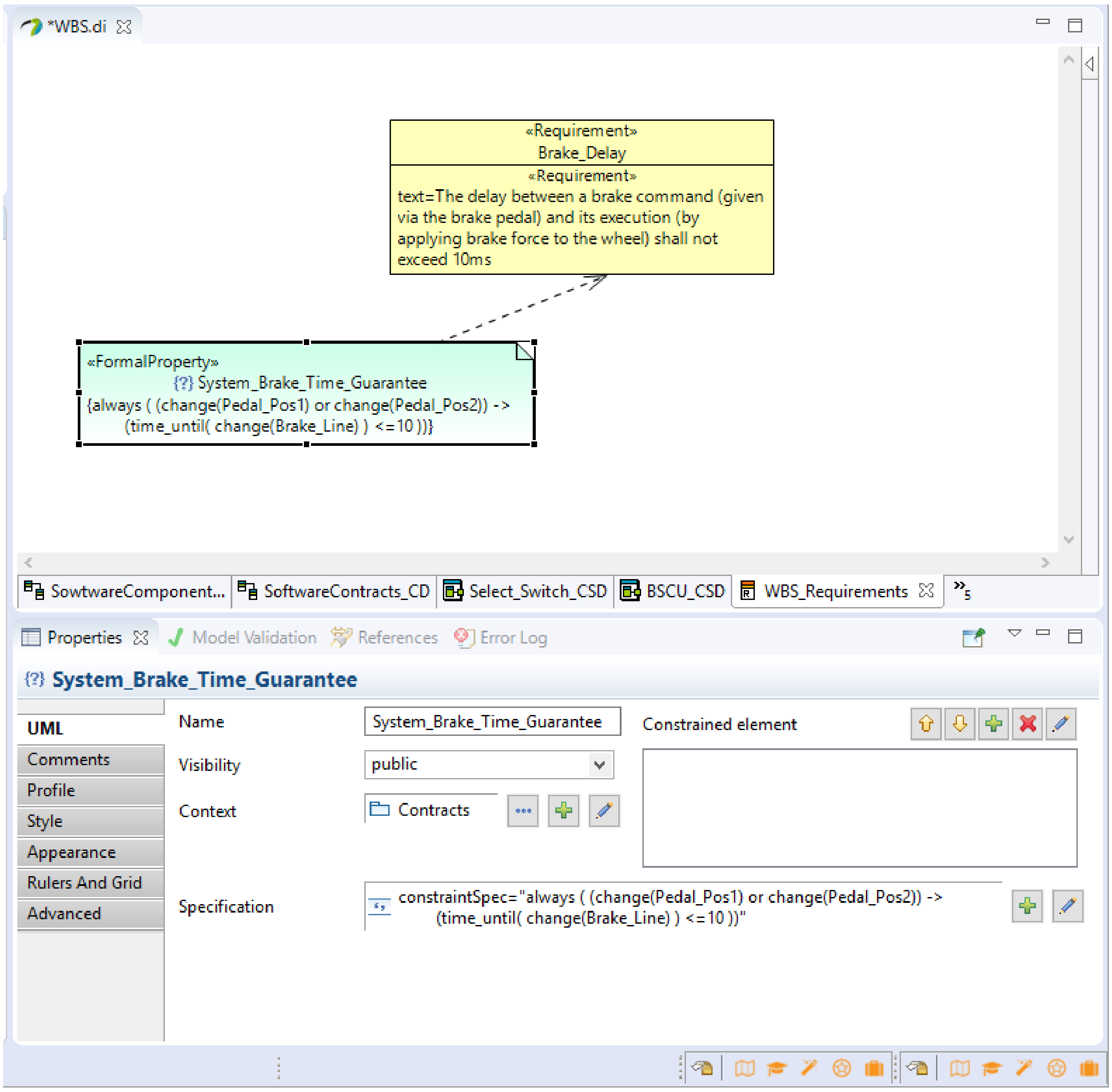
Requirements can be modelled in Papyrus using the SysML profile; indeed, SysML comes with the dedicated *Requirement* stereotype (see Figure 10) which can be managed through Requirement Diagrams. The availability of system requirements represented in the model allows the user to model their traceability 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 requirements traceability (see e.g. [8]), which is an important quality factor to be guaranteed while building systems, can be obtained while using model-driven support.

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, the CHESS profile defines a construct called `FormalProperty` as an extension of UML `Constraint` (see Figure 10). A `FormalProperty` can be created first in the model and then 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[6]. Then the formal description of the requirement is provided by using the *specification* attribute coming with the `FormalProperty` entity. This

[6] https://wiki.eclipse.org/Papyrus_User_Guide/Table_Documentation

attribute can refer to the UML `OpaqueExpression` element that contains language-specific texts to express one formal property in different modelling languages.


*WBS.di
«Requirement»
Brake_Delay
«Requirement»
text=The delay between a brake command (given via the brake pedal) and its execution (by applying brake force to the wheel) shall not exceed 10ms
«FormalProperty»
{?} System_Brake_Time_Guarantee
{always ((change(Pedal_Pos1) or change(Pedal_Pos2)) -> (time_until(change(Brake_Line)) <=10))}
SowtwareComponent...
SoftwareContracts_CD
Select_Switch_CSD
BSCU_CSD
WBS_Requirements
Properties
Model Validation
References
Error Log
UML
Comments
Profile
Style
Appearance
Rulers And Grid
Advanced

{?} System_Brake_Time_Guarantee

Name	System_Brake_Time_Guarantee
Visibility	public
Context	Contracts
Constrained element	
Specification	constraintSpec="always ((change(Pedal_Pos1) or change(Pedal_Pos2)) -> (time_until(change(Brake_Line)) <=10))"

Figure 10. Modelling FormalProperty

It is worth noting here that the CHESS 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. CHESS currently supports integration with the OCRA contract specification language[7]; in particular, through the CHESS Contract plugins explained in Section 0 it is possible to verify formal properties with respect to OCRA syntax.

[7] https://ocra.fbk.eu

2.2.1.3 Semi-Formal Requirement Definition

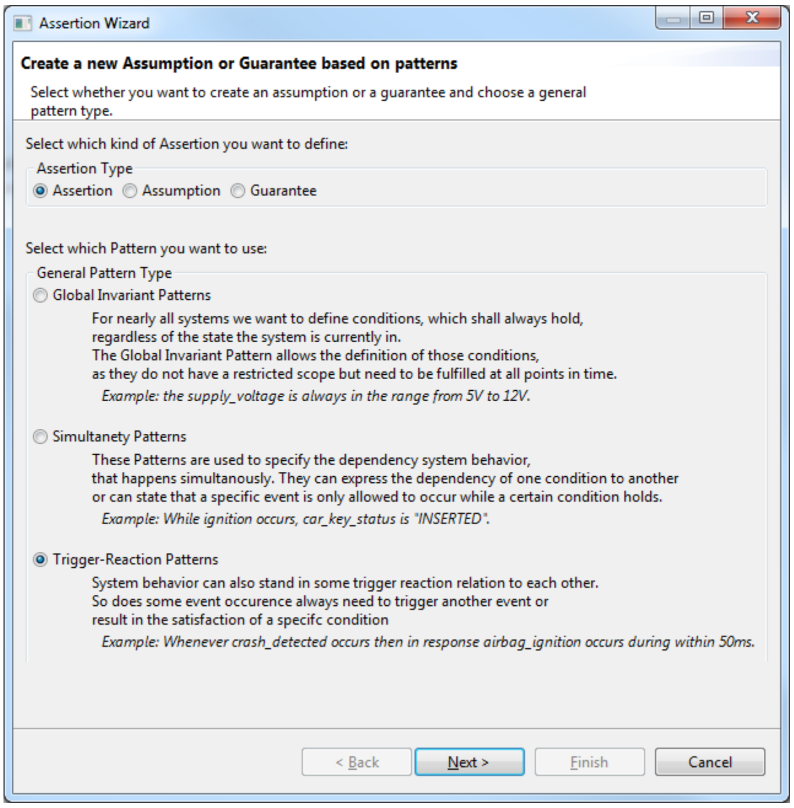
As users might not be familiar with formal expressions to define contracts, we adopted a custom text-based editor with syntax checks and auto-completion, and a wizard to set up assertions with pre-defined templates for the most common assertion patterns.

In the following paragraphs we explain both concepts in detail.

2.2.1.3.1 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 11). We have added a short description and an example for each one so that it is easier for the user to decide.

After selecting the main pattern type, several possible pattern instances of the type are presented to the user. Each of them features an example to demonstrate a possible application (see Figure 12). 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 13). 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 behaviour. Only if no non-terminals remain in the pattern instance and all terminals are replaced by actual interface names, values, units, etc., can the assertion be assigned to a selected component. Otherwise, the wizard will give a hint to the user about the remaining non-terminals or terminals.



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.

Simultanety Patterns

These Patterns are used to specify the dependency system behavior, that happens simultanously. 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.

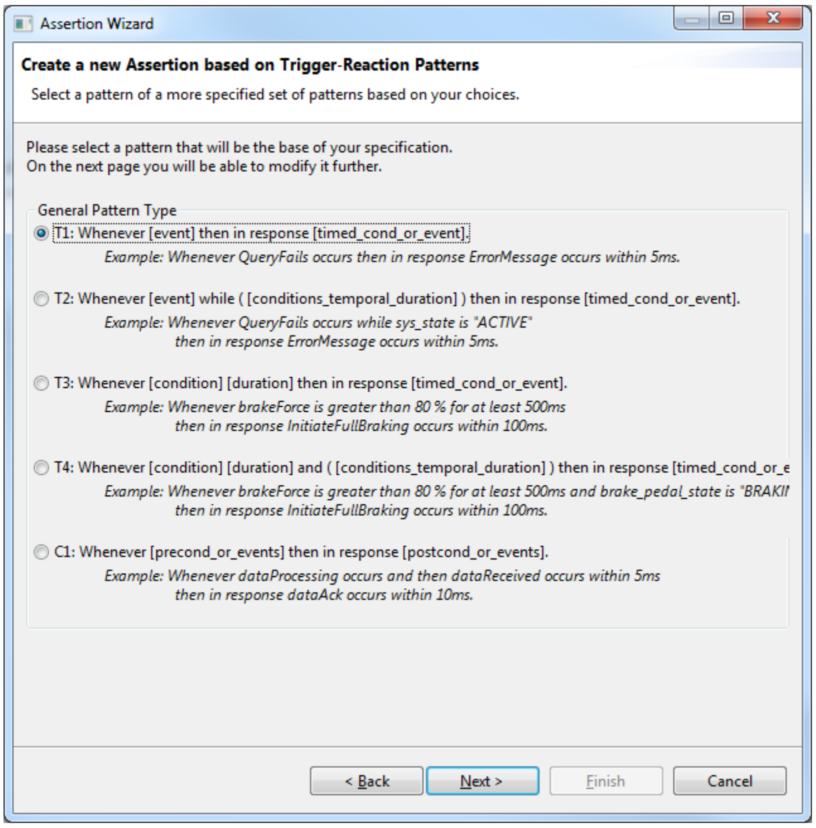
Example: While ignition occurs, car_key_status is "INSERTED".

[✓ Trigger-Reaction Patterns]

System behavior can also stand in some trigger reaction relation to each other. So does some event occurence always need to trigger another event or result in the satisfaction of a specifc condition

Example: Whenever crash_detected occurs then in response airbag_ignition occurs during within 50ms.

Figure 11. 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.

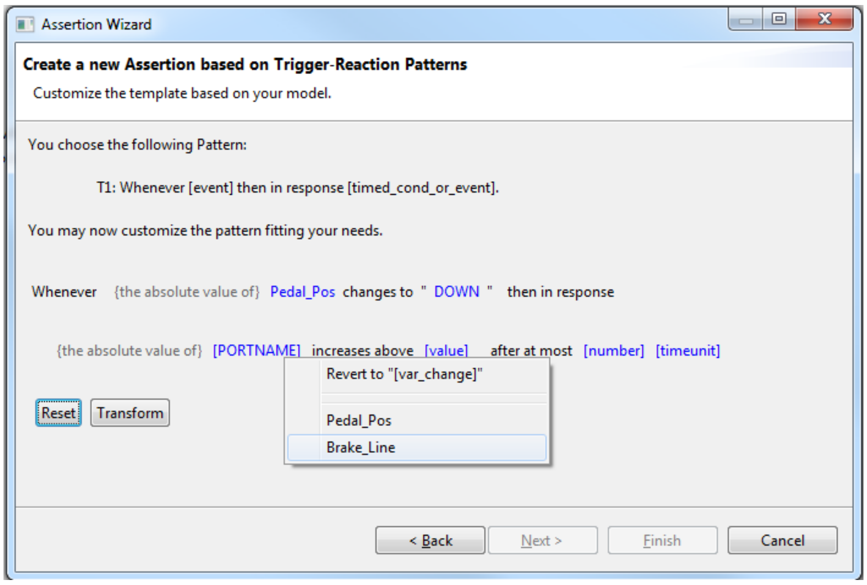
T4: Whenever [condition] [duration] and ([conditions_temporal_duration]) then in response [timed_cond_or_e

Example: Whenever brakeForce is greater than 80 % for at least 500ms and brake_pedal_state is "BRAKII 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.

Figure 12. Second step in the Assertion-Wizard: Choose a pattern instantiation of the previously selected general pattern type


Assertion Wizard

Create a new Assertion based on Trigger-Reaction Patterns

Customize the template based on your model.

You choose the following Pattern:

T1: Whenever [event] then in response [timed_cond_or_event].

You may now customize the pattern fitting your needs.

Whenever {the absolute value of} Pedal_Pos changes to " DOWN " then in response

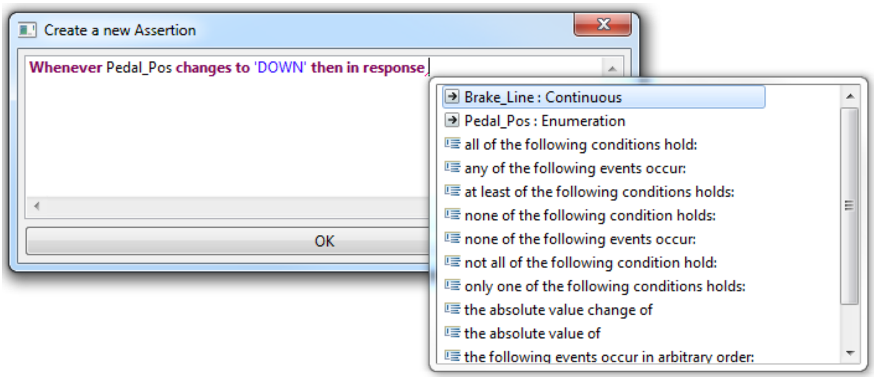
{the absolute value of} [PORTNAME] increases above [value] after at most [number] [timeunit]

Figure 13. 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 can be used

2.2.1.3.2 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, as one might expect from various programming IDEs. Figure 14 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 by providing the BNF in the Xtext grammar format and slightly adjusting the auto-completion suggestions. The rest was automated by the code generation feature of Xtext. Another important reason for choosing Xtext is that 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.


Create a new Assertion

Whenever Pedal_Pos changes to 'DOWN' then in response

Brake_Line : Continuous
Pedal_Pos : Enumeration
all of the following conditions hold:
any of the following events occur:
at least of the following conditions holds:
none of the following condition holds:
none of the following events occur:
not all of the following condition hold:
only one of the following conditions holds:
the absolute value change of
the absolute value of
the following events occur in arbitrary order:
OK

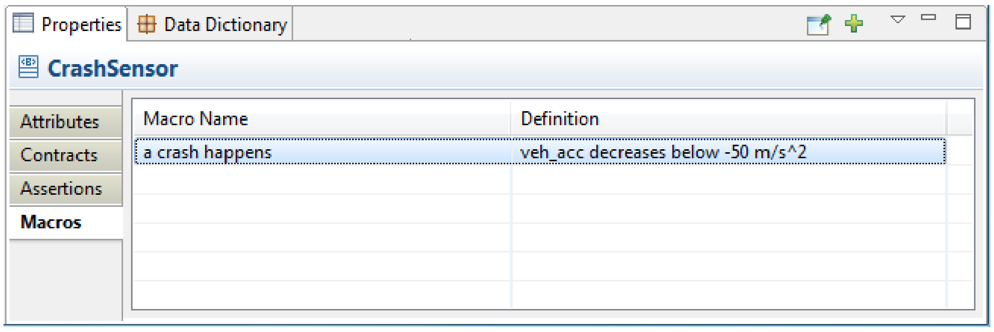
Figure 14. Pattern-suggestion feature of the Assertion Editor

Assertions which are created using the Assertion Wizard or Editor were planned to be automatically translated to formal language expression such as LTL. Since the completion of other features had a higher priority than this one, we could not manage to complete its implementation on time. This might be unfortunate from a research perspective but acceptable from the project's perspective since there were no requirements requesting this feature.

2.2.1.3.3 Macro Definition

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

As Macros are used to create assertions, they can be created on the same types of model elements. To add a new macro, click the Add-Button on the upper right-hand corner of the Macro Section (see Figure 15).



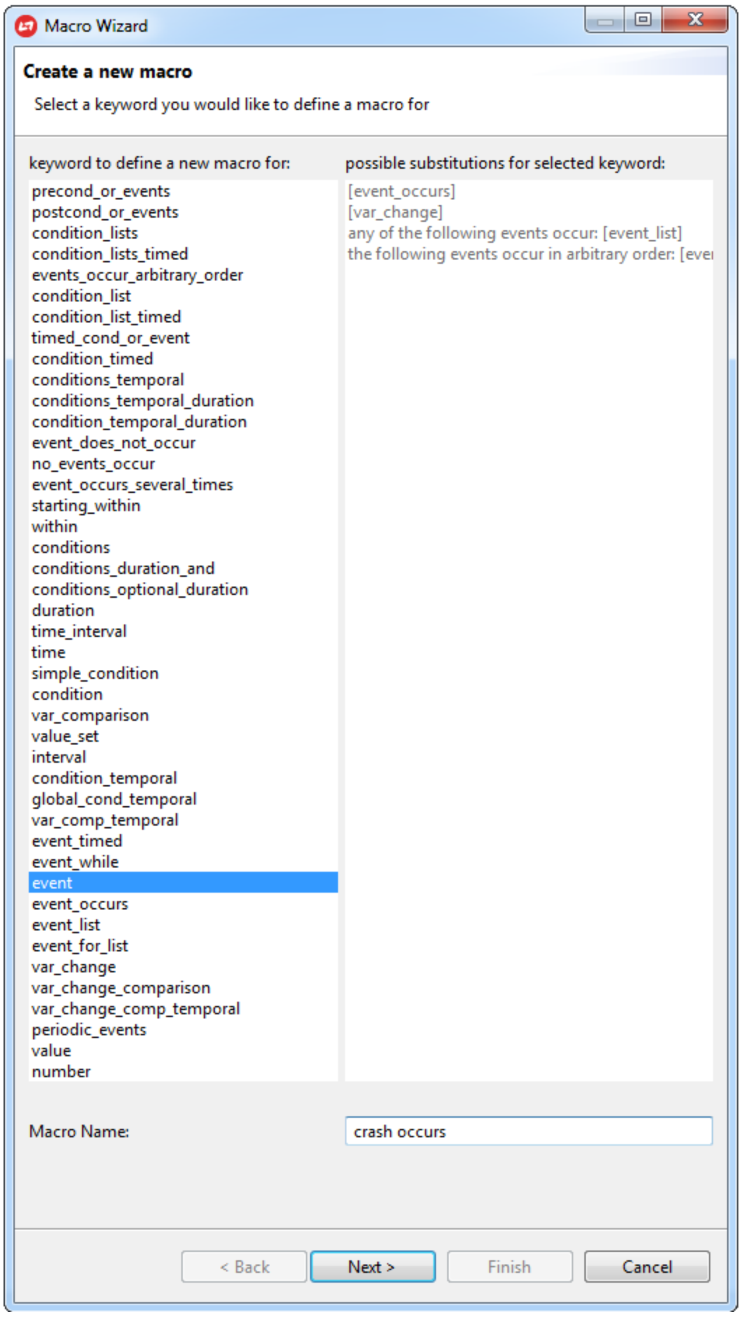
Macro Name	Definition
a crash happens	veh_acc decreases below -50 m/s^2

Figure 15. Macros Section of the Properties View in SAVONA

On the first wizard page, a keyword to define a macro for is selected (see Figure 16). The left side of the wizard shows possible replacements for the currently selected keyword. Additionally, the user sets the macro name.

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

The subsequent macro wizard pages allow the customization of the selected keyword. As it uses the same page layout as the Assertion Wizard, please refer to the description of the Assertion Wizard for detailed information.


Macro Wizard

Create a new macro

Select a keyword you would like to define a macro for

keyword to define a new macro for:

precond_or_events
postcond_or_events
condition_lists
condition_lists_timed
events_occur_arbitrary_order
condition_list
condition_list_timed
timed_cond_or_event
condition_timed
conditions_temporal
conditions_temporal_duration
condition_temporal_duration
event_does_not_occur
no_events_occur
event_occurs_several_times
starting_within
within
conditions
conditions_duration_and
conditions_optional_duration
duration
time_interval
time
simple_condition
condition
var_comparison
value_set
interval
condition_temporal
global_cond_temporal
var_comp_temporal
event_timed
event_while
event
event_occurs
event_list
event_for_list
var_change
var_change_comparison
var_change_comp_temporal
periodic_events
value
number

possible substitutions for selected keyword:

[event_occurs]
[var_change]
any of the following events occur: [event_list]
the following events occur in arbitrary order: [eve

Macro Name: crash occurs

Figure 16. The first page of SAVONA's Macro Wizard

2.2.1.3.4 Data Dictionary (*)

When specifying (semi-)formal assertions, there needs to be a way to define custom variables such as constants or units. SAVONA offers a Data Dictionary View (see Figure 17) where several model elements can be defined that can later be used within the definition of assertions:

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

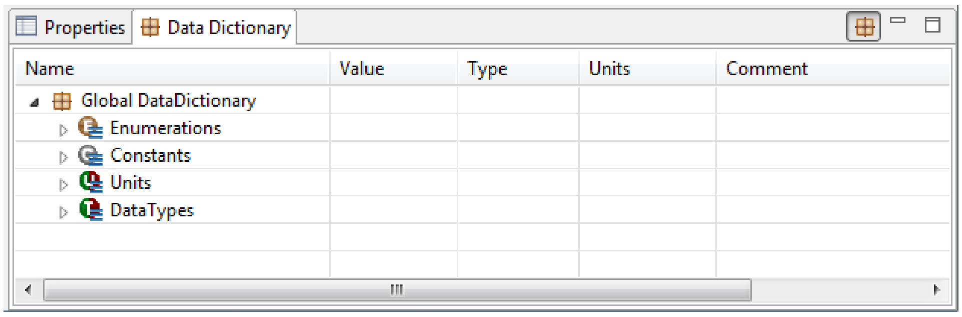


Figure 17. Data Dictionary View in SAVONA

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

The entries of the Data Dictionary are also exported during the CHESS Export. To allow CHESS to properly interpret the Data Dictionary and its entries, the Data Dictionary Papyrus profile is always contained within the exported model files.

2.2.1.4 Structure Properties into Contracts

Table 4. Requirements covering the structure of contracts

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_CAC_003	Structure properties into contracts (assumptions/guarantees)	The system shall be able to support the extraction of assumptions and guarantees to be used in component contracts based on component properties	Solved	CHESS/SAVONA	FBK, B&M

The CHESS profile supports the modelling of weak and strong contracts to support contract-based design (the reader can refer to AMASS D3.1 [10] for an introduction to weak and strong contracts and contract-based design).

Contracts are available in the CHESS 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 CHESS Papyrus extension, when a Contract is created in the model, the tool automatically creates a pair of empty `FormalProperties` to represent the assumption and guarantee of the Contract.

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 18 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.

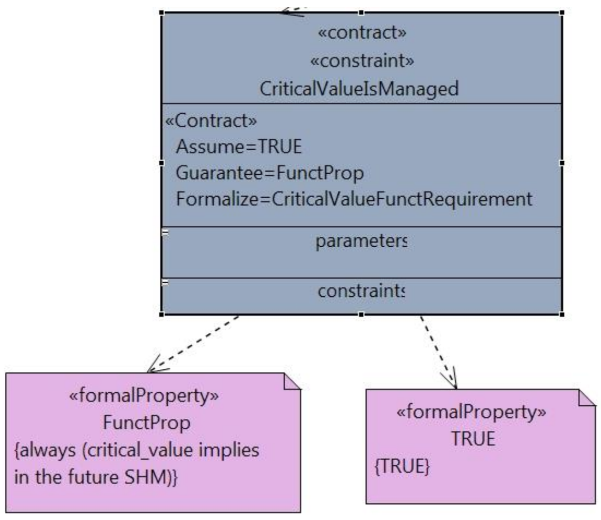


Figure 18. Contract and FormalProperty example

2.2.1.5 Assign Contract to Component (*)

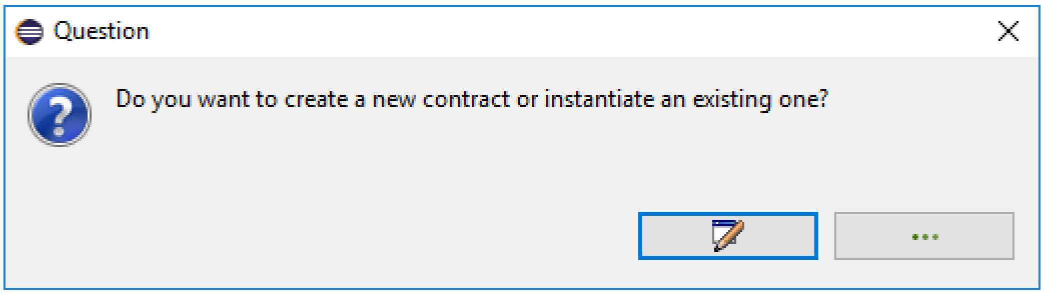
Table 5. Requirements covering contract management

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_CAC_002	Assign contract to component	The system shall allow to associate a contract to a component. Then, the system shall allow to drop a contract from a component	Solved	CHESS, SAVONA	MDH, FBK, B&M
WP3_CAC_003	Structure properties into contracts (assumptions/guarantees)	The system shall be able to support the extraction of assumptions and guarantees to be used in component contracts based on component properties	Solved	CHESS/SAVONA	FBK, B&M
WP3_CAC_005	General management of contract-component assignments	The system should enable users to have a view of the association between contracts and components for the entire system architecture (thus, not only a view on the single contract assignment for each component)	Solved	CHESS	INT, FBK
WP3_CAC_009	Improvement of Contract definition process	The operation of contract definition should be improved in terms of time spent.	Solved	CHESS, SAVONA	FBK, B&M
WP3_CAC_013	Specify contracts defining the assumption and the guarantee elements	The system shall provide the capability to create a contract defining two new properties (assumptions/guarantees) implicitly associated to that contract.	Solved	CHESS	INT

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_CAC_014	Drop contract from component	The system shall allow to drop a contract from a component	Solved	CHESS, SAVONA	INT, B&M
WP3_CAC_015	Reassign contract to component	The system shall allow to substitute the already assigned contract to a component with another contract	Solved	CHESS	INT

In CHESS, a contract can be assigned to a component by instantiating the contract in the component itself. The instantiation is realized by creating for a component a special kind of property, called `ContractProperty` which is typed with the given contract. This allows to potentially reuse the same contract in different contexts/systems (as analogous to the practice of sharing requirements across projects, i.e. software/system requirements reuse).

It is possible to first create the contract and then assign it to a component. The AMASS prototype enables also the possibility to automatically create a contract when a `ContractProperty` is created, see [21]. 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.


Question

Do you want to create a new contract or instantiate an existing one?

Figure 19. 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

`ContractProperty` has also an attribute that allows specifying whether the associated Contract has to be applied to the Component/Block according to the weak or strong semantics[8] [10].

As example, Figure 20 shows the *criticalValueIsManaged ContractProperty* owned by the *FunctionalSystem* Block (the *ContractProperty* is shown in the diagram in the *Constraint* compartment of the Block). The *criticalValueIsManaged* property is typed as *CriticalValueIsManaged* Contract, the latter is also represented in the diagram. The *criticalValueIsManaged* property represents the association of the *CriticalValueIsManaged* Contract *to the FunctionalSystem* Block.

[8] As discussed [10], 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.

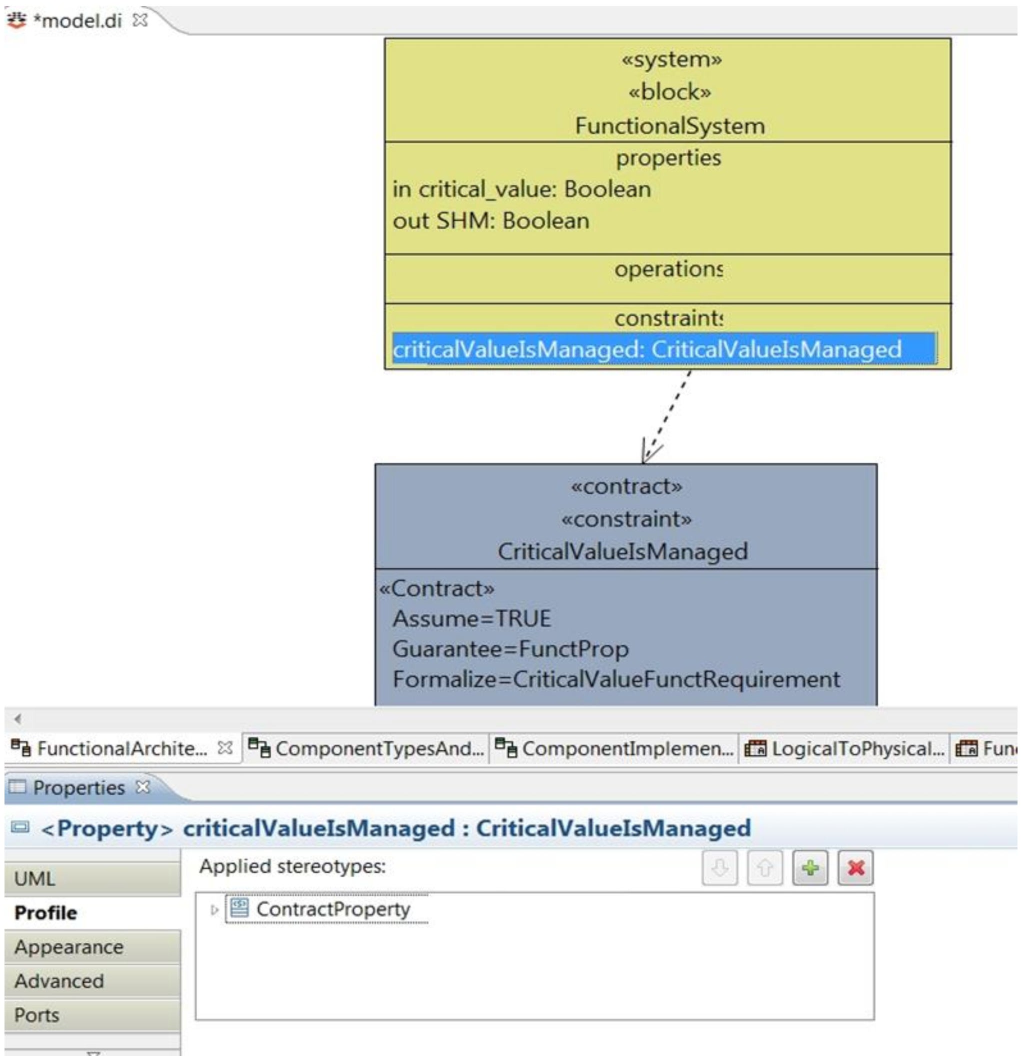
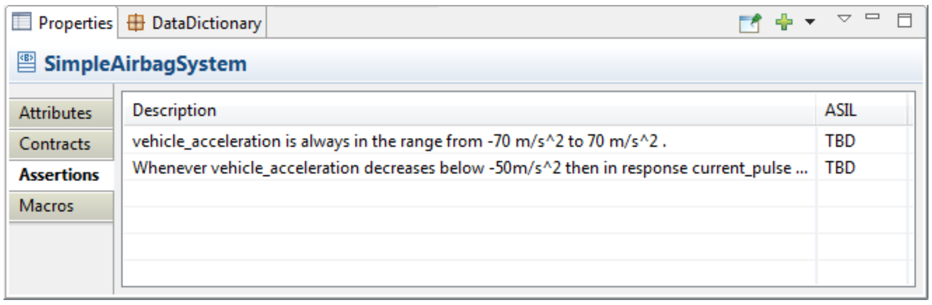


Figure 20. Assign Contract to Component

2.2.1.5.1 Contract Definition in SAVONA (*)

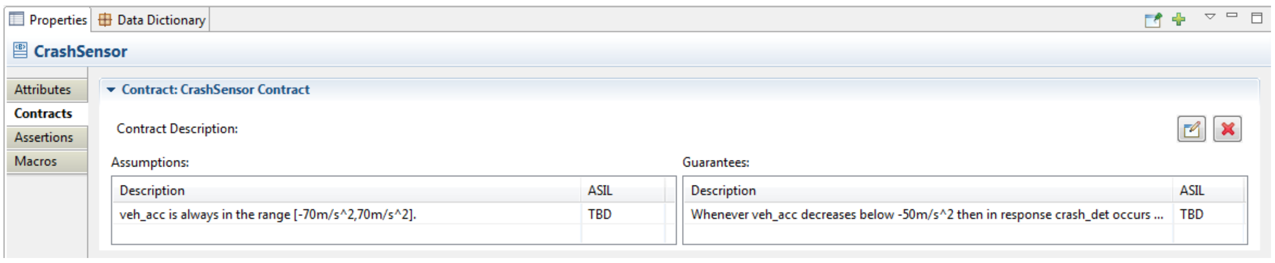
To specify a contract within SAVONA, assertions must be gathered that are later used either as assumptions or guarantees. The Assertions Section of the Properties View (see Figure 21) shows assertions that are defined for the currently selected SysML block, interface or connector.



Description	ASIL
vehicle_acceleration is always in the range from -70 m/s^2 to 70 m/s^2 .	TBD
Whenever vehicle_acceleration decreases below -50m/s^2 then in response current_pulse ...	TBD

Figure 21. Assertions Section in the Properties View of SAVONA

As contracts can only be defined on SysML blocks via the Contracts Section (see Figure 22), the Contracts section shows an overview of all contracts assigned to the currently selected type and allows the editing, creation and deletion of contracts.



Contract: CrashSensor Contract

Contract Description:

Assumptions:

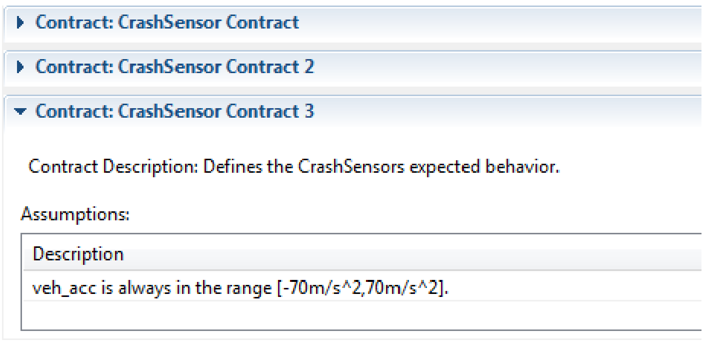
Description	ASIL
veh_acc is always in the range [-70m/s^2,70m/s^2].	TBD

Guarantees:

Description	ASIL
Whenever veh_acc decreases below -50m/s^2 then in response crash_det occurs ...	TBD

Figure 22. Contracts Section in the Properties View of SAVONA

In the Contracts Section, contracts can be collapsed or expanded as shown in Figure 23, allowing a clean overview of the existing contracts.


Contract: CrashSensor Contract

Contract: CrashSensor Contract 2

Contract: CrashSensor Contract 3

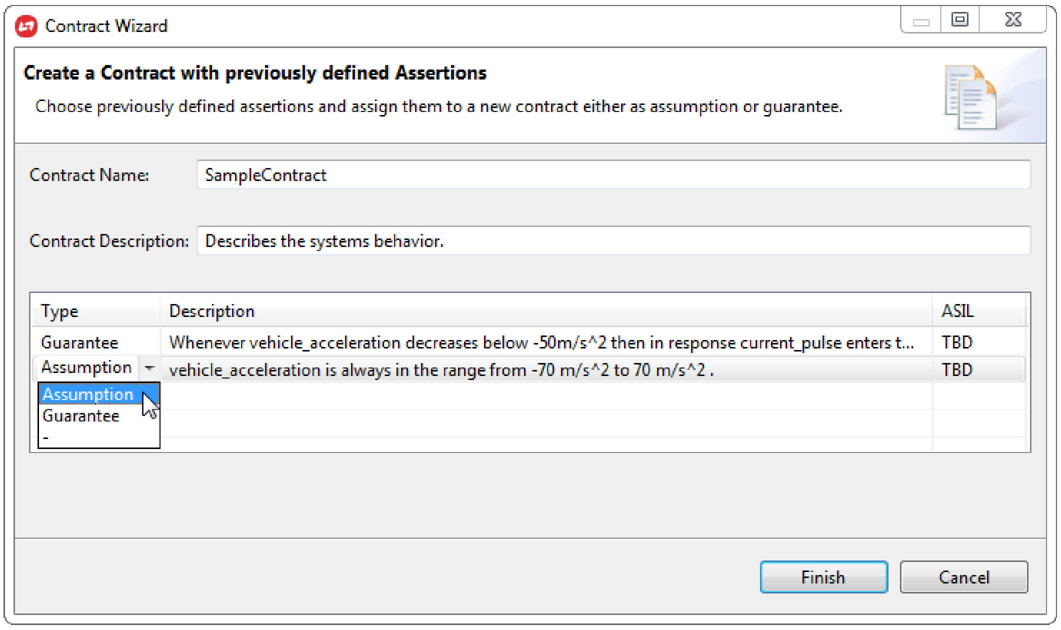
Contract Description: Defines the CrashSensors expected behavior.

Assumptions:

Description
veh_acc is always in the range [-70m/s^2,70m/s^2].

Figure 23. Collapsed contracts in the Contract Section of SAVONA

When defining a new contract in SAVONA, the Contract Wizard is used to ease the process of assigning assertions as assumptions or guarantees. Previously defined assertions can be used to create a contract for a component. The wizard offers assertions that are defined on the currently selected SysML block and the owned ports of the block (see Figure 24).



Create a Contract with previously defined Assertions

Choose previously defined assertions and assign them to a new contract either as assumption or guarantee.

Contract Name: SampleContract

Contract Description: Describes the systems behavior.

Type	Description	ASIL
Guarantee	Whenever vehicle_acceleration decreases below -50m/s^2 then in response current_pulse enters t...	TBD
Assumption	vehicle_acceleration is always in the range from -70 m/s^2 to 70 m/s^2 .	TBD

Figure 24. Contract Wizard of SAVONA

To use one of the offered assertions in a contract, simply assign a type (either assumption or guarantee) to it. At least one guarantee is needed to create a contract. Multiple assumptions as well as multiple guarantees are conjunct. Assertions without any type assignment will not be considered in the contract definition.

2.2.1.6 Contract Refinement

Table 6. Requirements covering contract refinement

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_CAC_004	Specify contract refinement	The system shall enable users to specify the refinement of the contract along the hierarchical component's architecture	Solved	CHESS/SAVONA	FBK, B&M
WP3_CAC_006	Refinement-based overview	The system should enable users to have a hierarchical view of the contract refinements along the system architecture	Solved	CHESS, SAVONA	FBK, B&M

The CHESS profile allows the modelling of contract refinement and decomposition along the refinement and 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 CHESS 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,…,A_n) and the contracts instances assigned to each subcomponent ($C_{1<1..k>}$,… ,$C_{n<1..j>}$)., it is possible to model how C is decomposed by (a subset of) ($C_{1<1..k>}$,…,$C_{n<1..j>}$); it is worth noting that contracts decomposition can then be modelled for the subcomponents as well, so to have multiple levels of contracts decomposition.

2.2.1.7 Modelling Failure Behaviour

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

Extension of the aforementioned CHESS dependability profile has been done in AMASS to address security concerns also, as documented in AMASS deliverable D4.3 [24] section 2.1.4.1.

2.2.2 System Architecture Modelling for Assurance

Table 7. Requirements covering architecture modelling for assurance

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_SAM_001	Trace component with assurance assets	The supplier of a component shall be able to trace all the assurance information with the specific component	Solved	CAPRA	INT
WP3_SAM_003	Compare different architectures according to different concerns which haven't been specified	The system shall be able to compare different system architectures based on predefined criteria, like	Solved	CHESS	FBK

Requirement No	Name	Description	Status	Tools	Involved Partners
	before	dependability or timing concerns			
WP3_SAM_004	Integration with external modelling tools	The system could interact with external tools for system design and development (e.g., Rhapsody, AutoFocus, Compass) to get the system architecture.	Solved	CHESS, Papyrus	INT, UC3, TRC, FBK, B&M

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 are described in the AMASS deliverable D3.3 [16].

As explained in the previous sections, the AMASS component model has been made available as Eclipse plugin as UML/SysML language extended with the CHESS profile for contracts, while the other parts of the CACM (argumentation, evidence, compliance management) are currently implemented as Ecore meta-models[9] (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 CHESS-Contract to an argumentation-Claim) cannot be expressed through the CHESS profile; the links have to be managed with some additional modelling support, as explained below in the text.

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`)[10]; 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 25 gives a picture of what has been stated above (CACM model in the figure is intended as the model for argumentation, evidence, and compliance management).

[9] 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 a 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.

[10] It is worth noting that EAnnotation can be added to UML model entities because UML models in Eclipse are implemented as Ecore models.

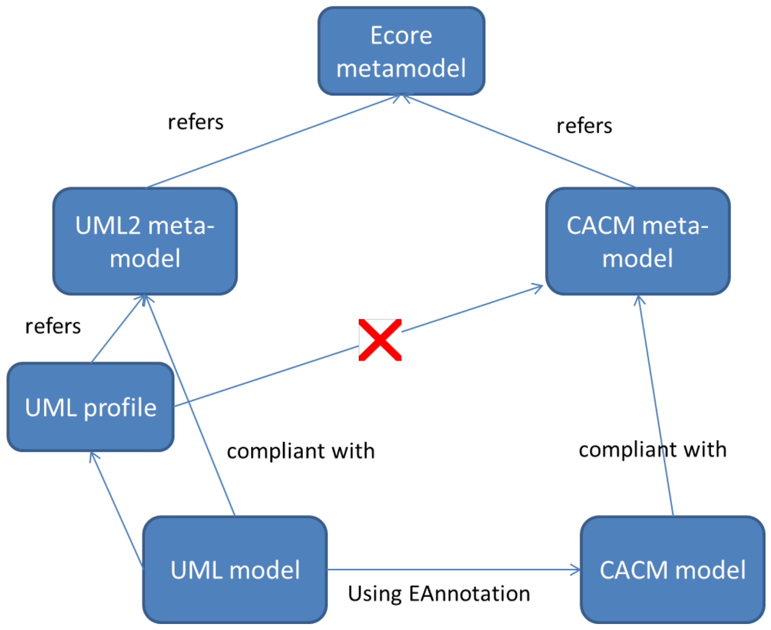


Figure 25. Links through EAnnotation

However, the solution of using EAnnotation does not allow to formalize the kind of connections that are allowed between the different metamodels.

The solution adopted in AMASS to support links between architectural entities and the other parts of the CACM AMASS meta-model foresees the usage of a dedicated traceability meta-model (see Figure 26). In this way, a link is created according to the traceability meta-model; each link owns a reference to the UML model entity and a reference to the CACM model entity to be associated.

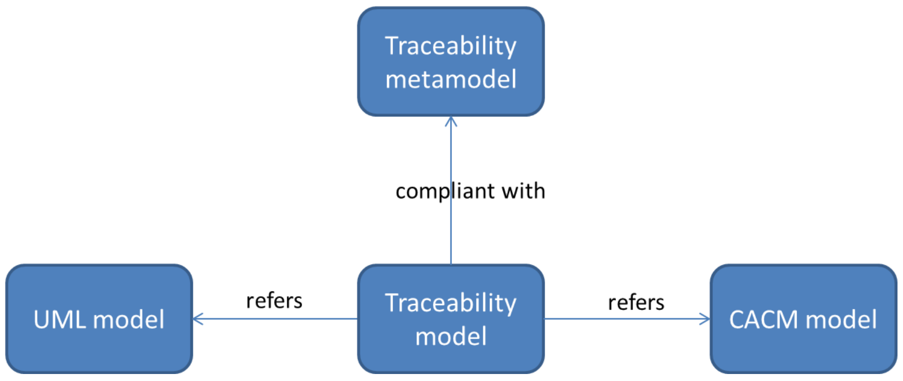


Figure 26. Links through traceability meta-model

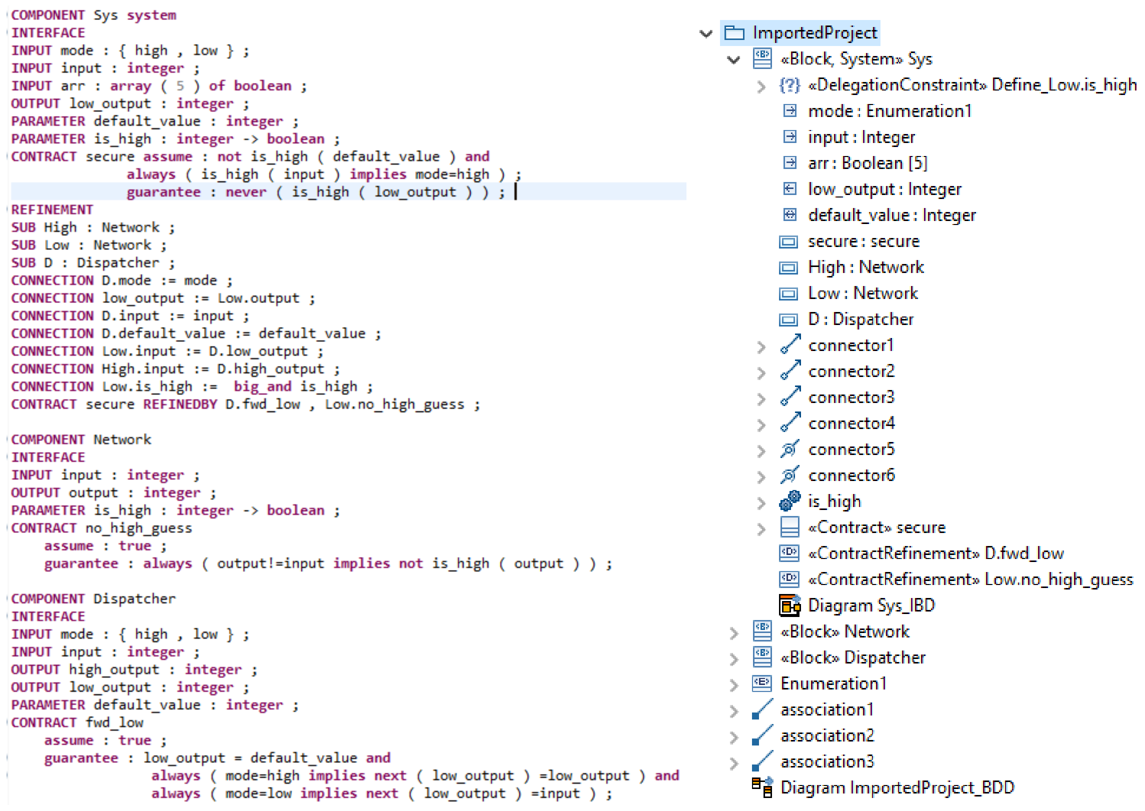
What is worth noting is that the usage of a dedicated traceability meta-model can be made generic in order to support traceability between assurance case information and architecture-related entities specified with other non-UML modelling languages. For instance, by assuming the availability of an Architecture Analysis and Design Language (AADL)[11] 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.

[11] http://www.aadl.info/aadl/currentsite

In AMASS we use the Eclipse Capra[12] tool which offers a framework supporting the solution depicted in Figure 26; see AMASS D5.6 Prototype for seamless interoperability (c) [26] for further details about Capra usage in AMASS.

2.2.2.2 Import System component specification (*)

The third prototype (Prototype P2) enables the import of the system architecture specification in oss format[13]. The process involves the parsing of the .oss file and the creation of the CHESS entities (including the SysML diagrams) of the model. Figure 27 left-side and right-side show respectively the file.oss and the "Model Explorer View" populated with the imported entities.



```
COMPONENT Sys system
INTERFACE
INPUT mode : { high , low } ;
INPUT input : integer ;
INPUT arr : array ( 5 ) of boolean ;
OUTPUT low_output : integer ;
PARAMETER default_value : integer ;
PARAMETER is_high : integer -> boolean ;
CONTRACT secure assume : not is_high ( default_value ) and
              always ( is_high ( input ) implies mode=high ) ;
              guarantee : never ( is_high ( low_output ) ) ;
REFINEMENT
SUB High : Network ;
SUB Low : Network ;
SUB D : Dispatcher ;
CONNECTION D.mode := mode ;
CONNECTION low_output := Low.output ;
CONNECTION D.input := input ;
CONNECTION D.default_value := default_value ;
CONNECTION Low.input := D.low_output ;
CONNECTION High.input := D.high_output ;
CONNECTION Low.is_high :=  big_and is_high ;
CONTRACT secure REFINEDBY D.fwd_low , Low.no_high_guess ;

COMPONENT Network
INTERFACE
INPUT input : integer ;
OUTPUT output : integer ;
PARAMETER is_high : integer -> boolean ;
CONTRACT no_high_guess
    assume : true ;
    guarantee : always ( output!=input implies not is_high ( output ) ) ;

COMPONENT Dispatcher
INTERFACE
INPUT mode : { high , low } ;
INPUT input : integer ;
OUTPUT high_output : integer ;
OUTPUT low_output : integer ;
PARAMETER default_value : integer ;
CONTRACT fwd_low
    assume : true ;
    guarantee : low_output = default_value and
                always ( mode=high implies next ( low_output ) =low_output ) and
                always ( mode=low implies next ( low_output ) =input ) ;
```

```
ImportedProject
  «Block, System» Sys
    «DelegationConstraint» Define_Low.is_high
    mode : Enumeration1
    input : Integer
    arr : Boolean [5]
    low_output : Integer
    default_value : Integer
    secure : secure
    High : Network
    Low : Network
    D : Dispatcher
    connector1
    connector2
    connector3
    connector4
    connector5
    connector6
    is_high
    «Contract» secure
    «ContractRefinement» D.fwd_low
    «ContractRefinement» Low.no_high_guess
    Diagram Sys_IBD
  «Block» Network
  «Block» Dispatcher
  Enumeration1
  association1
  association2
  association3
  Diagram ImportedProject_BDD
```

Figure 27. Example of file.oss and the "Model Explorer View" populated with the imported entities.

This feature allows the Prototype P2 to obtain the system architecture from external system design and development tools. A candidate tool is AutoFOCUS3[14] because it supports the import/export of oss files.

2.2.2.3 Compare different architectures (*)

This operation involves the following sequential activities:

1. Model the parameterized architecture
2. Instantiate the architecture
3. Perform the analyses over the architecture instances

12 https://projects.eclipse.org/projects/modeling.capra

13 https://es.fbk.eu/tools/ocra/download/OCRA_Language_User_Guide.pdf

14 https://af3.fortiss.org/

4. Compare the architecture instances according to the analysis results

A parameterized architecture is an architecture in which the number of components, the number of ports, the connections, and the static attributes of components depends on a (possibly infinite) set of parameters. In CHESS, such parameters are mapped to static UML ports. In this way, it is not required to define an additional element in the CHESS profile. The parameters can be constrained using the UML Constraint element. The elements that compose the architecture, such as the ports and the subcomponents, can be parameterized through the editing of their multiplicity attributes using the defined parameters, see Figure 28.

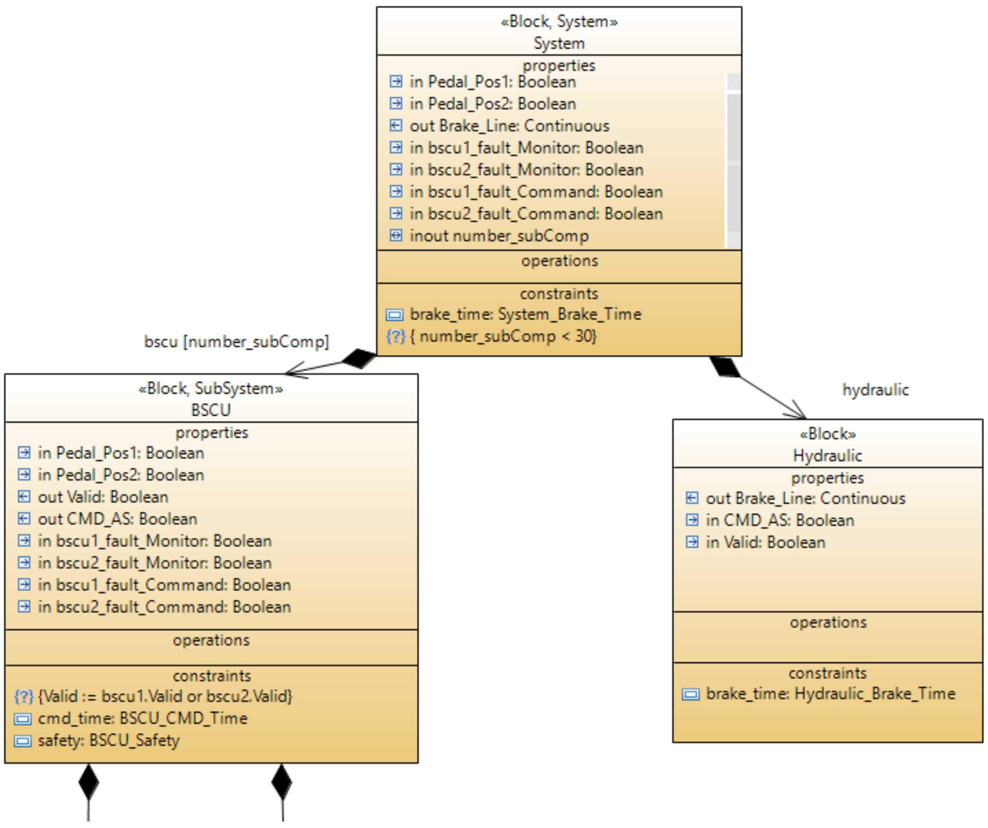

«Block, System»
System
properties
in Pedal_Pos1: Boolean
in Pedal_Pos2: Boolean
out Brake_Line: Continuous
in bscu1_fault_Monitor: Boolean
in bscu2_fault_Monitor: Boolean
in bscu1_fault_Command: Boolean
in bscu2_fault_Command: Boolean
inout number_subComp
operations
constraints
brake_time: System_Brake_Time
{?} { number_subComp < 30}
bscu [number_subComp]
hydraulic
«Block, SubSystem»
BSCU
properties
in Pedal_Pos1: Boolean
in Pedal_Pos2: Boolean
out Valid: Boolean
out CMD_AS: Boolean
in bscu1_fault_Monitor: Boolean
in bscu2_fault_Monitor: Boolean
in bscu1_fault_Command: Boolean
in bscu2_fault_Command: Boolean
operations
constraints
{?} {Valid := bscu1.Valid or bscu2.Valid}
cmd_time: BSCU_CMD_Time
safety: BSCU_Safety
«Block»
Hydraulic
properties
out Brake_Line: Continuous
in CMD_AS: Boolean
in Valid: Boolean
operations
constraints
brake_time: Hydraulic_Brake_Time

Figure 28. BDD describing a parameterized architecture

The System component has the parameter ‘number_subComp’ that is used to define the number of subcomponents of type BSCU. This parameter is constrained by the expression ‘number_subComp <30’.

The modelling of the parameterized architecture is followed by its instantiation. In this phase the user sets the values of the parameters, i.e. he/she defines the configuration of the architecture.

OCRA takes in input the parameterized architecture and one or more configurations. Then, OCRA produces the instances of the architecture, and for each of them, it performs a list of contract-based verifications. The output are the results derived from the contract-based verifications (that are described in Section 2.2.4). The comparison of the results is described in the D4.6 deliverable [25], Section Trade-off Analysis.

2.2.3 V&V-based Assurance Impact Assessment

2.2.3.1 Metrics

Following describes the metrics for models and checklist type implemented in the Requirement Quality Analyzer tool RQA [22] presented in the section *2.4.4.3 Metrics for models* and *2.4.4.4 Metric checklists* of the D3.3 deliverable [16].

2.2.3.1.1 **Correctness metrics for models (*)**

The RQA tool implements the follow correctness metrics to assess models (Figure 29).

Class model

- **Method hiding factor**

This metric is a measure of the encapsulation in the class. It is the ratio of the sum of hidden methods (private and protected) to the total number of methods defined in each class (public, private, and protected).

- **Attribute Hiding Factor**

This metric represents the average of the invisibility of attributes in the class diagram. It is the ratio of the sum of hidden attributes (private and protected) for all the classes to the sum of all defined attributes (public, private, and protected).

- **Public methods**

This metric calculates the public methods in a class.

- **Number of methods**

This metric Count all methods (public, protected, and private) in a class.

- **Design Size in Classes**

This metric is a count of the total number of classes in the design.

Package model

- **Abstractness**

The abstractness metric measures the package abstraction rate. A package abstraction level depends on its stability level. Calculations are performed on classes defined directly in the package and those defined in sub-packages. In UML models, this metric is calculated on all the model classes.

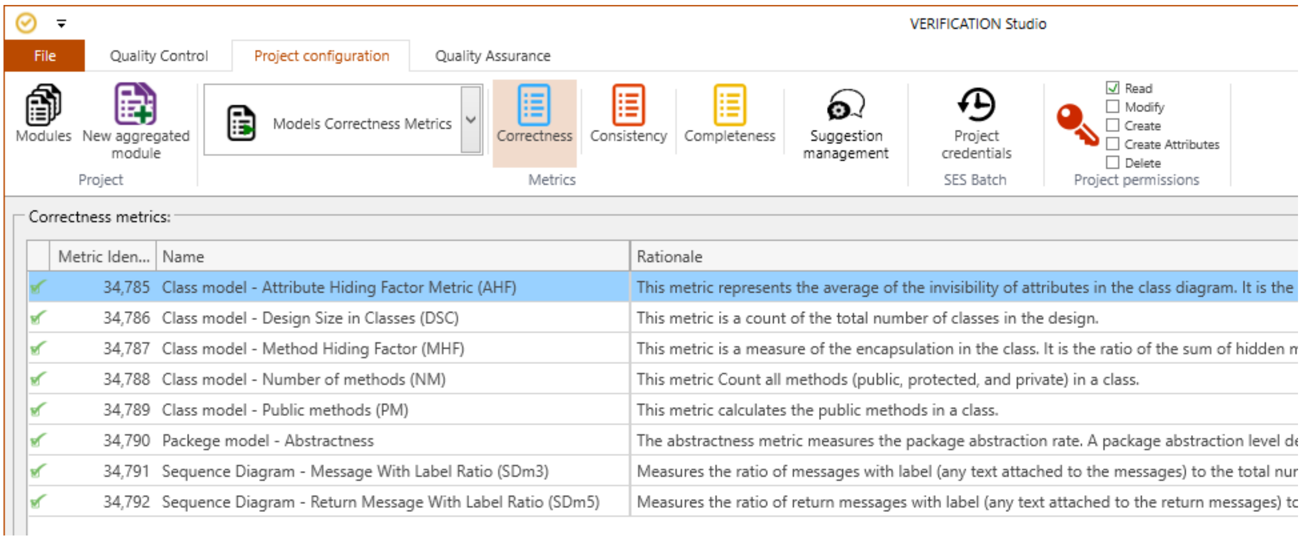
Sequence Diagram

- **Message With Label Ratio**

Measures the ratio of messages with label (any text attached to the messages) to the total number of messages in a sequence diagram.

- **Return Message With Label Ratio**

Measures the ratio of return messages with label (any text attached to the return messages) to the total number of return messages in a sequence diagram.



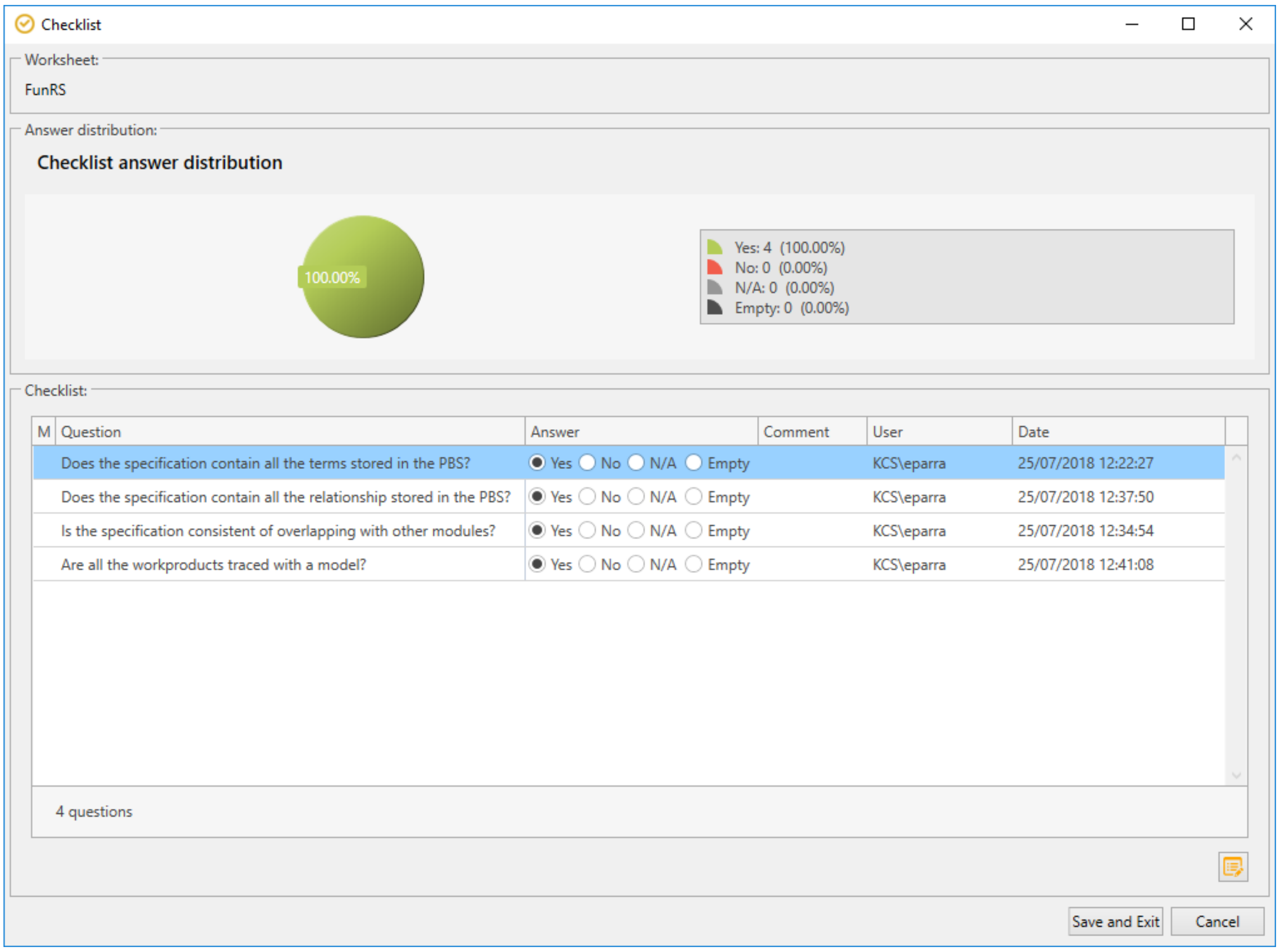
Correctness metrics:

Metric Iden...	Name	Rationale
34,785	Class model - Attribute Hiding Factor Metric (AHF)	This metric represents the average of the invisibility of attributes in the class diagram. It is the
34,786	Class model - Design Size in Classes (DSC)	This metric is a count of the total number of classes in the design.
34,787	Class model - Method Hiding Factor (MHF)	This metric is a measure of the encapsulation in the class. It is the ratio of the sum of hidden m
34,788	Class model - Number of methods (NM)	This metric Count all methods (public, protected, and private) in a class.
34,789	Class model - Public methods (PM)	This metric calculates the public methods in a class.
34,790	Packege model - Abstractness	The abstractness metric measures the package abstraction rate. A package abstraction level de
34,791	Sequence Diagram - Message With Label Ratio (SDm3)	Measures the ratio of messages with label (any text attached to the messages) to the total nur
34,792	Sequence Diagram - Return Message With Label Ratio (SDm5)	Measures the ratio of return messages with label (any text attached to the return messages) tc

Figure 29. Correctness metrics for models

2.2.3.1.2 Checklist metrics (*)

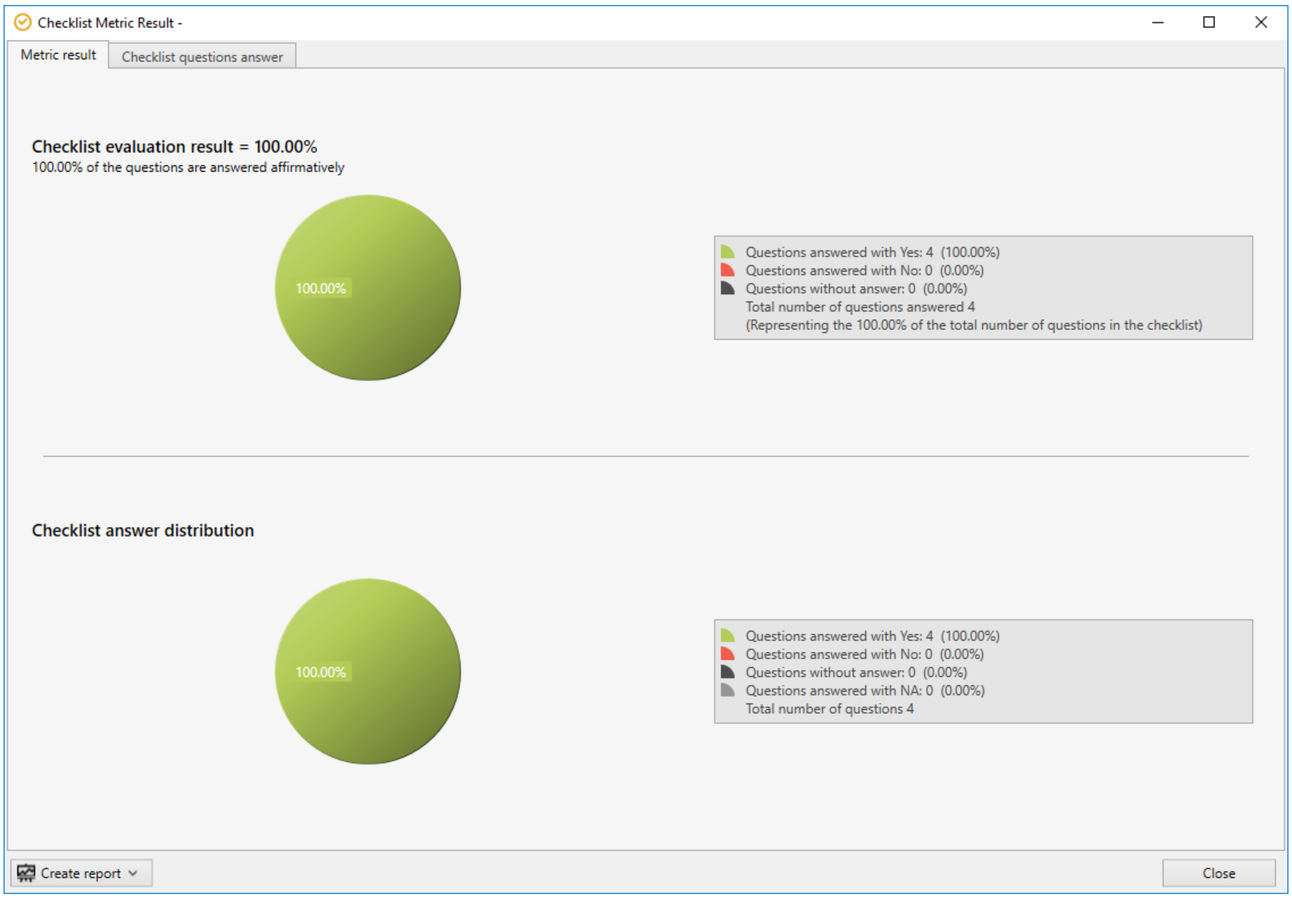
The RQA tool has included a new kind of metric based on Checklist. A checklist is a test with a series of questions that the user must answer (Figure 30).



M	Question	Answer	Comment	User	Date
	Does the specification contain all the terms stored in the PBS?	[✓ Yes] No N/A Empty		KCS\eparra	25/07/2018 12:22:27
	Does the specification contain all the relationship stored in the PBS?	[✓ Yes] No N/A Empty		KCS\eparra	25/07/2018 12:37:50
	Is the specification consistent of overlapping with other modules?	[✓ Yes] No N/A Empty		KCS\eparra	25/07/2018 12:34:54
	Are all the workproducts traced with a model?	[✓ Yes] No N/A Empty		KCS\eparra	25/07/2018 12:41:08

Figure 30. Window to answer the questions of the checklist metrics

Depending on the answers, it is possible to weight the result to provide a quality measure using quality ranges defined (Figure 31).



Checklist evaluation result = 100.00%
100.00% of the questions are answered affirmatively

100.00%

🟩 Questions answered with Yes: 4 (100.00%)
🟥 Questions answered with No: 0 (0.00%)
⬛ Questions without answer: 0 (0.00%)
Total number of questions answered 4
(Representing the 100.00% of the total number of questions in the checklist)

Checklist answer distribution

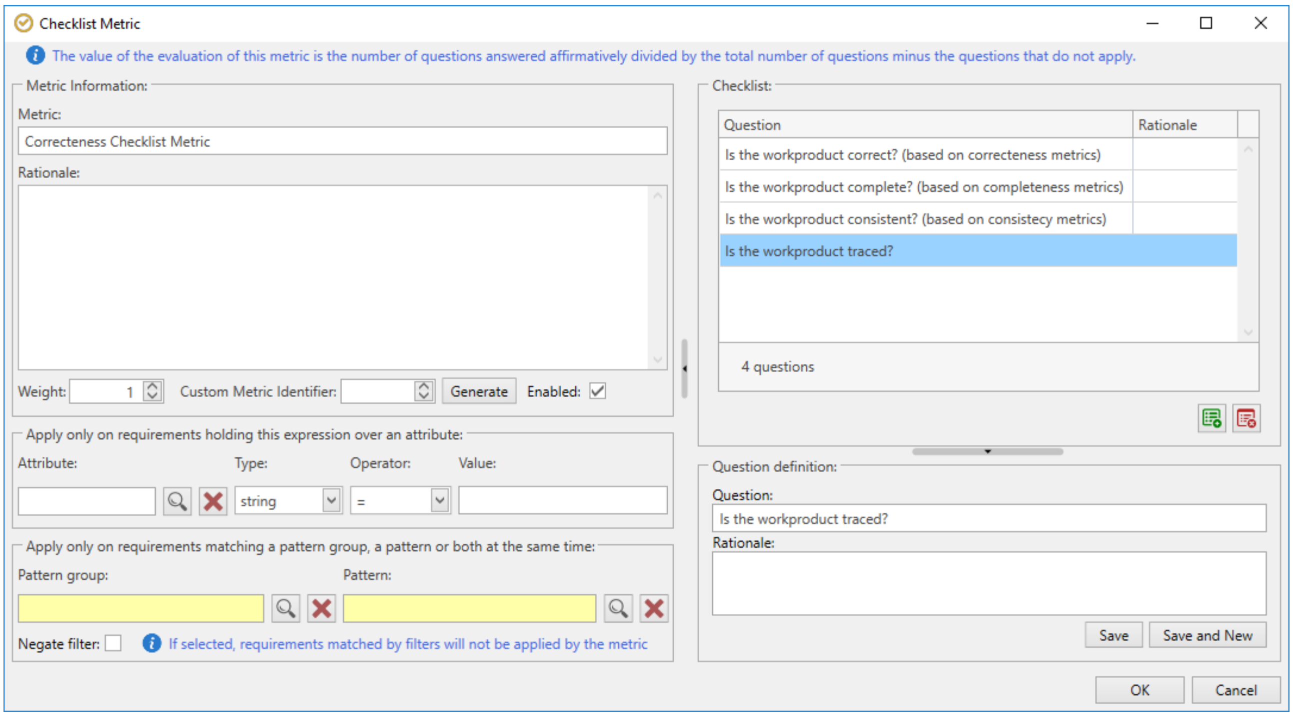
100.00%

🟩 Questions answered with Yes: 4 (100.00%)
🟥 Questions answered with No: 0 (0.00%)
⬛ Questions without answer: 0 (0.00%)
▨ Questions answered with NA: 0 (0.00%)
Total number of questions 4

Figure 31. Results presentation of the checklist metrics

There are two types of checklist metric:

- Correctness checklist metric: the question included in this metric must be answered by each workproduct of the specification (Figure 32).


Checklist Metric

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

Metric Information:

Metric: Correcteness Checklist Metric

Rationale:

Weight: 1 Custom Metric Identifier: Generate Enabled: ✓

Apply only on requirements holding this expression over an attribute:

Attribute: Type: string Operator: = Value:

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

Pattern group: Pattern:

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

Checklist:

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

4 questions

Question definition:

Question: Is the workproduct traced?

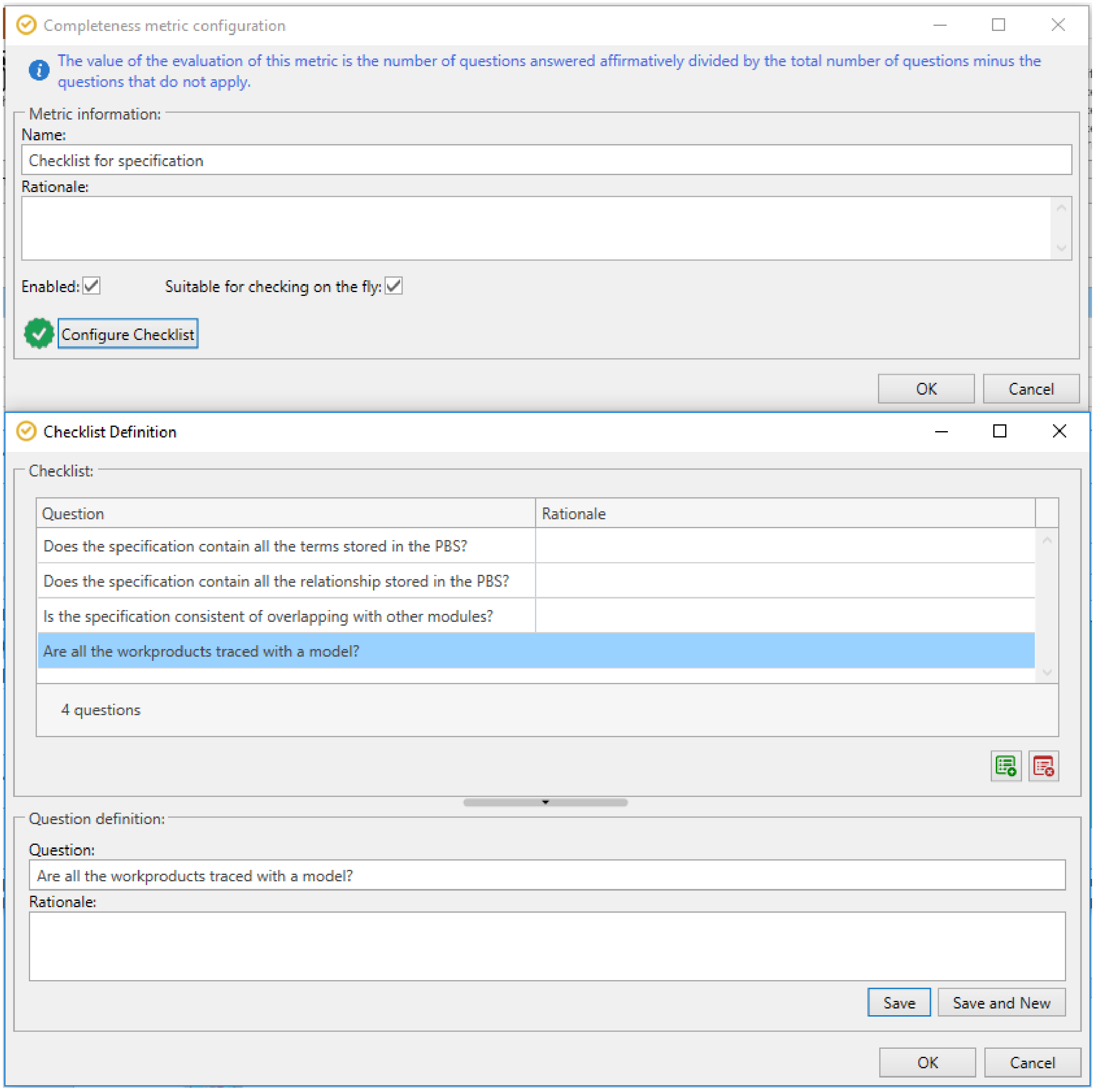
Rationale:

Save | Save and New

OK | Cancel

Figure 32. Correctness checklist metric configuration

- Completeness checklist metric: the questions included in this metric must be answered at the specification level (Figure 33).


Completeness metric configuration

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

Metric information:
Name: Checklist for specification
Rationale:
Enabled: ✓ Suitable for checking on the fly: ✓

Checklist Definition

Checklist:

Question	Rationale
Does the specification contain all the terms stored in the PBS?	
Does the specification contain all the relationship stored in the PBS?	
Is the specification consistent of overlapping with other modules?	
Are all the workproducts traced with a model?	

4 questions

Question definition:
Question: Are all the workproducts traced with a model?
Rationale:

Figure 33. Completeness checklist metric configuration

2.2.3.2 Connectors (*)

A connector between RQA and the AMASS platform has been implemented. This activity will be reported in the deliverable D5.6 Prototype for seamless interoperability [26].

2.2.3.3 V&V Manager

Table 8. Requirements covered by the V&V Manager

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_VVA_003	Validate requirements checking consistency, redundancy, … on formal properties	The system shall be able to validate formal requirements/properties	Solved	CHESS, OCRA, V&V Manager	FBK, HON, UOM
WP3_VVA_005	Verify (model checking) state machines	The system shall be able to verify the component behavioural model match with the specification	Solved	CHESS, NuXmv, V&VManager	FBK, HON, UOM
WP3_VVA_007	Generation of reports about system description/ verification results ….	The system shall generate reports about system/subsystem/component verification results	Pending	CHESS, V&VManager	FBK, HON

V&V Manager is an Eclipse plugin under development that enables invocation of multiple verification and validation tools, which process requirements or contracts directly from the AMASS platform. The V&V Manager for given requirements (and optionally also system architecture or design) connects to the verification server using OSLC Automation integration to get the V&V Assurance results. These results report whether the requirements are consistent, non-redundant, non-vacuous, and realizable; if system architecture or design information is available it also reports whether the given model complies with the requirements.

2.2.3.3.1 Implementation progress (*)

The V&V Manager plugin implementation was updated with new functionality (more human-readable language of contracts, merging of contracts of nested components possible). The verification status is now updated in the V&V Results view quasi-continually.

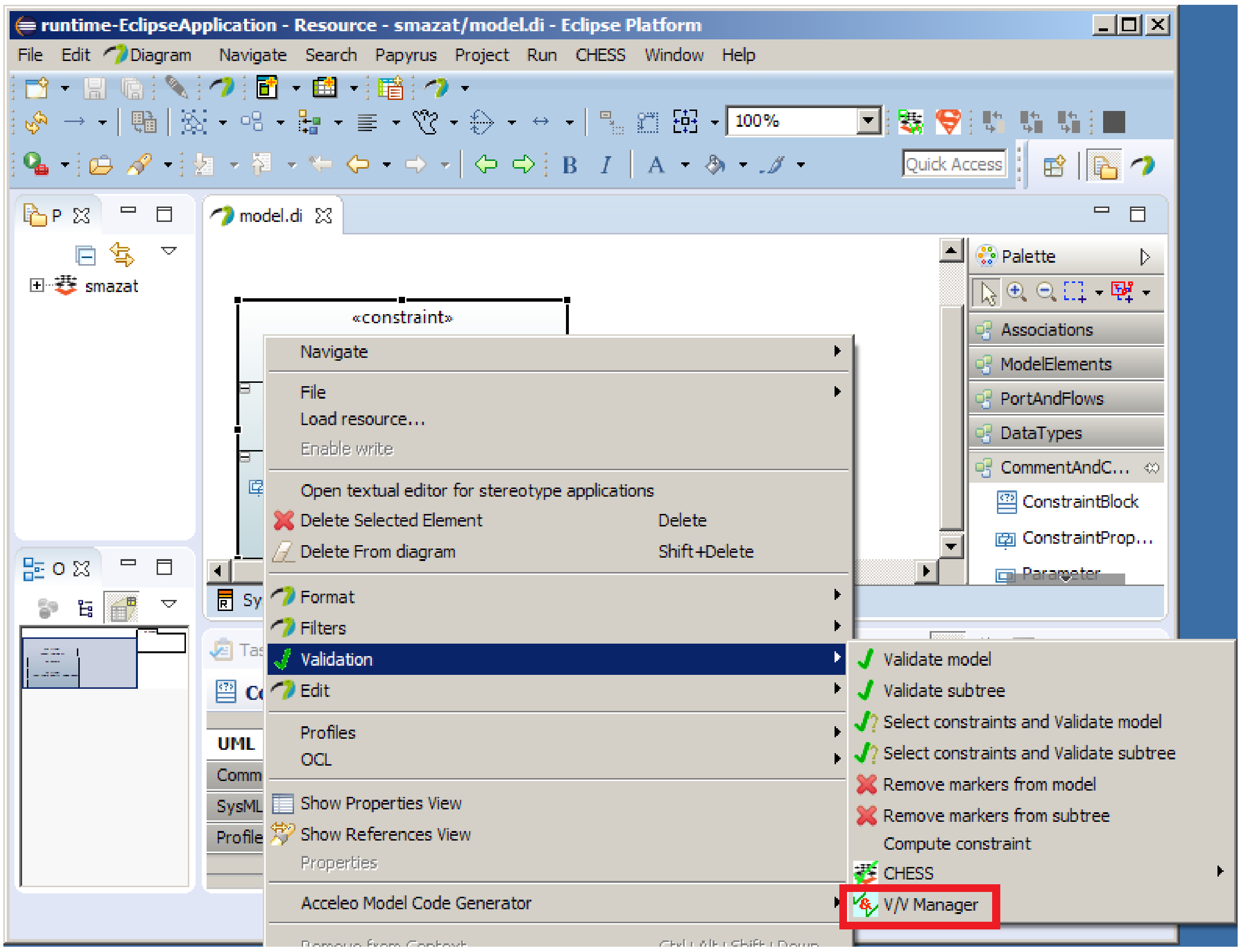

runtime-EclipseApplication - Resource - smazat/model.di - Eclipse Platform
File Edit Diagram Navigate Search Papyrus Project Run CHESS Window Help
model.di
«constraint»
Navigate
File
Load resource...
Enable write
Open textual editor for stereotype applications
Delete Selected Element Delete
Delete From diagram Shift+Delete
Format
Filters
Validation
Edit
Profiles
OCL
Show Properties View
Show References View
Properties
Acceleo Model Code Generator
Validate model
Validate subtree
Select constraints and Validate model
Select constraints and Validate subtree
Remove markers from model
Remove markers from subtree
Compute constraint
CHESS
V/V Manager
Palette
Associations
ModelElements
PortAndFlows
DataTypes
CommentAndC...
ConstraintBlock
ConstraintProp...

Figure 34. 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.

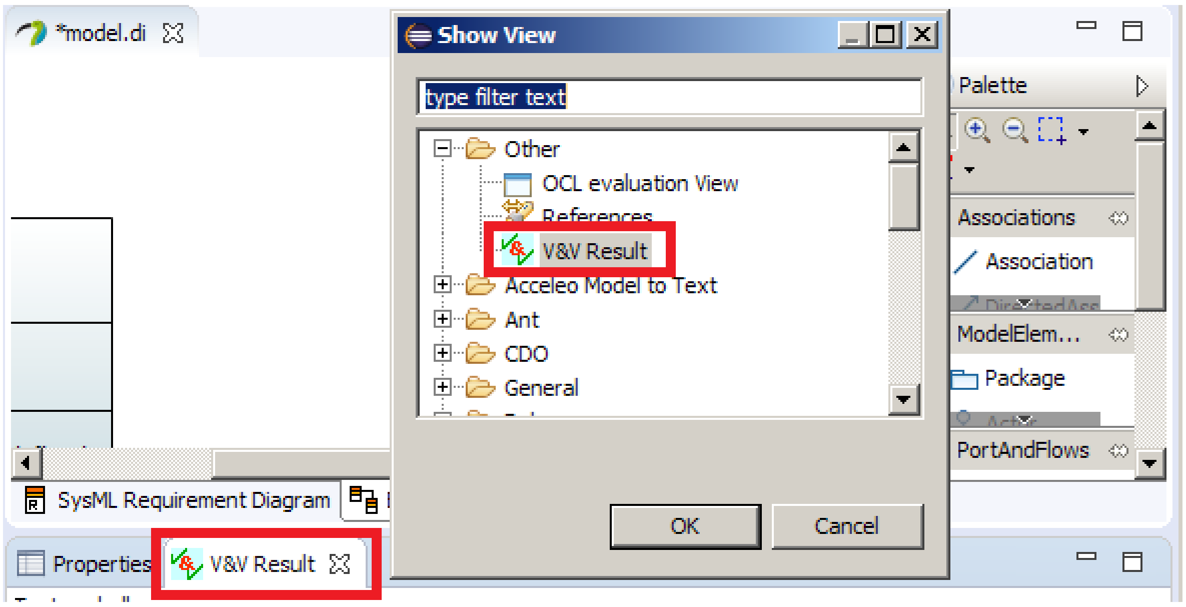
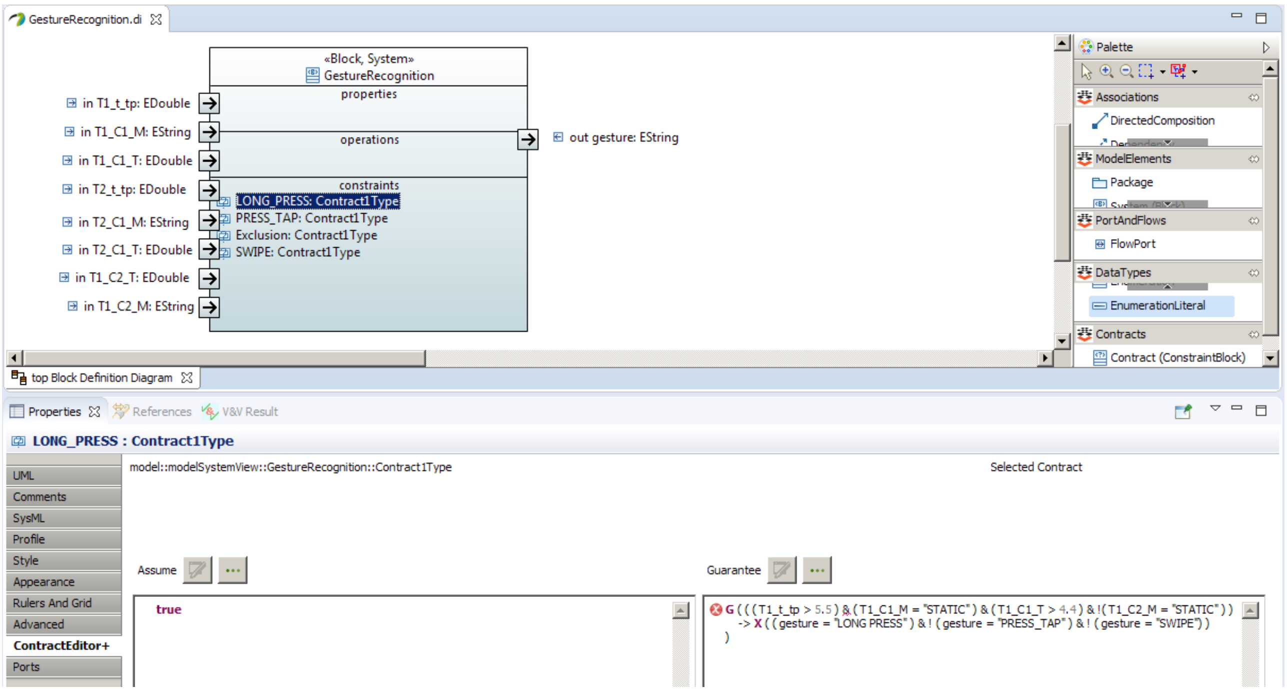


Figure 35. Switch in on the V&V Result view

The contracts are expected to be written in the LTL (Linear Temporal Logic) Language extended with arithmetic expressions and subset of MTL language (Metric Temporal Logic) syntax, allowing direct expression of real-time requirements with math equations. An example of the guarantee part of a contract is depicted in the Figure 36.The guarantee editor is tailored to the OCRA syntax and provides helpful hints about the inconsistencies in the text, e.g. underlining the names of variables having no counterpart in the list of ports. The remaining underline in the figure below is there because the syntax linked to the editor does not allow string literals, which are nevertheless allowed.



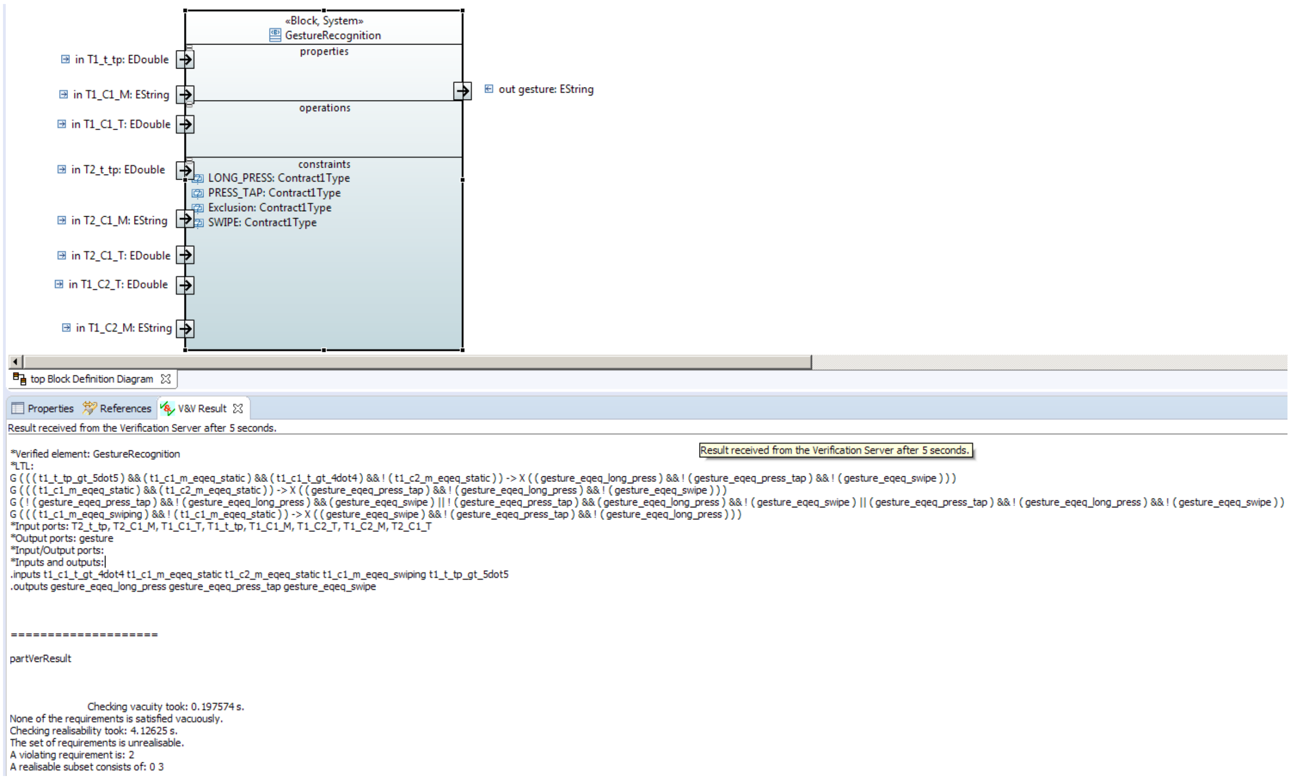
```
true
```

```
G ( ( ( T1_t_tp > 5.5 ) & ( T1_C1_M = "STATIC" ) & ( T1_C1_T > 4.4 ) & !( T1_C2_M = "STATIC" ) )
   -> X ( ( gesture = "LONG PRESS" ) & ! ( gesture = "PRESS_TAP" ) & ! ( gesture = "SWIPE") )
 )
```

Figure 36. Example of constraint's guarantee**.**

The presence of signal names as clearly isolated lexical elements enables their comparison to the relevant port names. Signal names that are used in the contract properties but not found among the port names are reported.

The formal properties are automatically translated to LTL before they are sent to the Verification Server. In the current stage of the plugin development it is beneficial to have the opportunity to visually inspect the inputs to the Verification Server. Therefore, the LTL representation of the contracts and the lists of input and output signals are displayed together with the V&V results, as can be seen in the Figure 37.



```
Result received from the Verification Server after 5 seconds.

*Verified element: GestureRecognition
*LTL:
G ( ( ( t1_t_tp_gt_5dot5 ) && ( t1_c1_m_eqeq_static ) && ( t1_c1_t_gt_4dot4 ) && ! ( t1_c2_m_eqeq_static ) ) -> X ( ( gesture_eqeq_long_press ) && ! ( gesture_eqeq_press_tap ) && ! ( gesture_eqeq_swipe ) ) )
G ( ( ( t1_c1_m_eqeq_static ) && ( t1_c2_m_eqeq_static ) ) -> X ( ( gesture_eqeq_press_tap ) && ! ( gesture_eqeq_long_press ) && ! ( gesture_eqeq_swipe ) ) )
G ( ! ( gesture_eqeq_press_tap ) && ! ( gesture_eqeq_long_press ) && ( gesture_eqeq_swipe ) || ! ( gesture_eqeq_press_tap ) && ( gesture_eqeq_long_press ) && ! ( gesture_eqeq_swipe ) || ( gesture_eqeq_press_tap ) && ! ( gesture_eqeq_long_press ) && ! ( gesture_eqeq_swipe ) )
G ( ( ( t1_c1_m_eqeq_swiping ) && ! ( t1_c1_m_eqeq_static ) ) -> X ( ( gesture_eqeq_swipe ) && ! ( gesture_eqeq_press_tap ) && ! ( gesture_eqeq_long_press ) ) )
*Input ports: T2_t_tp, T2_C1_M, T1_C1_T, T1_t_tp, T1_C1_M, T1_C2_T, T1_C2_M, T2_C1_T
*Output ports: gesture
*Input/Output ports:
*Inputs and outputs:
.inputs t1_c1_t_gt_4dot4 t1_c1_m_eqeq_static t1_c2_m_eqeq_static t1_c1_m_eqeq_swiping t1_t_tp_gt_5dot5
.outputs gesture_eqeq_long_press gesture_eqeq_press_tap gesture_eqeq_swipe



======================

partVerResult



              Checking vacuity took: 0.197574 s.
None of the requirements is satisfied vacuously.
Checking realisability took: 4.12625 s.
The set of requirements is unrealisable.
A violating requirement is: 2
A realisable subset consists of: 0 3
```

Figure 37. LTL going to and V&V results coming from the Verification Server

The findings of the V&V tools are also displayed in the V&V Results view, as shown in the Figure 37. Some tools may provide their answer almost immediately (which can be the case e.g. for checking of consistency or redundancy), while other tools take more time to terminate their tasks (e.g. realizability checking). Therefore, the V&V Manager monitors the consolidated response provided by the Verification Server and updates the V&V Results view in short periods of time in order to present the most recent status of validation and verification.

2.2.3.3.2 Verification Servers and Implementation Progress (*)

The communication between the 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 conforms 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 one public Verification Server (hosted at Masaryk University). Verification servers at Honeywell are not public since they also host 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 (increase its flexibility, make use of the traceability available in the AMASS Platform); the communication with the Verification Server is both fully specified and tested.

It is relatively easy to add another new OSLC provider similar to the existing Verification Server. The interaction between the V&V Manager and the new verification server will have to be completely tested. Therefore, each user can set up whatever selection of verification tools is required. 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.

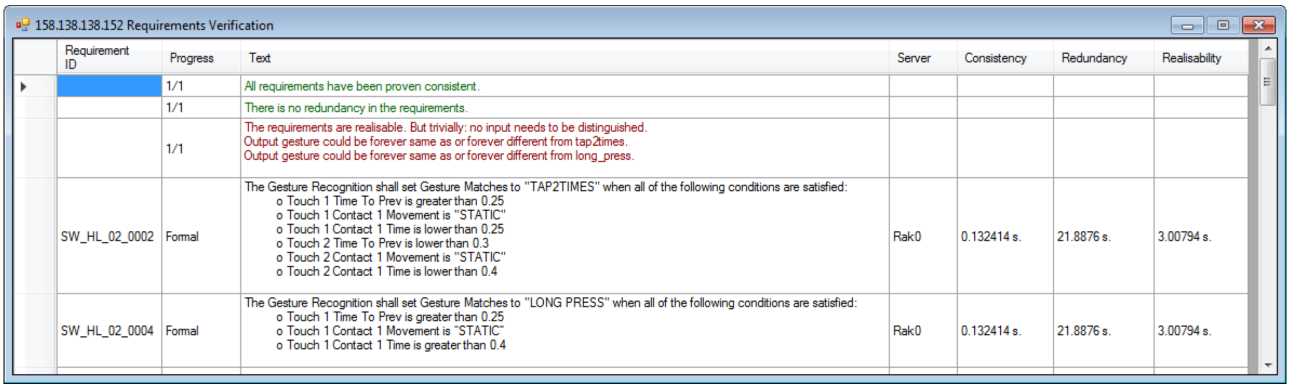
2.2.3.3.3 Semantic Requirement Analysis (*)

This V&V technique formally proves if a given set of formal requirements or contracts is consistent, non-redundant, non-vacuous, realizable and complete. Our approach is to execute multiple V&V tools and their configurations at once on multiple Verification Servers in parallel since often even V&V tool experts proficient in formal methods cannot determine which V&V tool and configuration will yield the best and/or quickest result. Moreover, especially for model checking this approach distributes the computationally expensive V&V tasks to multiple servers and 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 the AMASS platform after the V&V Manager implementation is finished. Figure 38 shows requirements that could be realised by some trivial system, which suggests that the requirements are incomplete. Figure 39 shows requirements that are non-realizable; more specifically the first two requirements are realizable and when the third requirement is added to them, it makes them unrealizable.

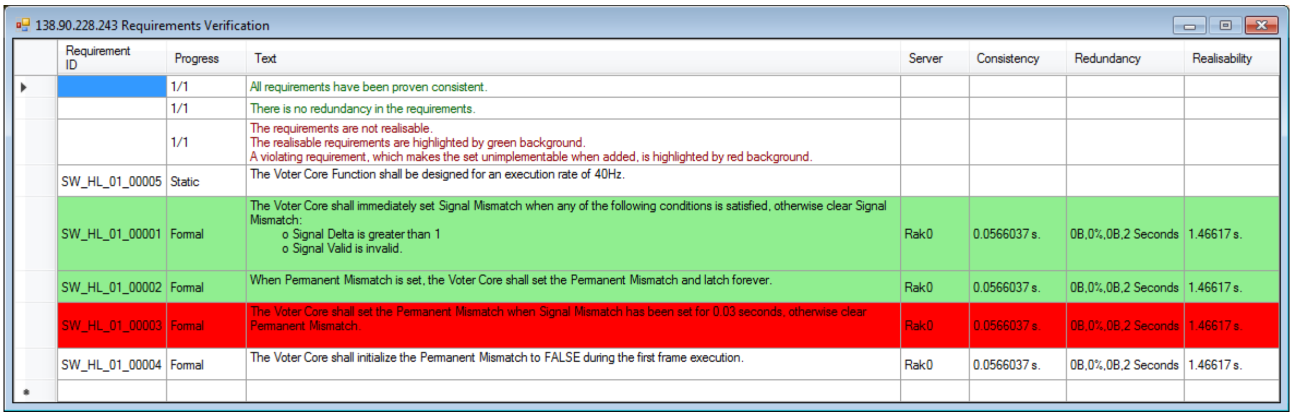
The analysis internally calls Acacia+ (which is a tool internally also called by the RQA tool mentioned above) to obtain the realizability witness: a strategy that prescribes what reactions to input signals will lead to requirements satisfaction. In addition to demonstrating that the requirements are 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.

Since performing realizability checking is often very time-consuming task, the requirements are also checked for logical consistency which is a weaker property than realizability, yet easier to compute. The consistency checking is performed by the tool ReMUS which was developed by Masaryk University. If the requirements are found to be inconsistent, ReMUS also identifies the minimal inconsistent subsets of the whole set of requirements, i.e. the sources of the inconsistencies. This information can be then used by the user to refine the requirements.


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: o Touch 1 Time To Prev is greater than 0.25 o Touch 1 Contact 1 Movement is "STATIC" o Touch 1 Contact 1 Time is lower than 0.25 o Touch 2 Time To Prev is lower than 0.3 o Touch 2 Contact 1 Movement is "STATIC" o Touch 2 Contact 1 Time is lower than 0.4	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: o Touch 1 Time To Prev is greater than 0.25 o Touch 1 Contact 1 Movement is "STATIC" o Touch 1 Contact 1 Time is greater than 0.4	Rak0	0.132414 s.	21.8876 s.	3.00794 s.

Figure 38. Example of requirements from Gesture Recognition system (Case Study 7) that are only trivially realisable


138.90.228.243 Requirements Verification

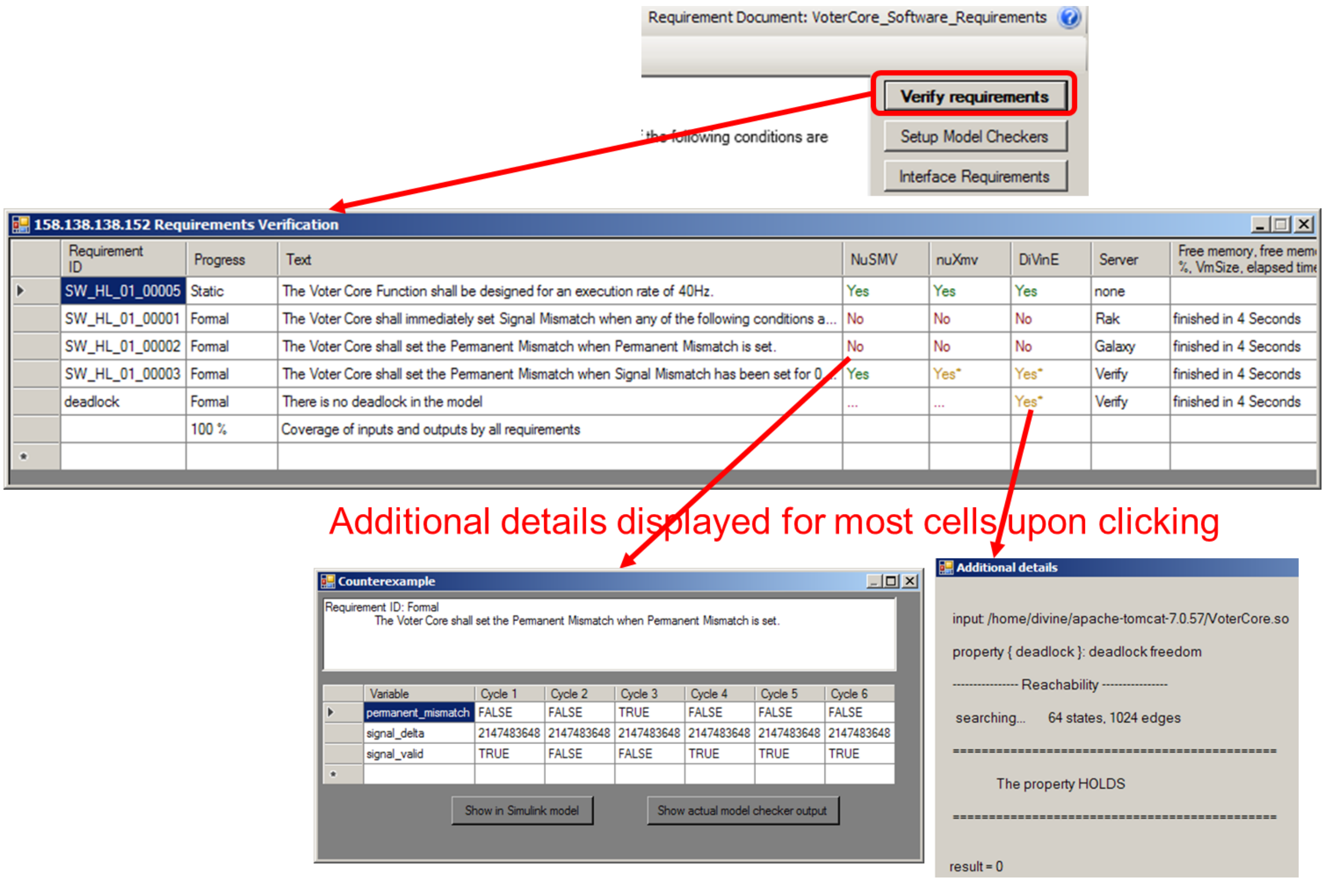
Requirement ID	Progress	Text	Server	Consistency	Redundancy	Realisability
	1/1	All requirements have been proven consistent.				
	1/1	There is no redundancy in the requirements.				
	1/1	The requirements are not realisable. The realisable requirements are highlighted by green background. A violating requirement, which makes the set unimplementable when added, is highlighted by red background.				
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: o Signal Delta is greater than 1 o Signal Valid is invalid.	Rak0	0.0566037 s.	0B,0%,0B,2 Seconds	1.46617 s.
SW_HL_01_00002	Formal	When Permanent Mismatch is set, the Voter Core shall set the Permanent Mismatch and latch forever.	Rak0	0.0566037 s.	0B,0%,0B,2 Seconds	1.46617 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	0.0566037 s.	0B,0%,0B,2 Seconds	1.46617 s.
SW_HL_01_00004	Formal	The Voter Core shall initialize the Permanent Mismatch to FALSE during the first frame execution.	Rak0	0.0566037 s.	0B,0%,0B,2 Seconds	1.46617 s.

Figure 39. Example of requirements that are consistent, non-redundant and not realisable

2.2.3.3.4 Formal Verification of Requirements against System Design

When system architecture or system design is available, 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 an example of a few requirements and the results from 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 their 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 a Simulink system design, a counterexample model that shows the falsifying behaviour.


Requirement Document: VoterCore_Software_Requirements
Verify requirements
Setup Model Checkers
Interface Requirements
the following conditions are
Additional details displayed for most cells upon clicking
Show in Simulink model
Show actual model checker output

158.138.138.152 Requirements Verification

Requirement ID	Progress	Text	NuSMV	nuXmv	DiVinE	Server	Free memory, free mem %, VmSize, elapsed time
SW_HL_01_00005	Static	The Voter Core Function shall be designed for an execution rate of 40Hz.	Yes	Yes	Yes	none	
SW_HL_01_00001	Formal	The Voter Core shall immediately set Signal Mismatch when any of the following conditions a...	No	No	No	Rak	finished in 4 Seconds
SW_HL_01_00002	Formal	The Voter Core shall set the Permanent Mismatch when Permanent Mismatch is set.	No	No	No	Galaxy	finished in 4 Seconds
SW_HL_01_00003	Formal	The Voter Core shall set the Permanent Mismatch when Signal Mismatch has been set for 0...	Yes	Yes*	Yes*	Verify	finished in 4 Seconds
deadlock	Formal	There is no deadlock in the model	...	...	Yes*	Verify	finished in 4 Seconds
	100 %	Coverage of inputs and outputs by all requirements					

Counterexample

Requirement ID: Formal
The Voter Core shall set the Permanent Mismatch when Permanent Mismatch is set.

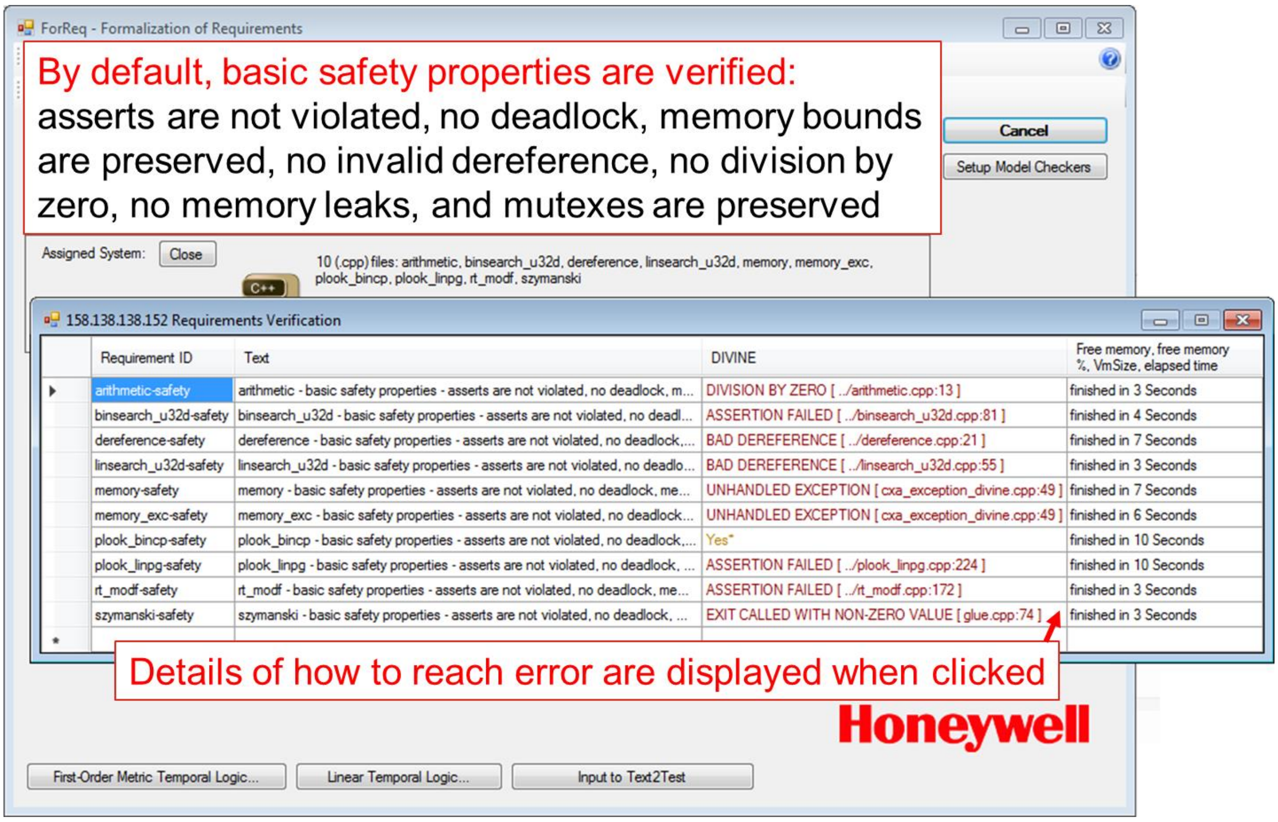
Variable	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6
permanent_mismatch	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE
signal_delta	2147483648	2147483648	2147483648	2147483648	2147483648	2147483648
signal_valid	TRUE	FALSE	FALSE	TRUE	TRUE	TRUE

Additional details

```
input: /home/divine/apache-tomcat-7.0.57/VoterCore.so
property { deadlock }: deadlock freedom
--------------- Reachability ---------------
 searching...    64 states, 1024 edges
==============================================
        The property HOLDS
==============================================

result = 0
```

Figure 40. Details for requirements checking

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


By default, basic safety properties are verified: asserts are not violated, no deadlock, memory bounds are preserved, no invalid dereference, no division by zero, no memory leaks, and mutexes are preserved

Assigned System: Close

10 (.cpp) files: arithmetic, binsearch_u32d, dereference, linsearch_u32d, memory, memory_exc, plook_bincp, plook_linpg, rt_modf, szymanski

158.138.138.152 Requirements Verification

Requirement ID	Text	DIVINE	Free memory, free memory %, VmSize, elapsed time
arithmetic-safety	arithmetic - basic safety properties - asserts are not violated, no deadlock, m...	DIVISION BY ZERO [../arithmetic.cpp:13]	finished in 3 Seconds
binsearch_u32d-safety	binsearch_u32d - basic safety properties - asserts are not violated, no deadl...	ASSERTION FAILED [../binsearch_u32d.cpp:81]	finished in 4 Seconds
dereference-safety	dereference - basic safety properties - asserts are not violated, no deadlock,...	BAD DEREFERENCE [../dereference.cpp:21]	finished in 7 Seconds
linsearch_u32d-safety	linsearch_u32d - basic safety properties - asserts are not violated, no deadlo...	BAD DEREFERENCE [../linsearch_u32d.cpp:55]	finished in 3 Seconds
memory-safety	memory - basic safety properties - asserts are not violated, no deadlock, me...	UNHANDLED EXCEPTION [cxa_exception_divine.cpp:49]	finished in 7 Seconds
memory_exc-safety	memory_exc - basic safety properties - asserts are not violated, no deadlock...	UNHANDLED EXCEPTION [cxa_exception_divine.cpp:49]	finished in 6 Seconds
plook_bincp-safety	plook_bincp - basic safety properties - asserts are not violated, no deadlock,...	Yes*	finished in 10 Seconds
plook_linpg-safety	plook_linpg - basic safety properties - asserts are not violated, no deadlock, ...	ASSERTION FAILED [../plook_linpg.cpp:224]	finished in 10 Seconds
rt_modf-safety	rt_modf - basic safety properties - asserts are not violated, no deadlock, me...	ASSERTION FAILED [../rt_modf.cpp:172]	finished in 3 Seconds
szymanski-safety	szymanski - basic safety properties - asserts are not violated, no deadlock, ...	EXIT CALLED WITH NON-ZERO VALUE [glue.cpp:74]	finished in 3 Seconds

Details of how to reach error are displayed when clicked

Figure 41. Checking and proposed error handling

2.2.3.4 Model Checking (*)

Table 9. Covered requirements regarding the verification of state machines

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_VVA_005	Verify (model checking) state machines	The system shall be able to verify the component behavioural model match with the specification	Solved	CHESS, NuXmv, V&VManager	FBK, HON, UOM

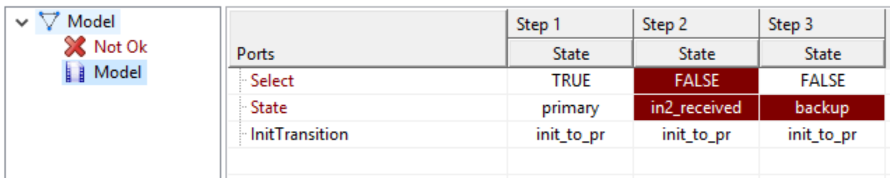
To check if the behaviour of the entire system or the behaviour of single components is compliant with a set of formal properties, CHESS interacts with nuXmv[15]. nuXmv is a symbolic model checker for the analysis of synchronous finite-state and infinite-state systems.

To perform the model checking command, the state machines that are used to describe the system behaviour, are translated in the SMV language[16] and stored as a file with the .smv extension. nuXmv takes

[15] https://nuxmv.fbk.eu

[16] The language used by xSAP to represent the nominal model, see http://nusmv.fbk.eu/NuSMV/papers/sttt_j/html/node7.html

in input the file, the list of formal properties and returns the result of the check. If the properties are not satisfied, a counterexample is shown in the dedicated “Behaviour Trace View” of CHESS (see Figure 42.).



	Step 1	Step 2	Step 3
Ports	State	State	State
Select	TRUE	FALSE	FALSE
State	primary	in2_received	backup
InitTransition	init_to_pr	init_to_pr	init_to_pr

Figure 42. Counterexample shown in the "Behaviour Trace View"

2.2.3.5 Generate fault trees from the behavioural model and the fault injection

Table 10. Covered requirements regarding generation of fault trees

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_SC_006	Specify component behavioural model (state machines)	The system shall be able to specify the component behavioural model	Solved	CHESS	FBK
WP3_VVA_010	Model-based safety analysis	The system shall allow the user to generate fault trees and FMEA tables from the behavioural model and the fault injection	Pending	CHESS, XSAP	INT, FBK

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[17].

CHESS implements a seamless integration with xSAP to allow the automatic generation of fault trees 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): SysML Blocks or UML Components with port definitions and composite relationships
- For each component:
  - The nominal behaviour, modelled using state machines; the activities in the state machine have to be specified using the NUSMV language[18].
  - The error behaviour, modelled by using a state machine stereotyped with the <<ErrorModel>> (see Figure 43) stereotype available from the CHESS dependability profile[19]. The CHESS dependability profile is also used to model error states, error propagation (e.g. InternalPropagation in Figure 43) and failure conditions (e.g. stuckAt value, inverted error) in component properties.

[17] https://xsap.fbk.eu/

[18] The language used by xSAP to represent the nominal model, see http://nusmv.fbk.eu/NuSMV/papers/sttt_j/html/node7.html

[19] 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 D3.3 [16] Appendix C.

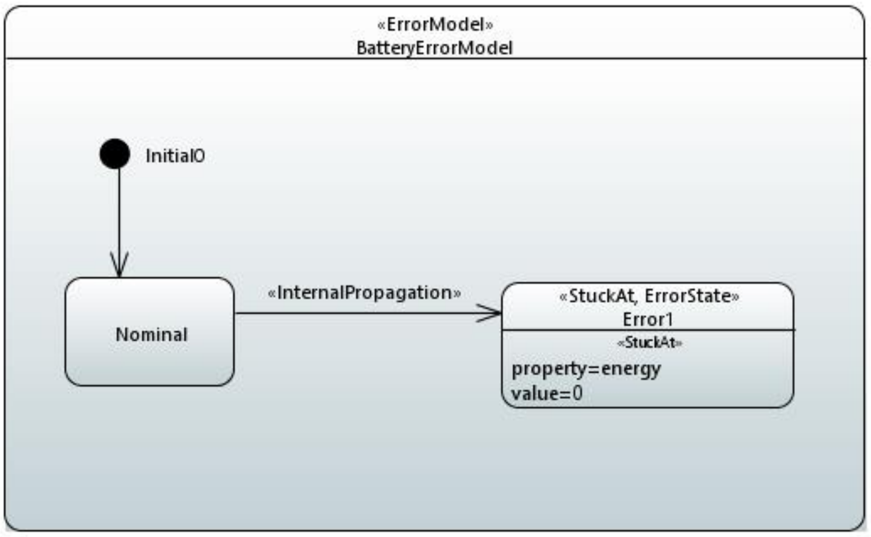
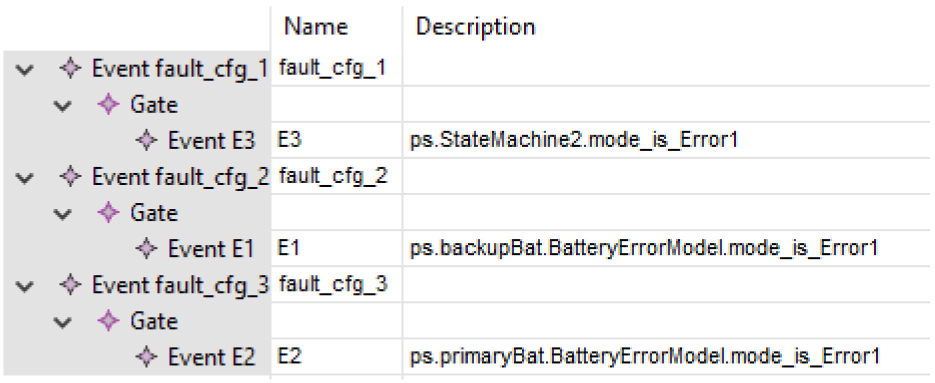


Figure 43. CHESS error model state machine

An initial integration between CHESS 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 CHESS profile, in particular of the CHESS Contract sub-profile. Moreover, some bugs have been discovered and fixed.

In the second prototype, CHESS provides a Fault Tree View to graphically represent the result of the analysis as a table or tree, see respectively Figure 44 and Figure 45.



	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 44. Example of fault tree represented as a table

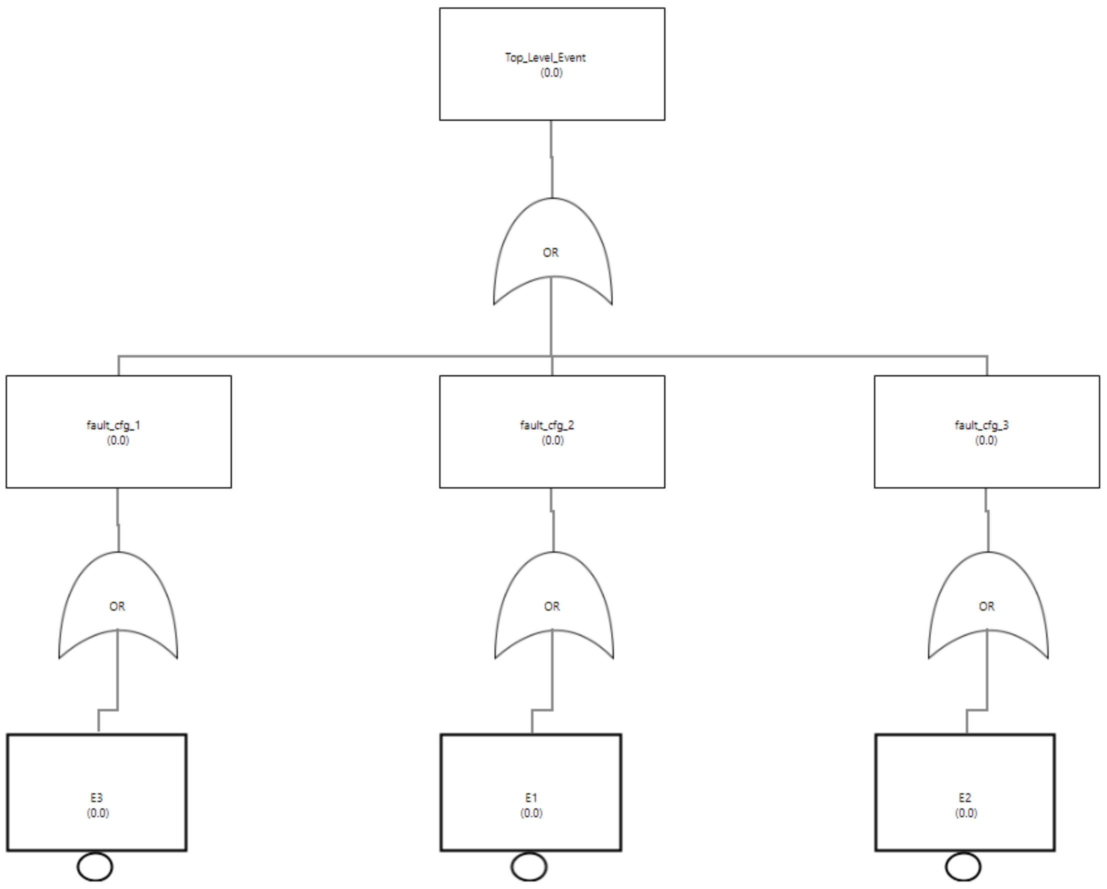


Figure 45. Example of fault tree represented as tree

2.2.3.6 Simulation-based Fault Injection (*)

Table 11. Requirements covering simulation-based fault injection

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_SC_007	Fault injection (include faulty behaviour of a component)	The system shall have fault injection capabilities	Solved	CHESS, SABOTAGE	INT, TEC
WP3_VVA_011	Simulation-based Fault Injection	The system should allow the user to generate fault injection simulations from the fault trees and FMEA tables	Pending	SABOTAGE	TEC, AIT, B&M

Model-based design combined with a simulation-based fault injection technique is a promising solution for the early safety assessment of systems. The fault injection functionality is supported by the Sabotage tool, which is based on a simulation fault injection framework. The Sabotage tool helps to specify different failures within a model-based system design performed in Matlab/Simulink. The Eclipse modelling framework (EMF) in combination with Massif [15], which converts from MATLAB Simulink models to EMF, supports the specification of failures with an intuitive fault list. Sabotage automatically adds "saboteur" blocks into Simulink models to reproduce those specific failures. The saboteur block is a Simulink s-function block, whose custom C code implements the injected fault. The tests are run and the results are analysed and visualised.

One of the main goals that AMASS promotes is the use and creation of model-based solutions. In that direction, the configuration, creation, run and visualisation of the fault injection experiments is developed into an Eclipse framework. In addition, this framework provides the communication with Matlab/Simulink tool.

Due to the use of Eclipse framework in the Sabotage tool, the user does not need to be familiar with the low-level configuration technologies for the automatic generation of fault injection experiments, for example, EMF, Xtend, Matlab/Simulink, Java and C code. Figure 46 depicts the technologies involved in integrating our Eclipse framework with Sabotage.

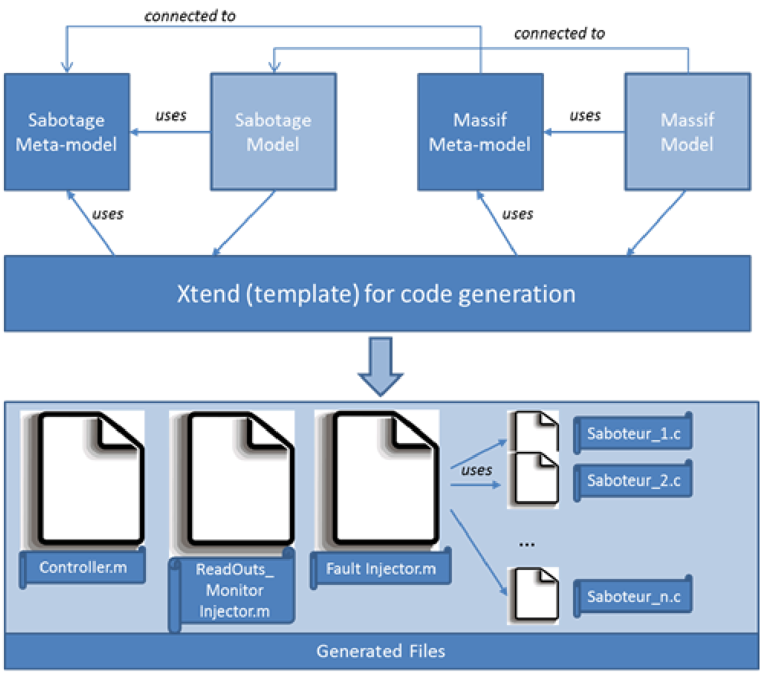


Figure 46. Sabotage design architecture.

Even though this functionality is released as part of Prototype P2, it extends work started in Prototype P1. Thus, some preliminary results and concepts are already covered in the D3.5 deliverable [21]. However, those results and concepts are improved for this third Prototype (P2).

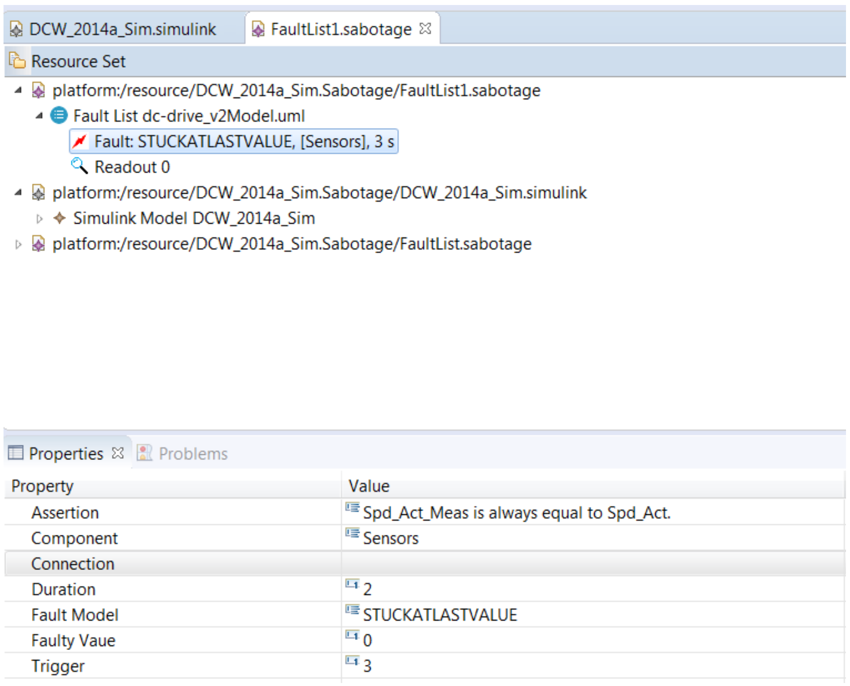
After investigating how to model the configuration of fault injection experiments, EMF technology has been chosen. The reason why, is the possibility to link Sabotage configuration to the Massif meta-model. The meta-model, which configures the experiments of the fault injection, is called Sabotage. The Sabotage meta-model is used to configure the failures that will be reproduced in the system. To establish where those failures will be injected, the Massif meta-model is used. The Massif meta-model, which is created based on EMF technology, provides the architecture of the system in a manner compatible with the Sabotage meta-model.

On the other hand, Model-to-text transformations are adopted for automation of the experiments. More specifically, the template language Xtend [14] is applied to generate Matlab and C code. Xtend technology includes a template language to generate code. As explained in D3.3, the Sabotage framework creates the golden (fault-free) and the faulty System Model Under Test (SMUT). The Xtend technology is employed to export the resulting C code that generates each failure, Matlab code to create a golden and a faulty SMUT, and Matlab code to execute the experiments and visualise the results. Xtend allows the creation of code replacing the dynamic areas of the template with information from a metamodel. In this case, that information comes from the Sabotage and Massif meta-models. 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 [16] is to define where to inject the faults. Those faults reproduce failure behaviours on certain components.

In Figure 47 a fault list is defined in the Sabotage model where all the faults, readouts and their properties are specified.



DCW_2014a_Sim.simulink | FaultList1.sabotage

Resource Set
- platform:/resource/DCW_2014a_Sim.Sabotage/FaultList1.sabotage
  - Fault List dc-drive_v2Model.uml
    - Fault: STUCKATLASTVALUE, [Sensors], 3 s
    - Readout 0
- platform:/resource/DCW_2014a_Sim.Sabotage/DCW_2014a_Sim.simulink
  - Simulink Model DCW_2014a_Sim
- platform:/resource/DCW_2014a_Sim.Sabotage/FaultList.sabotage

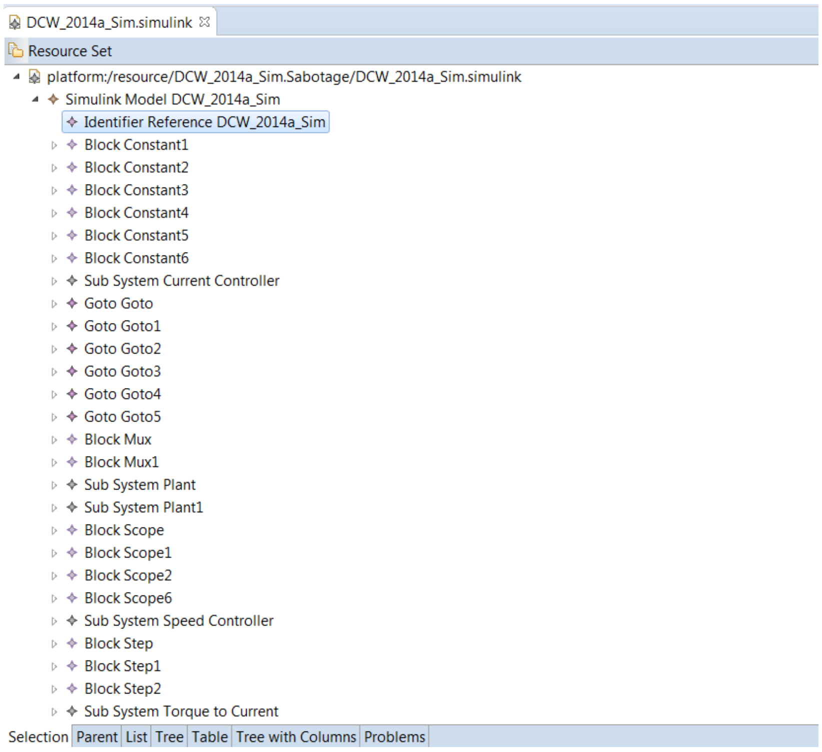
Properties | Problems

Property	Value
Assertion	Spd_Act_Meas is always equal to Spd_Act.
Component	Sensors
Connection	
Duration	2
Fault Model	STUCKATLASTVALUE
Faulty Vaue	0
Trigger	3

Figure 47. Sabotage Fault List

To extract the necessary information regarding possible injection points, Massif is used. The necessary information regarding possible injection points (i.e. Connection) is extracted, importing the Simulink model to Massif. This includes information regarding the architecture of the system and the connection between input and output ports. The same concept applies to the observation points, monitors or read-outs. This information needs to be specified based on the current system model.

An example architecture of a DC drive system defined in Massif model is shown in Figure 48.



DCW_2014a_Sim.simulink

Resource Set

- platform:/resource/DCW_2014a_Sim.Sabotage/DCW_2014a_Sim.simulink
  - Simulink Model DCW_2014a_Sim
    - Identifier Reference DCW_2014a_Sim
    - Block Constant1
    - Block Constant2
    - Block Constant3
    - Block Constant4
    - Block Constant5
    - Block Constant6
    - Sub System Current Controller
    - Goto Goto
    - Goto Goto1
    - Goto Goto2
    - Goto Goto3
    - Goto Goto4
    - Goto Goto5
    - Block Mux
    - Block Mux1
    - Sub System Plant
    - Sub System Plant1
    - Block Scope
    - Block Scope1
    - Block Scope2
    - Block Scope6
    - Sub System Speed Controller
    - Block Step
    - Block Step1
    - Block Step2
    - Sub System Torque to Current

Selection | Parent | List | Tree | Table | Tree with Columns | Problems

Figure 48. Massif model of the DC drive system

- Generation of the fault injection experiments:

One of the main remarkable features is the construction of the golden and the faulty Simulink models. After generation of the fault list, the Xtend technology creates Matlab code files (Figure 50), which construct the golden and faulty Simulink models. These Matlab code files create a golden Simulink model (fault-free system with readouts inside) and a faulty Simulink model (Figure 49) with the faults of the fault list and the readouts. The Matlab code files handles the generation of the C code files where the behaviour of faults are reproduced.

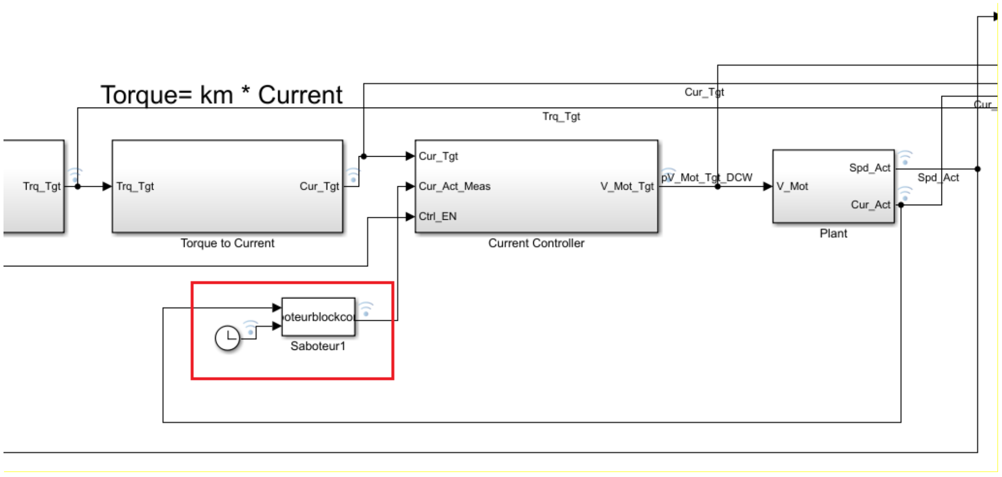
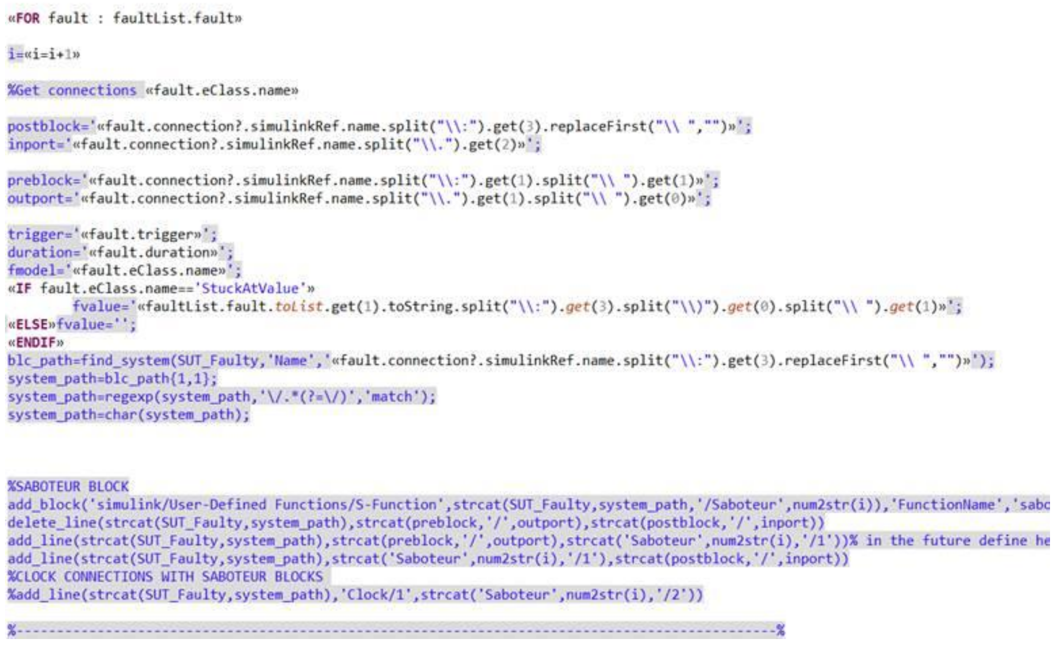


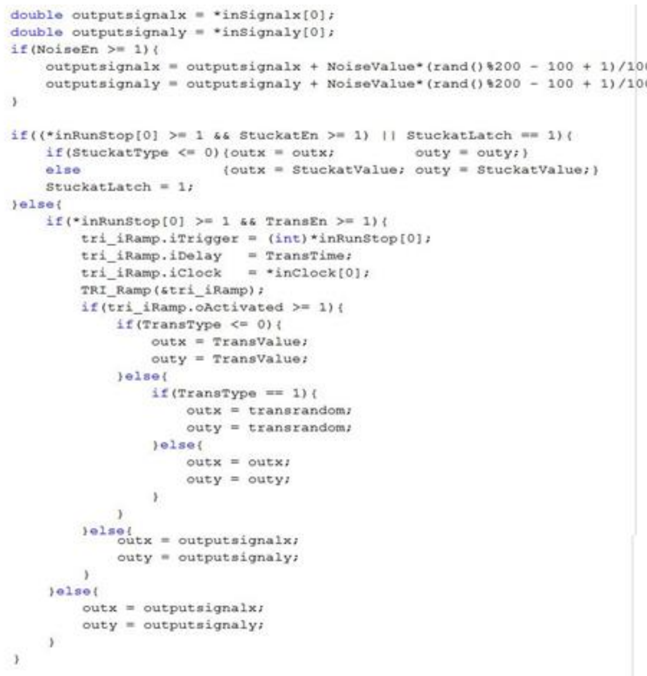
Figure 49. Example of the generated saboteur



```
«FOR fault : faultList.fault»

i=«i=i+1»

%Get connections «fault.eClass.name»

postblock='«fault.connection?.simulinkRef.name.split("\\:").get(3).replaceFirst("\\ ","")»';
inport='«fault.connection?.simulinkRef.name.split("\\.").get(2)»';

preblock='«fault.connection?.simulinkRef.name.split("\\:").get(1).split("\\ ").get(1)»';
outport='«fault.connection?.simulinkRef.name.split("\\.").get(1).split("\\ ").get(0)»';

trigger='«fault.trigger»';
duration='«fault.duration»';
fmodel='«fault.eClass.name»';
«IF fault.eClass.name=='StuckAtValue'»
        fvalue='«faultList.fault.toList.get(1).toString.split("\\:").get(3).split("\\)").get(0).split("\\ ").get(1)»';
«ELSE»fvalue='';
«ENDIF»
blc_path=find_system(SUT_Faulty,'Name','«fault.connection?.simulinkRef.name.split("\\:").get(3).replaceFirst("\\ ","")»');
system_path=blc_path{1,1};
system_path=regexp(system_path,'\/.*(?=\/)','match');
system_path=char(system_path);


%SABOTEUR BLOCK
add_block('simulink/User-Defined Functions/S-Function',strcat(SUT_Faulty,system_path,'/Saboteur',num2str(i)),'FunctionName','sabc
delete_line(strcat(SUT_Faulty,system_path),strcat(preblock,'/',outport),strcat(postblock,'/',inport))
add_line(strcat(SUT_Faulty,system_path),strcat(preblock,'/',outport),strcat('Saboteur',num2str(i),'/1'))% in the future define he
add_line(strcat(SUT_Faulty,system_path),strcat('Saboteur',num2str(i),'/1'),strcat(postblock,'/',inport))
%CLOCK CONNECTIONS WITH SABOTEUR BLOCKS
%add_line(strcat(SUT_Faulty,system_path),'Clock/1',strcat('Saboteur',num2str(i),'/2'))

%------------------------------------------------------------------------------------------%
```

Figure 50. Xtend templates for the generation of saboteurs and readouts

The Fault Injector creates and completes the C code (see Figure 51) of each saboteur for the corresponding S-functions block in Simulink. In the case where the fault is easily represented as a Simulink default blocks, e.g. random or delay failures, that block is used instead of an S-function blocks. This decision is related to fault representativeness, where it is tested that those blocks sufficiently represent the behaviour of those specific fault models.

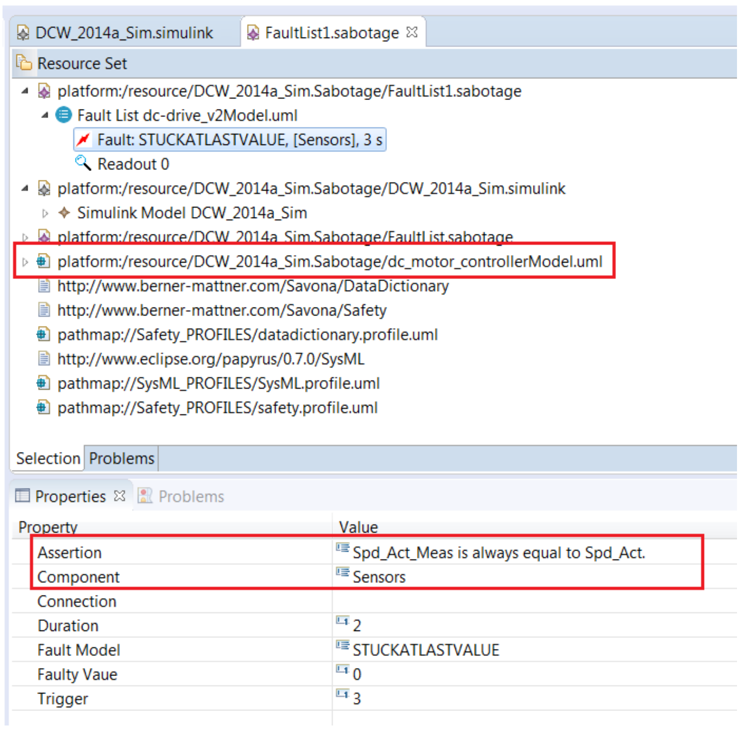


```
double outputsignalx = *inSignalx[0];
double outputsignaly = *inSignaly[0];
if(NoiseEn >= 1){
    outputsignalx = outputsignalx + NoiseValue*(rand()%200 - 100 + 1)/100;
    outputsignaly = outputsignaly + NoiseValue*(rand()%200 - 100 + 1)/100;
}

if((*inRunStop[0] >= 1 && StuckatEn >= 1) || StuckatLatch == 1){
    if(StuckatType <= 0){outx = outx;          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 = outx;
                    outy = outy;
                }
            }
        }else{
            outx = outputsignalx;
            outy = outputsignaly;
        }
    }else{
        outx = outputsignalx;
        outy = outputsignaly;
    }
}
```

Figure 51. Example of a saboteur code

Figure 52 illustrates how the integration of the Sabotage framework is carried out with respect to contracts. This allows to read the CHESS/Savona Model and relate information such as the aforementioned failure mode of a certain component defined as part of the system architecture.

During this the development of the last prototype iteration this work has been enhanced in order to investigate and add the following features as follows. The possible role and integration of other architecture-driven assurance functionalities with Sabotage have been 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 CHESS profile but not as part of the AMASS building block.



- platform:/resource/DCW_2014a_Sim.Sabotage/FaultList1.sabotage
  - Fault List dc-drive_v2Model.uml
    - Fault: STUCKATLASTVALUE, [Sensors], 3 s
    - Readout 0
- platform:/resource/DCW_2014a_Sim.Sabotage/DCW_2014a_Sim.simulink
  - Simulink Model DCW_2014a_Sim
- platform:/resource/DCW_2014a_Sim.Sabotage/FaultList.sabotage
- platform:/resource/DCW_2014a_Sim.Sabotage/dc_motor_controllerModel.uml
- http://www.berner-mattner.com/Savona/DataDictionary
- http://www.berner-mattner.com/Savona/Safety
- pathmap://Safety_PROFILES/datadictionary.profile.uml
- http://www.eclipse.org/papyrus/0.7.0/SysML
- pathmap://SysML_PROFILES/SysML.profile.uml
- pathmap://Safety_PROFILES/safety.profile.uml

Property	Value
Assertion	Spd_Act_Meas is always equal to Spd_Act.
Component	Sensors
Connection	
Duration	2
Fault Model	STUCKATLASTVALUE
Faulty Vaue	0
Trigger	3

Figure 52. Integration with safety contracts

2.2.3.7 Support for traceability between different kinds of V&V evidence

Table 12. Requirements covering traceability of different V&V artefacts

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_VVA_001	Traceability between different kinds of V&V evidence	The system shall provide the ability to trace immediate evidence (obtained during the execution of the left-hand side of the V-model) with direct evidence (obtained during the execution of the right-hand side of the V-model). For instance: a contract-based, component-based specification should be traced with the corresponding analysis-results.	Solved	CAPRA	INT
WP3_VVA_002	Trace model-to-model transformation	The system shall be able to trace all component model transformations executed during V&V model-based analysis	Pending	CAPRA	INT

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_VVA_004	Trace requirements validation checks	The system shall be able to trace requirements validations	Solved	Papyrus, CAPRA	INT

One of the AMASS requirements about architecture-driven assurance 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 to 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.

The CHESS modelling language has been extended with AMASS-specific analysis contexts; for instance, the new stereotype named *ContractRefinementAnalysisContext* (see Figure 53) identifies the information available in the CHESS model for a given contract refinement analysis. The aforementioned information is the set of components with associated contracts that has to be analysed; the CHESS model can comprise different views (e.g. functional, logical, physical) and different analyses can be run on each of the different views, or even different parts of the same view.

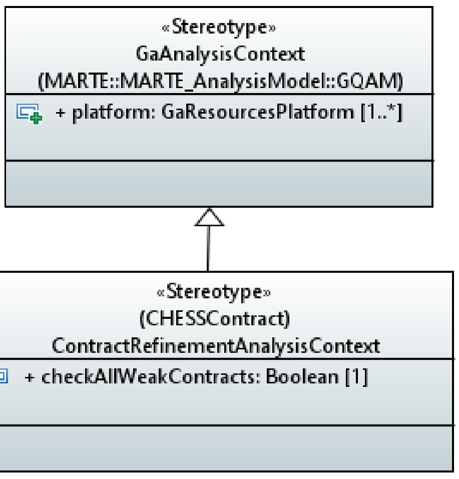

«Stereotype»
GaAnalysisContext
(MARTE::MARTE_AnalysisModel::GQAM)
+ platform: GaResourcesPlatform [1..*]
«Stereotype»
(CHESSContract)
ContractRefinementAnalysisContext
+ checkAllWeakContracts: Boolean [1]

Figure 53. Analysis Context

The ContractRefinementAnalysisContext stereotype also comes with a Boolean attribute *checkAllWeakContracts* which can be used; 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, the 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 hence to the target analysed set of components and associated contracts; this last step is not currently automated and must be made by the user using the traceability capabilities discussed in section 2.2.2.1.

2.2.3.8 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 into OCRA format. 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=>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 the CHESS tool to allow for property validation of the weak contract assumptions in OCRA. The results 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 the Validate Weak Contracts functionality.

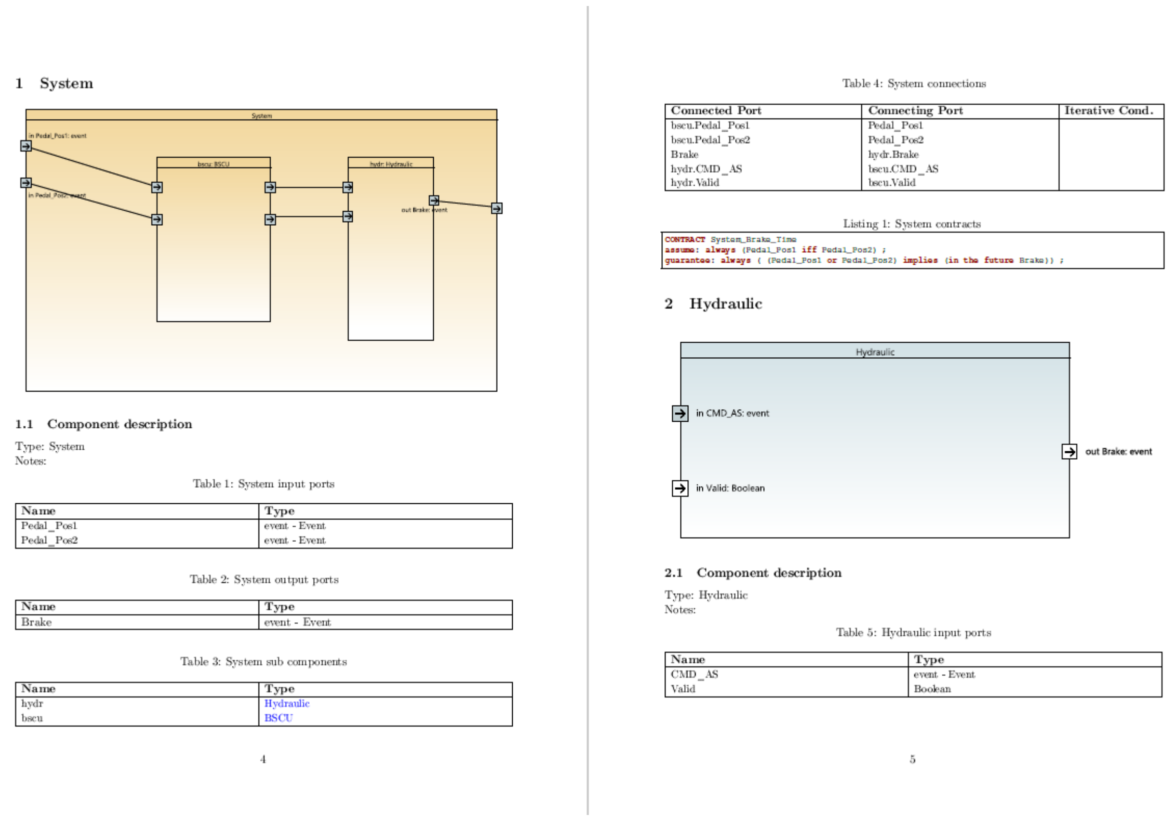
Based on the contract status we create a set of argument fragments in the corresponding assurance case project where they can be viewed in the assurance case editor. The generator uses a pre-existing argument pattern for the generation using information from the traceability editor of the contracts and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

2.2.3.9 Reports Generation (*)

Table 13. Requirements covering the generation of system reports

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_VVA_007	Generation of reports about system description/ verification results ….	The system shall generate reports about system/subsystem/component verification results	Pending	CHESS, V&VManager	FBK, HON

The third prototype (Prototype P2) enables the generation of reports stored in files. In particular, the tool can generate a document, either as doc or tex format, containing tables and diagrams as shown in Figure 54.



1 System

1.1 Component description

Type: System
Notes:

Table 1: System input ports

Name	Type
Pedal_Pos1	event - Event
Pedal_Pos2	event - Event

Table 2: System output ports

Name	Type
Brake	event - Event

Table 3: System sub components

Name	Type
hydr	Hydraulic
bscu	BSCU

4

Table 4: System connections

Connected Port	Connecting Port	Iterative Cond.
bscu.Pedal_Pos1	Pedal_Pos1	
bscu.Pedal_Pos2	Pedal_Pos2	
Brake	hydr.Brake	
hydr.CMD_AS	bscu.CMD_AS	
hydr.Valid	bscu.Valid	

Listing 1: System contracts

```
CONTRACT System_Brake_Time
assume: always (Pedal_Pos1 iff Pedal_Pos2) ;
guarantee: always ( (Pedal_Pos1 or Pedal_Pos2) implies (in the future Brake)) ;
```

2 Hydraulic

2.1 Component description

Type: Hydraulic
Notes:

Table 5: Hydraulic input ports

Name	Type
CMD_AS	event - Event
Valid	Boolean

5

Figure 54. Excerpt of 2 pages of the generated report.

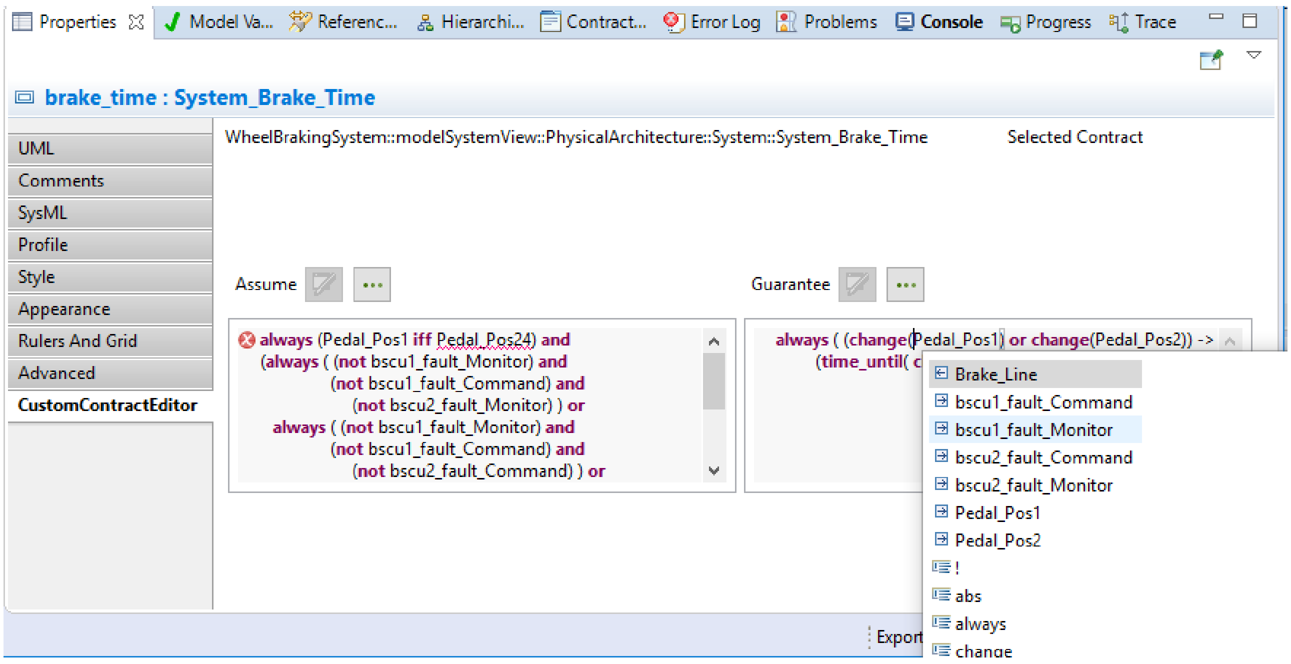
2.2.4 Contract-based Assurance Composition

2.2.4.1 Contract Editor with content assist

Table 14. Requirements covering the improvement of contract creation

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_CAC_009	Improvement of Contract definition process	The operation of contract definition should be improved in terms of time spent.	Solved	CHESS, SAVONA	FBK, B&M

In the AMASS prototype, the contract definition and the property definition can be edited using an editor with content assist, see Figure 55. 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.



brake_time : System_Brake_Time

WheelBrakingSystem::modelSystemView::PhysicalArchitecture::System::System_Brake_Time Selected Contract

Assume:

```
always (Pedal_Pos1 iff Pedal_Pos24) 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
```

Guarantee:

```
always ( (change(Pedal_Pos1) or change(Pedal_Pos2)) ->
    (time_until( c
```

Content assist: Brake_Line, bscu1_fault_Command, bscu1_fault_Monitor, bscu2_fault_Command, bscu2_fault_Monitor, Pedal_Pos1, Pedal_Pos2, !, abs, always, change

Figure 55. Contract Editor with content assist

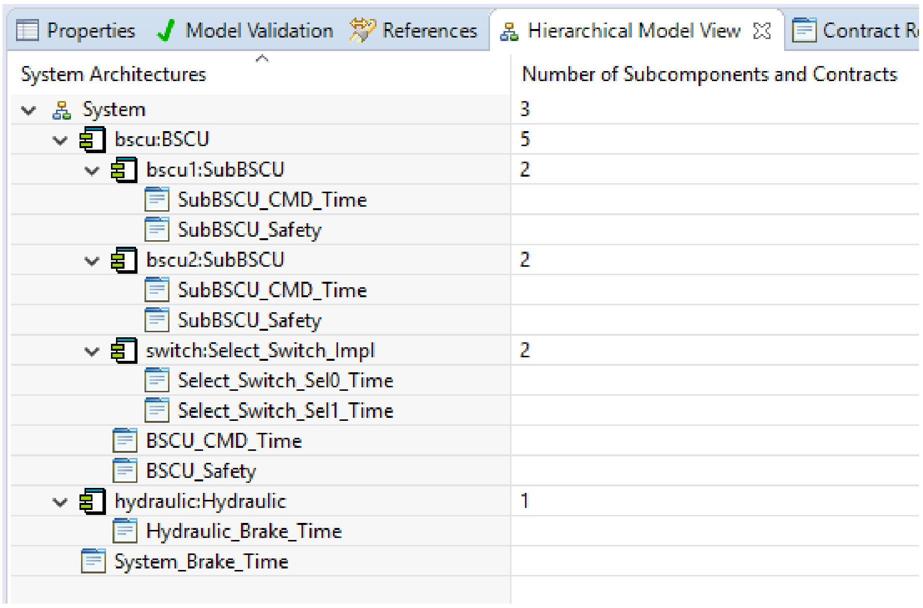
In this example, in the editing area of the assume property, an incorrect port name is highlighted. In the editing area of the guarantee property, it is suggested which are the compatible keywords or identifiers to insert.

2.2.4.2 Contract-based Views

Table 15. Requirements covering the overview of contracts

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_CAC_011	Overview of contract-based validation for behavioural models	The system could enable users to have an overview of the validation of a contract over a state-machine. In case of failure, the system could enable users to have information about the trace that does not fulfill the contract.	Solved	CHESS	FBK

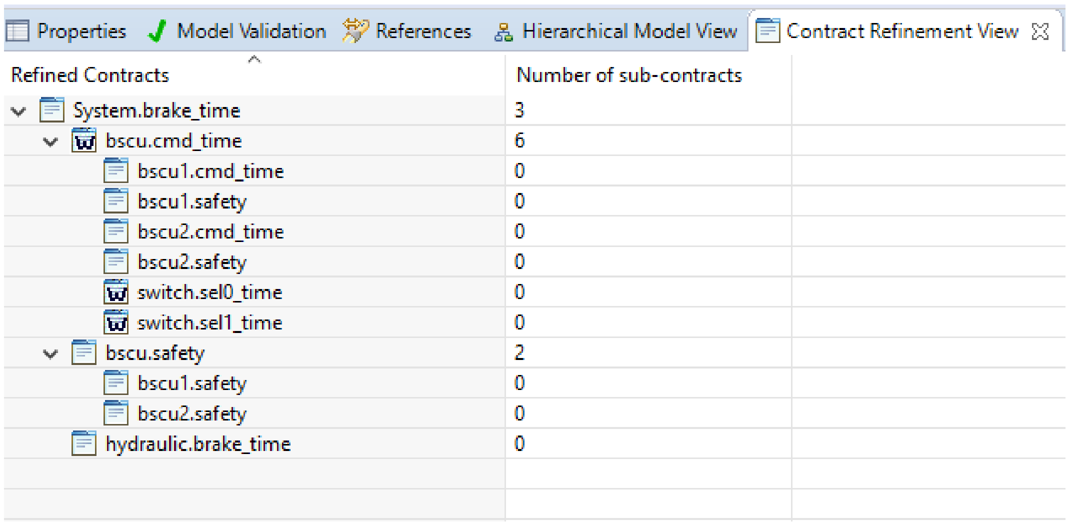
In the AMASS prototype, CHESS provides a hierarchical view that shows the decomposition of the system component into sub-components. It also shows the contracts assigned for each component. The system is represented graphically as the top element of the view (see Figure 56).



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 56. 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 57. The weak contracts are graphically represented as a document with a "W" on top.



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 57. Contract Refinement View

2.2.4.3 Contract refinement analysis

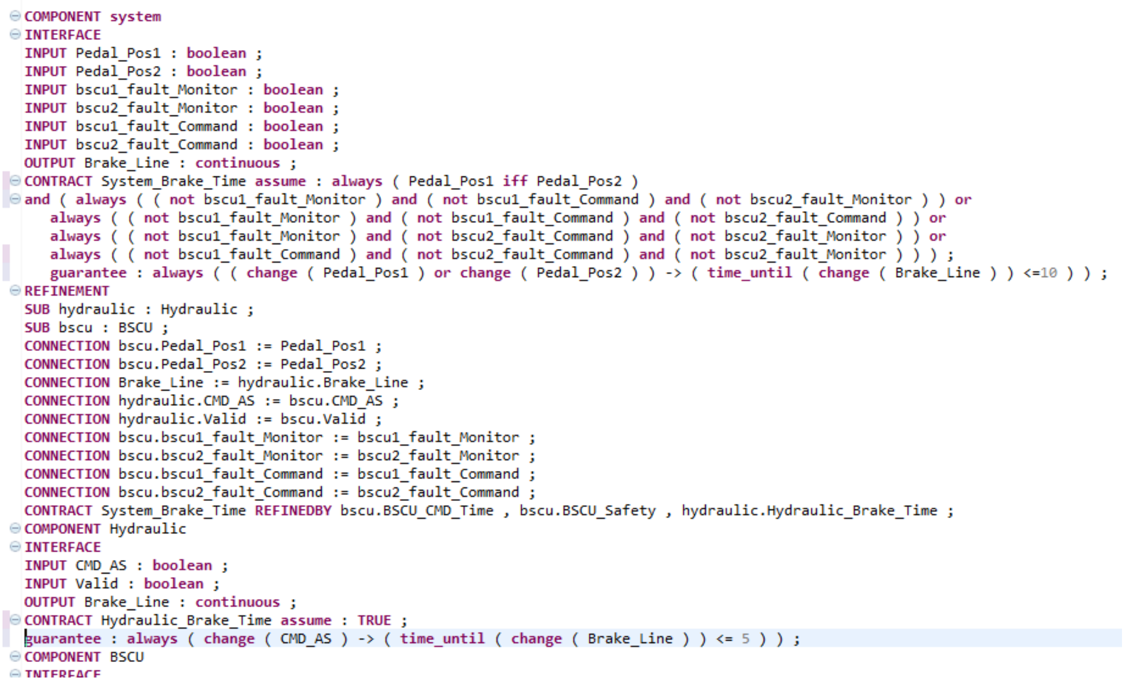
Table 16. Requirements covering contract refinement analysis

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_CAC_007	Overview of check refinements results	The system should enable users to have an overview in terms of status of check refinement of all the defined contracts.	Solved	CHESS	FBK
WP3_CAC_008	Contract-based validation and verification	The system must provide support for contract-based system validation and verification, including	Solved	CHESS	FBK

Requirement No	Name	Description	Status	Tools	Involved Partners
		refinement checking, compositional verification of behavioural models, contract-based fault-tree generation			

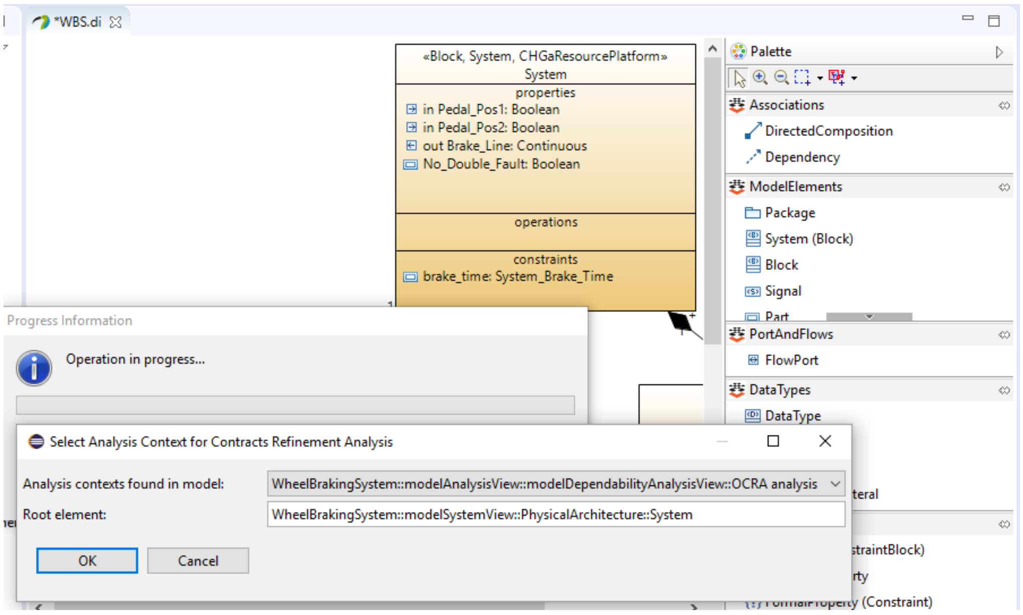
Contract refinement analysis is supported by the OCRA tool. CHESS comes with a seamless integration with OCRA which allows invocation of the analysis starting from the components and associated contracts available in the CHESS model. When the analysis is invoked through the CHESS tool the following steps 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 58).
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 58. 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)



«Block, System, CHGaResourcePlatform»
System
properties
in Pedal_Pos1: Boolean
in Pedal_Pos2: Boolean
out Brake_Line: Continuous
No_Double_Fault: Boolean
operations
constraints
brake_time: System_Brake_Time

Progress Information
Operation in progress...

Select Analysis Context for Contracts Refinement Analysis
Analysis contexts found in model: WheelBrakingSystem::modelAnalysisView::modelDependabilityAnalysisView::OCRA analysis
Root element: WheelBrakingSystem::modelSystemView::PhysicalArchitecture::System

Figure 59. Selecting analysis context for contract refinement

The integration between CHESS 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 CHESS profile, in particular those of the CHESS Contract sub-profile.

2.2.4.4 Contract-based Safety Analysis

The contract-based safety analysis detects the component failures as the failure of its implementation to satisfy 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. CHESS interacts with OCRA to produce a fault tree in which each intermediate event represents the failure of a component or its environment.

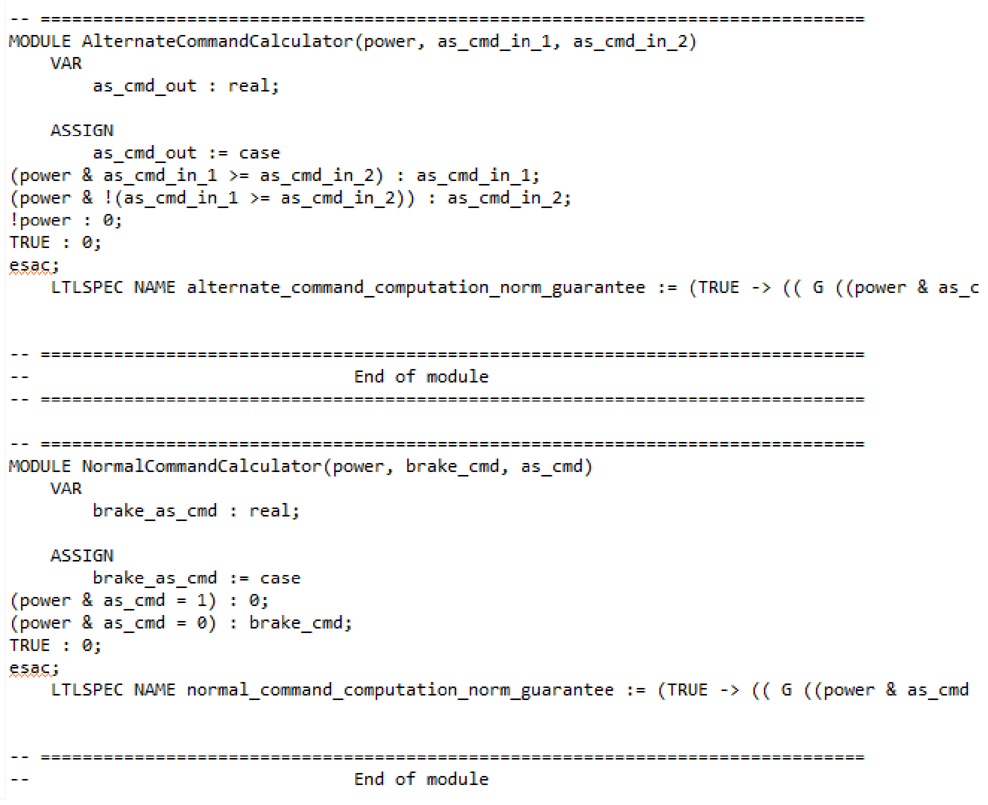
2.2.4.5 Contract-based verification of the behavioural model

Table 17. Requirements covering contract verification

Requirement No	Name	Description	Status	Tools	Involved Partners
WP3_CAC_001	Validate composition of components by validating their contracts	The system shall be able to validate the composition of components by supporting the validation of their contracts, analyzing the relationship among assumptions and guarantees	Solved	CHESS, OCRA	FBK
WP3_CAC_008	Contract-based validation and verification	The system must provide support for contract-based system validation and verification, including refinement checking, compositional verification of behavioural models, contract-based fault-tree generation	Solved	CHESS	FBK
WP3_CAC_012	Browse Contract status	The user shall be able to browse the contracts associated within a	Solved	CHESS	INT

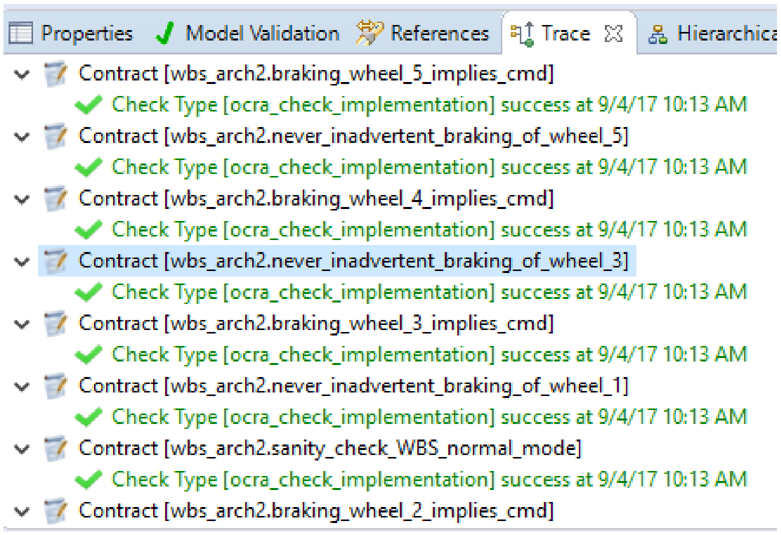
Requirement No	Name	Description	Status	Tools	Involved Partners
		component and their status (fulfilled or not)			
WP3_VVA_003	Validate requirements checking consistency, redundancy, … on formal properties	The system shall be able to validate formal requirements/properties	Solved	CHESS, OCRA, V&V Manager	FBK, HON, UOM

The Contract-based verification of the behavioural model is supported by the OCRA tool. This functionality verifies whether the finite state machines defined in the CHESS model verify the contracts. The state machines are translated into the SMV language, where the behaviour is described by means of logical formulas that describe the initial states and the state transitions; see Figure 60. Meanwhile, the contracts, as already mentioned in Section 2.2.3.3, are translated into the OCRA language in an .oss file. CHESS sends such information as input to OCRA, and then to the Trace View; for each contract the result of the check is shown, see Figure 61.



```
-- ===============================================================================
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 60. Part of an '.SMV' file representing the behaviour of the leaf components of the model



```
Contract [wbs_arch2.braking_wheel_5_implies_cmd]
    Check Type [ocra_check_implementation] success at 9/4/17 10:13 AM
Contract [wbs_arch2.never_inadvertent_braking_of_wheel_5]
    Check Type [ocra_check_implementation] success at 9/4/17 10:13 AM
Contract [wbs_arch2.braking_wheel_4_implies_cmd]
    Check Type [ocra_check_implementation] success at 9/4/17 10:13 AM
Contract [wbs_arch2.never_inadvertent_braking_of_wheel_3]
    Check Type [ocra_check_implementation] success at 9/4/17 10:13 AM
Contract [wbs_arch2.braking_wheel_3_implies_cmd]
    Check Type [ocra_check_implementation] success at 9/4/17 10:13 AM
Contract [wbs_arch2.never_inadvertent_braking_of_wheel_1]
    Check Type [ocra_check_implementation] success at 9/4/17 10:13 AM
Contract [wbs_arch2.sanity_check_WBS_normal_mode]
    Check Type [ocra_check_implementation] success at 9/4/17 10:13 AM
Contract [wbs_arch2.braking_wheel_2_implies_cmd]
```

Figure 61. 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 final 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.

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.

A methodological guideline on how to use the presented tools will be published within D3.8 [23].

4. Implementation Description

4.1 Implemented Modules

4.1.1 System Component Specification Block

As documented in AMASS deliverable D2.4 [7], the System Component Specification logical building block decomposes into two sub-blocks (see Figure 62): 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 CHESS plugins. In particular, Papyrus contains plugins for editing the architectural/component-based models, together with the possibility to model requirements (by using the SysML profile support). CHESS provides plugins for management of formal properties and contract specifications and their association with the architectural components.

The CHESS profile for Contract (see D3.1 [10]) is implemented as a UML/SysML profile; the profile has been designed using the Papyrus editor facilities.

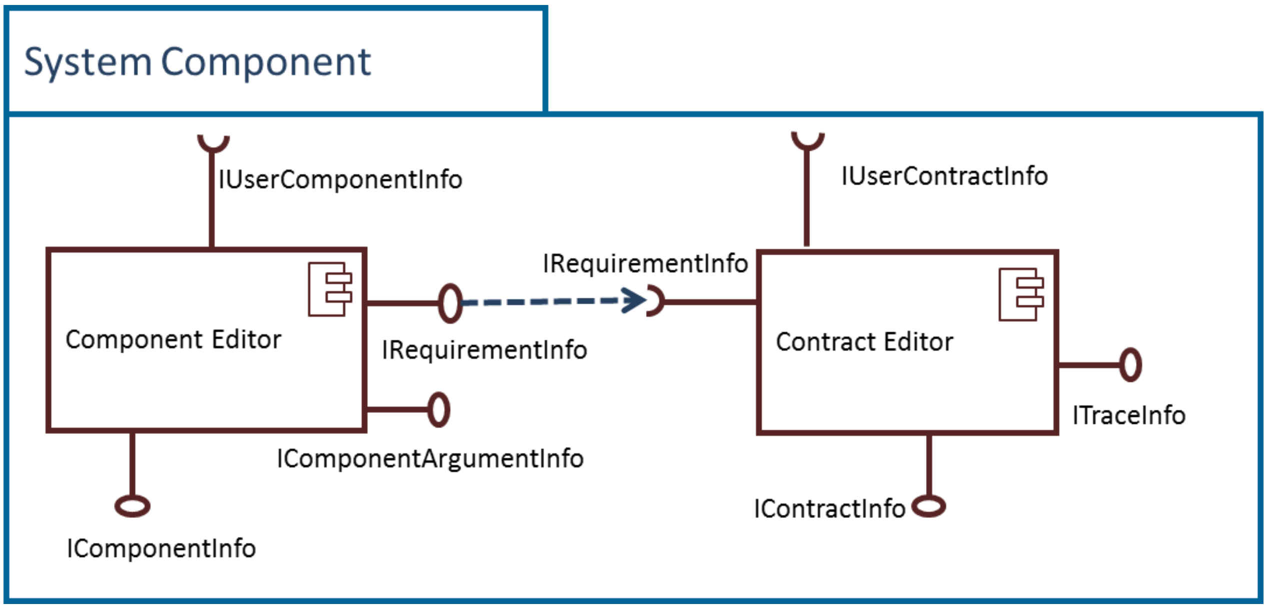


Figure 62. Tool modules for System Component Specification

4.1.2 Architecture-Driven Assurance Block

As documented in AMASS deliverable D2.4 [7], 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.

4.2 Source Code Description

4.2.1 System Component Specification Block

Papyrus[20] is an Eclipse project and its source code is freely available through the Eclipse GIT server[21].

The source code of the CHESS contract editor is available through the Polarsys CHESS project[22].

Extensions to the Polarsys CHESS 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). Then, once the extensions are sufficiently mature, they will be pushed to the Polarsys CHESS repository as AMASS contributions.

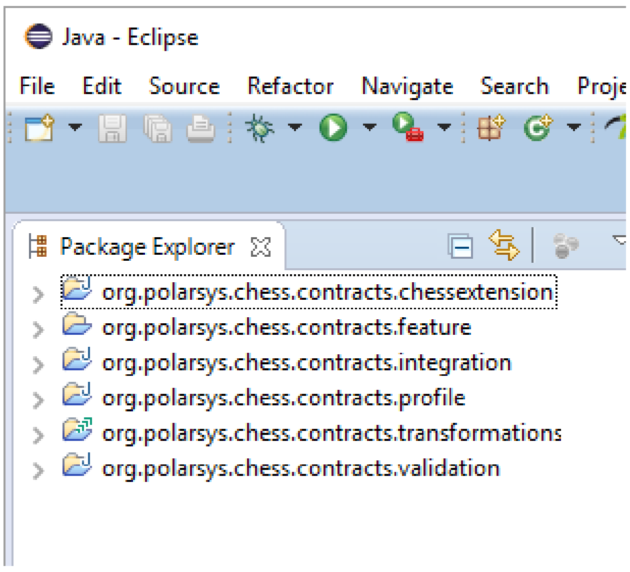
The additional CHESS plugins that need to be installed on top of Papyrus environment to enable the CHESS-based AMASS Contract Editor features are the following (see also Figure 63):

- `org.polarsys.chess.contracts.chessextension`: provides the Papyrus extension to easily work with the CHESS Contract profile, for instance to facilitate the creation of CHESS 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 CHESS profile for contracts.
- `org.polarsys.chess.contracts.transformations`: implements the model of text transformation for 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 CHESS profile. The plugin adds a dedicated command to the CHESS Eclipse menu to invoke the transformations.
- `org.polarsys.chess.contracts.validation`: implements the validation of the constraints that the CHESS 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 CHESS 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 with a component.

20 https://eclipse.org/papyrus/

21 https://git.eclipse.org/c/papyrus/org.eclipse.papyrus.git/

22 https://git.polarsys.org/c/chess/chess.git?h=develop



org.polarsys.chess.contracts.chessextension
org.polarsys.chess.contracts.feature
org.polarsys.chess.contracts.integration
org.polarsys.chess.contracts.profile
org.polarsys.chess.contracts.transformations
org.polarsys.chess.contracts.validation

Figure 63. CHESS plugins supporting Contract Based Design

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

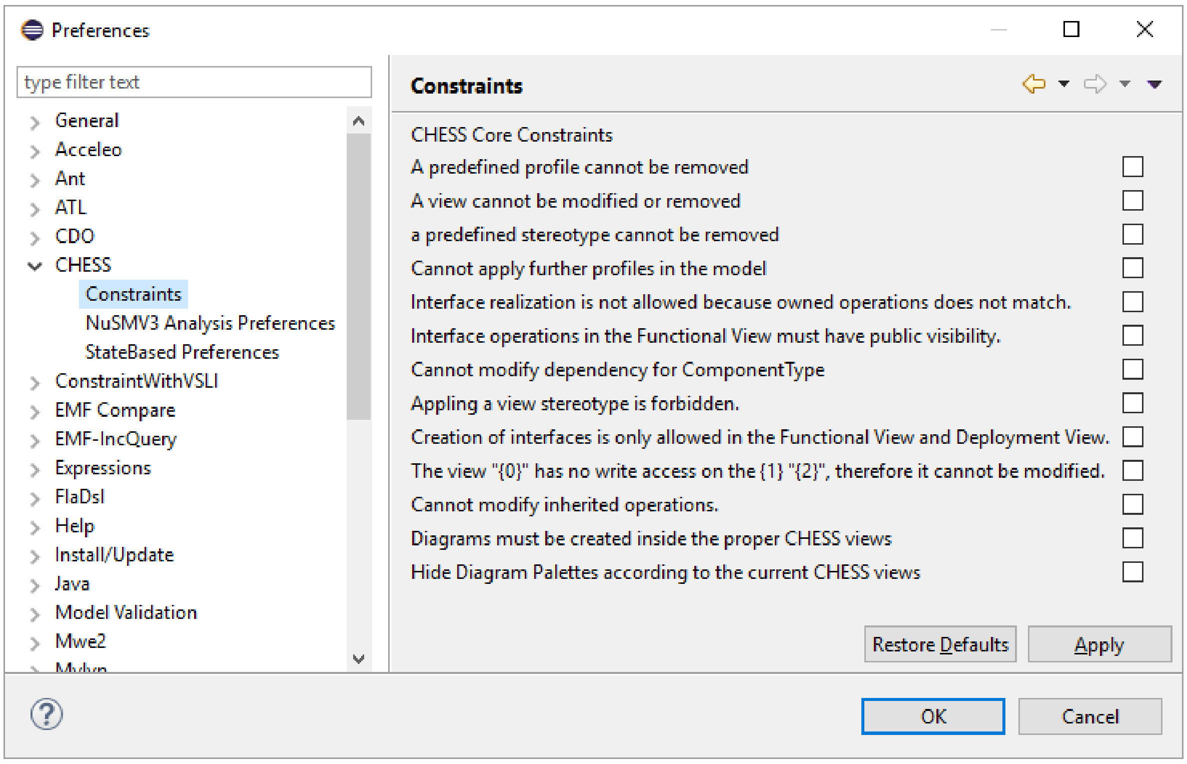
In particular, the Polarsys CHESS project provides a set of *core* plugins that allow the application of the CHESS methodology ([1][2]). This is the base upon which the AMASS methodology has been build. The actual CHESS methodology allows the design, verification and implementation of cyber physical software systems; CHESS adopts a dedicated component model language [4] and ad-hoc model transformations to enable timing/dependability analysis and code generation. Moreover, the CHESS 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 CHESS plugins extend the Papyrus editor to support the CHESS modelling language and design-by-view approach; so, by using the CHESS Papyrus extension, the constraints imposed by the CHESS 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 CHESS.

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

- `org.polarsys.chess.core`: provides some facilities regarding selections and diagram status.
- `org.polarsys.chess.services`: provides functionalities about the CHESS 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 CHESS modelling language [3]. Moreover, it provides dedicated diagram palettes extending the Papyrus ones to easily manage the creation of CHESS stereotypes in a given diagram. Therefore, CHESS core plugins are required in order to use the CHESS Contract feature.
- `org.polarsys.chess.diagramsCreator`: enables the creation of SysML/UML diagrams taking as input the CHESS entities located in the model.
- `org.polarsys.chess.OSSImporter`: provides functionalities to parse a OSS file and to populate a CHESS project with the entities that are imported.

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

Figure 64 below provides a snapshot of the CHESS methodology constraints that can be enabled/disabled through the Eclipse preferences page.



Constraints

CHESS Core Constraints

- A predefined profile cannot be removed ☐
- A view cannot be modified or removed ☐
- a predefined stereotype cannot be removed ☐
- Cannot apply further profiles in the model ☐
- Interface realization is not allowed because owned operations does not match. ☐
- Interface operations in the Functional View must have public visibility. ☐
- Cannot modify dependency for ComponentType ☐
- Appling a view stereotype is forbidden. ☐
- Creation of interfaces is only allowed in the Functional View and Deployment View. ☐
- The view "{0}" has no write access on the {1} "{2}", therefore it cannot be modified. ☐
- Cannot modify inherited operations. ☐
- Diagrams must be created inside the proper CHESS views ☐
- Hide Diagram Palettes according to the current CHESS views ☐

Figure 64. CHESS methodology constraint

4.2.2 Architecture-Driven Assurance Block

4.2.2.1 Requirements Formalization with Temporal Logics – RQA approach

To create the custom-coded metric needed to detect linear temporal logic consistency issues in the requirements, it 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 to choose this field using the provided assembly.
- **Method**: the name of the method containing the code of the metric. RQA should help you to choose this field using the provided class.

4.2.2.2 Simulation-based Fault Injection

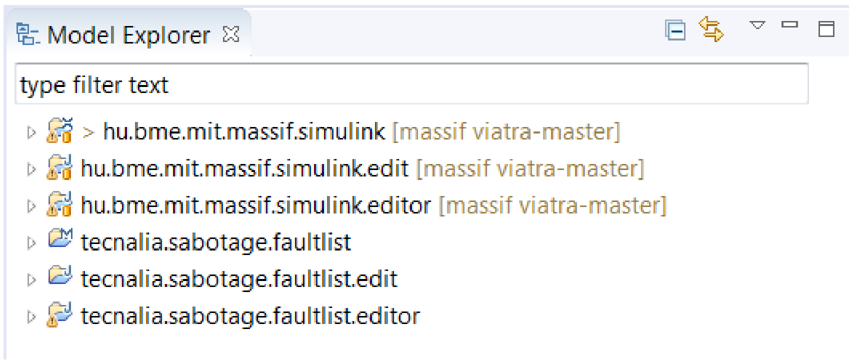
As specified in Section 2.2.3.6 the MASSIF Simulink Integration Framework for Eclipse is used for accessing Simulink model information. Importing uses the command line interface of Matlab rather than directly parsing mdl or slx files. This is the API recommended by MathWorks for accessing Simulink model information.

Following the procedure explained in [https://github.com/viatra/massif], the user installs MASSIF into the Eclipse Neon environment. The most important 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/. Note 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.

Note also the 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.
- hu.bme.mit.massif.simulink → Massif meta-model is designed to store all information for each MATLAB block of a system.


Model Explorer

type filter text

- > hu.bme.mit.massif.simulink [massif viatra-master]
- hu.bme.mit.massif.simulink.edit [massif viatra-master]
- hu.bme.mit.massif.simulink.editor [massif viatra-master]
- tecnalia.sabotage.faultlist
- tecnalia.sabotage.faultlist.edit
- tecnalia.sabotage.faultlist.editor

Figure 65. Massif and Sabotage meta-models

It is necessary to load the Massif model system to the fault list to help the user to define where the faults are injected into that specific Massif model. To do that “load resource” functionality needs to be carried out at meta-model and model level in order to establish the connection. This means that not only the Sabotage meta-model and Massif meta-model are connected, but also when the user is using Sabotage model needs to link a Simulink model with a Sabotage model.

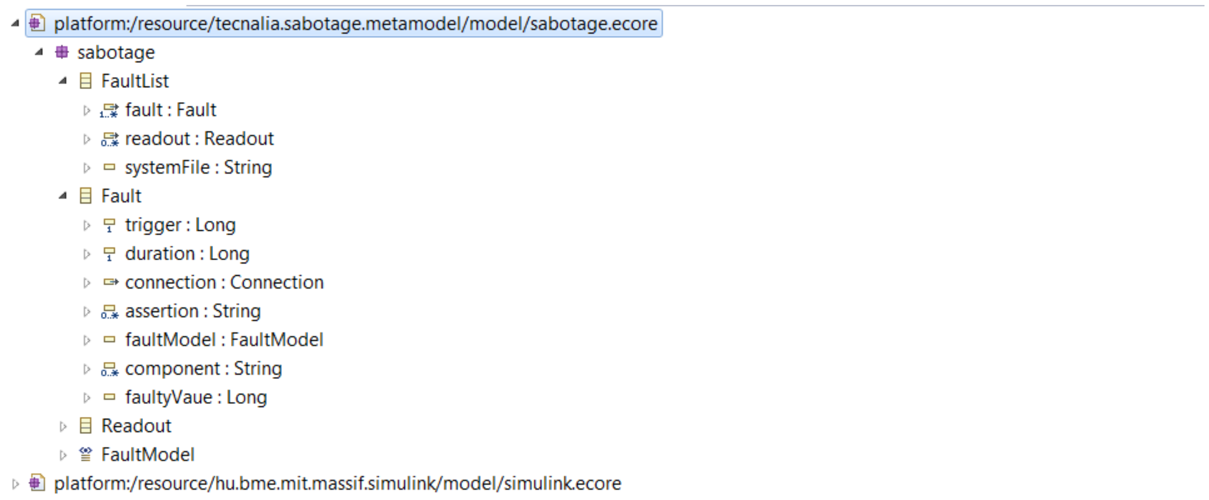
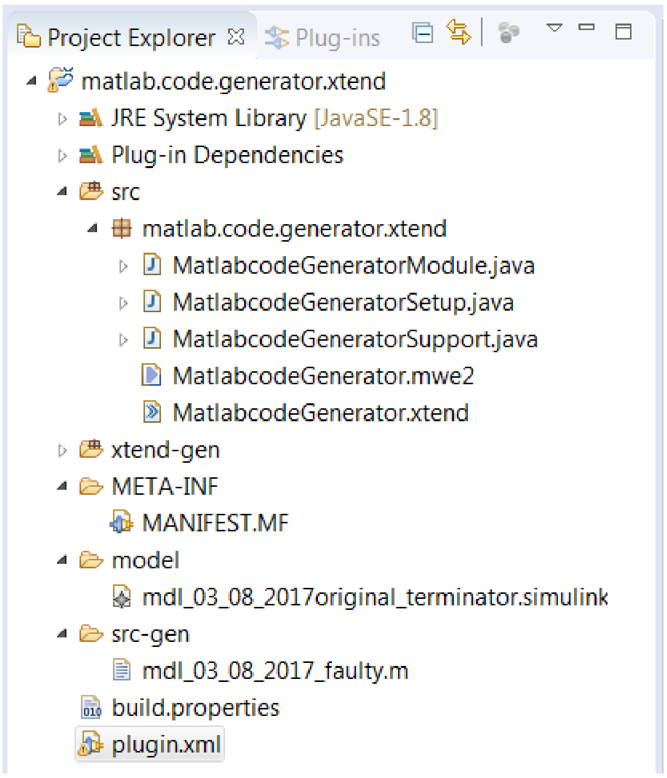

```
platform:/resource/tecnalia.sabotage.metamodel/model/sabotage.ecore
  sabotage
    FaultList
      fault : Fault
      readout : Readout
      systemFile : String
    Fault
      trigger : Long
      duration : Long
      connection : Connection
      assertion : String
      faultModel : FaultModel
      component : String
      faultyVaue : Long
    Readout
    FaultModel
platform:/resource/hu.bme.mit.massif.simulink/model/simulink.ecore
```

Figure 66. Connection between Sabotage and Massif at meta-model level

In order to perform code generation 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 composition of 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 logging 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 these are included in the manifest file.



```
matlab.code.generator.xtend
  JRE System Library [JavaSE-1.8]
  Plug-in Dependencies
  src
    matlab.code.generator.xtend
      MatlabcodeGeneratorModule.java
      MatlabcodeGeneratorSetup.java
      MatlabcodeGeneratorSupport.java
      MatlabcodeGenerator.mwe2
      MatlabcodeGenerator.xtend
  xtend-gen
  META-INF
    MANIFEST.MF
  model
    mdl_03_08_2017original_terminator.simulink
  src-gen
    mdl_03_08_2017_faulty.m
  build.properties
  plugin.xml
```

Figure 67. Code Generation workspace

4.2.2.3 Fault Trees generation (*)

The plugins to visualise fault tree are the following:

- eu.fbk.eclipse.standardtools.faultTreeViewer

- eu.fbk.eclipse.standardtools.faultTreeViewer.emfta
- eu.fbk.eclipse.standardtools.faultTreeViewer.emfta.design

These are Eclipse plugins located in the EST git repository[23]. The plugins are derived from the open source tool EMFTA[24], to embed the fault tree viewer inside the CHESS platform. The plugins require the Sirius Eclipse tool[25] to work correctly.

4.2.2.4 Metrics

4.2.2.4.1 Metrics for requirements

4.2.2.4.1.1 Correctness metrics

Following, the functions of the source code are presented. For each metric two function are specified: one about the numerical value and the other giving 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.

[23] https://gitlab.fbk.eu/CPS_Design/EST.git

[24] https://github.com/juli1/emfta

[25] https://eclipse.org/sirius/

Metric to verbs

In-System Conceptual Model Verbs (SCM Verbs)

`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)

`SccVerbCount:` return a double with the result of the metric.

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

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

`OutOfSccVerbCount:` return a double with the result of the metric.

`OutOfSccVerbFeatures:` 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.

4.2.2.4.2 Applying machine learning to improve the quality of requirements

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

4.2.2.4.3 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 the quality of the models. The information extracted from the models is processed by the metrics to evaluate the quality. The following list gives 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`

4.2.2.4.4 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 loads the quality value from the snapshot saved in the project and a graphical area chart is shown.

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

4.2.2.5 Contract-Based Assurance Composition and Model Checking (*)

The plugins that need to be installed on top of the CHESS 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 also enables 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 CHESS 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 plugins:

- org.polarsys.chess.verificationService: it provides different analysis commands invoking the OCRA and nuXmv. They include “check contract refinement”, “contract-based safety analysis”, “check contract implementation” and “model checking”.

- org.polarsys.chess.smvExporter: it translates the UML state machine in a coherent SMV file that is used as input for the V&V tools.

These depend on a set of Eclipse plugins located in the EST git repository[26]. The plugins are:

- eu.fbk.eclipse.standardtools.contractEditor.core: it contains the core functionalities used in the CHESS plugin contractEditor.
- eu.fbk.eclipse.standardtools.constraintEditor.core: it contains the core functionalities used in the CHESS plugin constraintEditor.
- eu.fbk.eclipse.standardtools.propertyEditor.core: it contains the core functionalities used in the CHESS plugin propertyEditor.
- eu.fbk.eclipse.standardtools.hierarchicalContractView.core: it contains the core functionalities used in the CHESS plugin hierarchicalContractView.
- eu.fbk.eclipse.standardtools.refinementView.core: it contains the core functionalities used in the CHESS plugin refinementView.
- eu.fbk.eclipse.standardtools.xtextService: it contains the core functionalities used in the three CHESS plugins editors.
- eu.fbk.eclipse.standardtools.ExecOcraCommands: it contains the core functionalities to interact with OCRA, that are used in the CHESS plugin verificationService.
- eu.fbk.eclipse.standardtools.nuXmvService: it contains the core functionalities to interact with nuXmv, that are used in the CHESS plugin verificationService.
- eu.fbk.eclipse.standardtools.ModelTranslatorToOcra: it contains the core functionalities to translate the CHESS model in a coherent OSS model.
- eu.fbk.eclipse.standardtools.StateMachineTranslatorToSmv: it contains the core functionalities to translate the state machines defined in the CHESS model in coherent SMV models.

There are some Eclipse plugins used by these 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. 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 needed for the editor plugins.

Figure 68 shows in more detail the dependencies among the plugins. The set org.polarsys.chess.* are the plugins described in Section 4.2.1.

[26] https://gitlab.fbk.eu/CPS_Design/EST.git

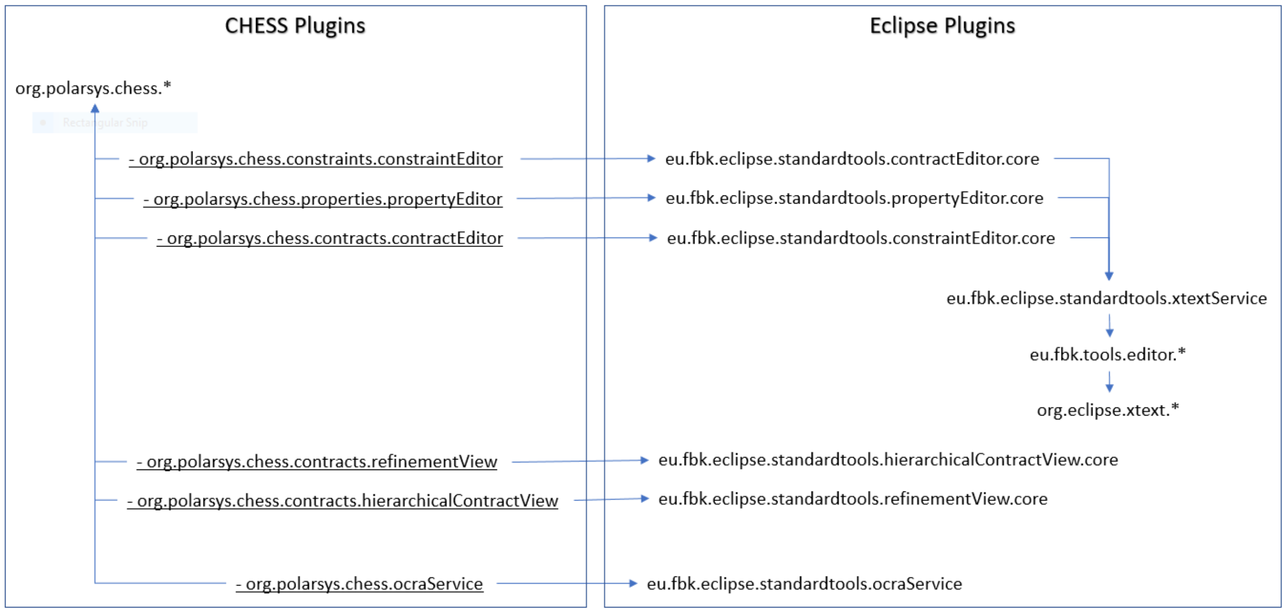

CHESS Plugins
Eclipse Plugins
org.polarsys.chess.*
- org.polarsys.chess.constraints.constraintEditor
- org.polarsys.chess.properties.propertyEditor
- org.polarsys.chess.contracts.contractEditor
- org.polarsys.chess.contracts.refinementView
- org.polarsys.chess.contracts.hierarchicalContractView
- org.polarsys.chess.ocraService
eu.fbk.eclipse.standardtools.contractEditor.core
eu.fbk.eclipse.standardtools.propertyEditor.core
eu.fbk.eclipse.standardtools.constraintEditor.core
eu.fbk.eclipse.standardtools.xtextService
eu.fbk.tools.editor.*
org.eclipse.xtext.*
eu.fbk.eclipse.standardtools.hierarchicalContractView.core
eu.fbk.eclipse.standardtools.refinementView.core
eu.fbk.eclipse.standardtools.ocraService

Figure 68. Diagram showing the dependencies among the plugins. The direction of the arrow means that the origin plugin depends on the target plugin.

4.2.2.6 Report Generation (*)

The plugin developed to generate the report describing the system architecture is:

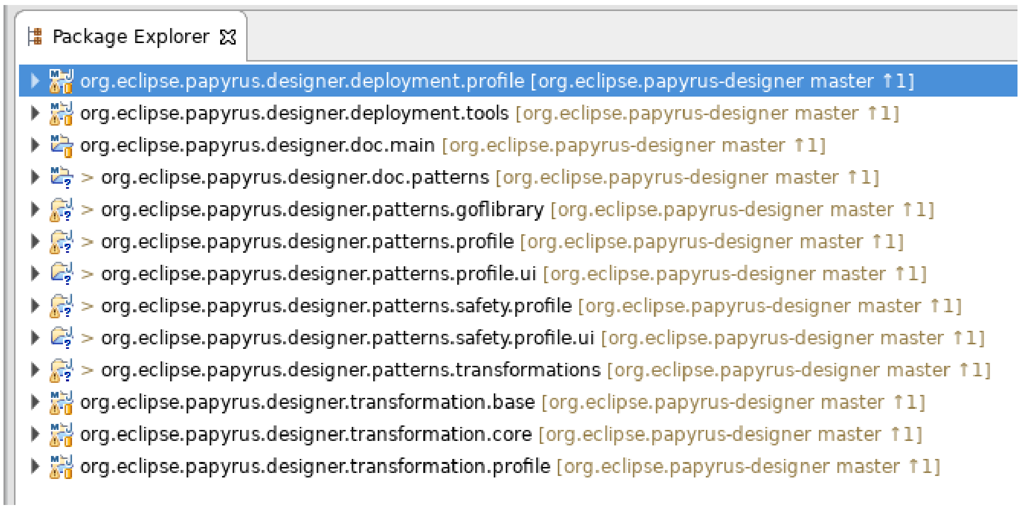
- org.polarsys.chess.diagram.ui: it enables the generation of a document in doc or tex format containing information about system components. Moreover, it provides functionalities to export UML/SysML diagrams as images. Those functionalities are used to generate the report.

4.2.2.7 Architectural Patterns (*)

The implementation related to architectural pattern definition and instantiation support is currently ongoing in AMASS.

A set of Papyrus plugins concerning architectural patterns support has been released by CEA in the context of the Papyrus Software Designer release[27]. The plugins can be reached on the following gerrit patch commit: https://git.eclipse.org/r/#/c/126526/. To be able to run them, additional plugins from Papyrus designer are required to satisfy dependencies as shown on Figure 69.

27 https://wiki.eclipse.org/Papyrus_Software_Designer


Package Explorer

```
org.eclipse.papyrus.designer.deployment.profile [org.eclipse.papyrus-designer master ↑1]
org.eclipse.papyrus.designer.deployment.tools [org.eclipse.papyrus-designer master ↑1]
org.eclipse.papyrus.designer.doc.main [org.eclipse.papyrus-designer master ↑1]
> org.eclipse.papyrus.designer.doc.patterns [org.eclipse.papyrus-designer master ↑1]
> org.eclipse.papyrus.designer.patterns.goflibrary [org.eclipse.papyrus-designer master ↑1]
> org.eclipse.papyrus.designer.patterns.profile [org.eclipse.papyrus-designer master ↑1]
> org.eclipse.papyrus.designer.patterns.profile.ui [org.eclipse.papyrus-designer master ↑1]
> org.eclipse.papyrus.designer.patterns.safety.profile [org.eclipse.papyrus-designer master ↑1]
> org.eclipse.papyrus.designer.patterns.safety.profile.ui [org.eclipse.papyrus-designer master ↑1]
> org.eclipse.papyrus.designer.patterns.transformations [org.eclipse.papyrus-designer master ↑1]
org.eclipse.papyrus.designer.transformation.base [org.eclipse.papyrus-designer master ↑1]
org.eclipse.papyrus.designer.transformation.core [org.eclipse.papyrus-designer master ↑1]
org.eclipse.papyrus.designer.transformation.profile [org.eclipse.papyrus-designer master ↑1]
```

Figure 69. Papyrus plugins for architectural pattern definition and manipulation support

The release enriches the Papyrus editor with a specific UML profile for pattern definition and provides support for pattern instantiation. It also contains associated Papyrus customization, as well as a subset of GoF patterns. There is support to apply / integrate a design pattern in application models. There is a small extension profile for safety patterns concerning a set of non-functional properties related to availability, reliability and resource consumption. Related to pattern instantiation, the tool provides support for managing the binding between the pattern roles, i.e. the abstract entities foreseen by the pattern definition, and the actual entities available in the given system model where the pattern is instantiated. The plugins include a minimal documentation as user manual.

The provided set of plugins is used as a baseline in AMASS; the original release targets the Eclipse Oxygen/Photon environment. A first experimentation has shown the usability of the plugins in Neon environment with degraded customization capabilities: basically, dedicated palette and stylesheets attached to the diagrams cannot be used. Although the customization features are not required for pattern definition and instantiation, the plugins have to be adapted to the Eclipse Neon AMASS environment to support full capability offers by the released plugins. Moreover, the integration with CHESS has to be realized, in particular to allow the modelling of contracts for patterns, enabling the application of the argumentation fragment for architectural patterns described in AMASS D3.3 [16].

5. Conclusions(*)

This deliverable D3.6 “Prototype for Architecture-Driven Assurance (c)” is the third and last 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. Innovative approaches have been described within this document and were incorporated into the AMASS Tool Platform to reach this objective. CHESS and SAVONA allow the creation of system models based on the Papyrus Framework. Furthermore, system requirements and contracts can be defined, validated and verified using OCRA and the V&V Manager. Different metrics can be used via the RQA tool to analyse the modelled system and increase its quality. To support early safety assessments during development, the SABOTAGE tool offers simulation-based fault injection on the created system model. All of the assurance and model artefacts can thereby be traced using CAPRA.

With three planned prototype iterations for the framework, this deliverable reports the status for the third and final prototype release (Prototype P2), 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.

Following the two previous versions of this deliverable, D3.6 strongly focused on the integration of different approaches and ideas into one unified AMASS tooling framework supporting architecture-driven assurance. Among others, the AMASS Platform tools for the final Prototype P2, the finalized User Manuals and installation Instructions and the source code description for Prototype P2 were presented in detail in the document.

Based on the data provided for the deliverable, all partners showed significant effort with the implementation of their individual features and functionalities, which can also be seen by looking at the requirements table in section 2.2. Most of the requirements regarding architecture based assurance have already been covered when this document is published. Nevertheless, the remaining ones are still being worked on and will be finished by the release of the final AMASS Prototype P2.

Abbreviations

Abbreviation	*Explanation*
AADL	Architecture Analysis and Design Language
API	Application Programming Interface
AUTOSAR	AUTomotive Open System ARchitecture
BNF	Backus-Naur Form
BSCU	Braking System Control Unit
CACM	Common Assurance and Certification Meta-model
CHESSML	CHESS Modelling Language
CPS	Cyber Physical System
DC	Direct Current
ECSEL	Electronic Components and Systems for European Leadership
ELK	Eclipse Layout Kernel
EMF	Eclipse Modelling Framework
EMFTA	EMF-based Fault-Tree Analysis Tool
FMEA	Failure Mode and Effects Analysis
FTA	Fault Tree Analysis
GUI	Graphical User Interface
HARA	Hazard Analysis and Risk Assessment
HTTP	Hypertext Transfer Protocol
IBD	Internal Block Diagram
IDE	Integrated Development Environment
IMA	Integrated Modular Avionics
JU	Joint Undertaking
LTL	Linear Temporal Logic
NLP	Natural Language Processing
MARTE	Modelling and Analysis of Real Time and Embedded systems
MTL	Metric Temporal Logic
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	Symbolic Model Verifier
SW	Software
SysML	System Modelling Language
TARA	Threat Analysis and Risk Assessment
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), 29 September 2017

[7] AMASS D2.4 AMASS Reference Architecture (c), 4 June 2018

[8] M. dos Santos Soares, J. Vrancken: "Model-Driven User Requirements Specification using SysML", JOURNAL OF SOFTWARE, VOL. 3, No. 6, June 2008

[9] XML Metadata Interchange, www.omg.org/spec/XMI/

[10] AMASS D3.1 Baseline and requirements for architecture-driven assurance, 30 September 2016

[11] 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

[12] Eclipse Layout Kernel, https://www.eclipse.org/elk/

[13] Acacia+, http://lit2.ulb.ac.be/acaciaplus/

[14] Xtend, https://eclipse.org/xtend/documentation/2.7.0/Xtend%20User%20Guide.pdf

[15] Massif Ecore description, https://github.com/viatra/massif/tree/master/plugins/hu.bme.mit.massif.simulink/model

[16] AMASS D3.3 Design of the AMASS tools and methods for architecture-driven assurance (b), 30 March 2018

[17] Massif: MATLAB Simulink Integration Framework for Eclipse, https://github.com/viatra/massif

[18] AMASS D2.1 Business cases and high-level requirements, 28 February 2017

[19] AMASS D2.2 AMASS Reference Architecture (a), 30 November 2016

[20] AMASS D3.4 Prototype for Architecture Driven Assurance (a), 23 December 2016

[21] AMASS D3.5 Prototype for Architecture-Driven Assurance (b), 29 September 2017

[22] Verification Studio (V & V Studio) previously called RQA. The Reuse Company. Accessed August 22th, 2018. https://www.reusecompany.com/verification-studio

[23] AMASS D3.8 Methodological Guide for Architecture-Driven Assurance (b), 31 October 2018

[24] AMASS D4.3 Design of the AMASS tools and methods for multiconcern assurance (b), 30 April 2018

[25] AMASS D4.6 Prototype for multi-concern assurance (c), 31 August 2018

[26] AMASS D5.6 Prototype for seamless interoperability (c), 30 September 2018

[27] AMASS D7.3 open source platform project proposal, 31 January 2018

[28] AMASS D7.5 open source platform provisioning and website (a), 31 July 2017

[29] AMASS D7.6 open source platform provisioning and website (b), 31 March 2018