CHESS 1.1.0

Contract-based Analysis, Model Checking, and Safety Analysis

FONDAZIONE
BRUNO KESSLER

Contents

1. Introduction

This document guides the user through the usage of CHESS for the design and analysis of a system architecture based con contracts specification, model checking, and safety analysis. These activities are supported in CHESS with the interaction of backend V&V tools, namely, OCRA, nuXmv, and xSAP.

2. Setup of External V&V Tools

In the subfolder of the bundle **\V&VTools**, there are builds of the V&V tools.

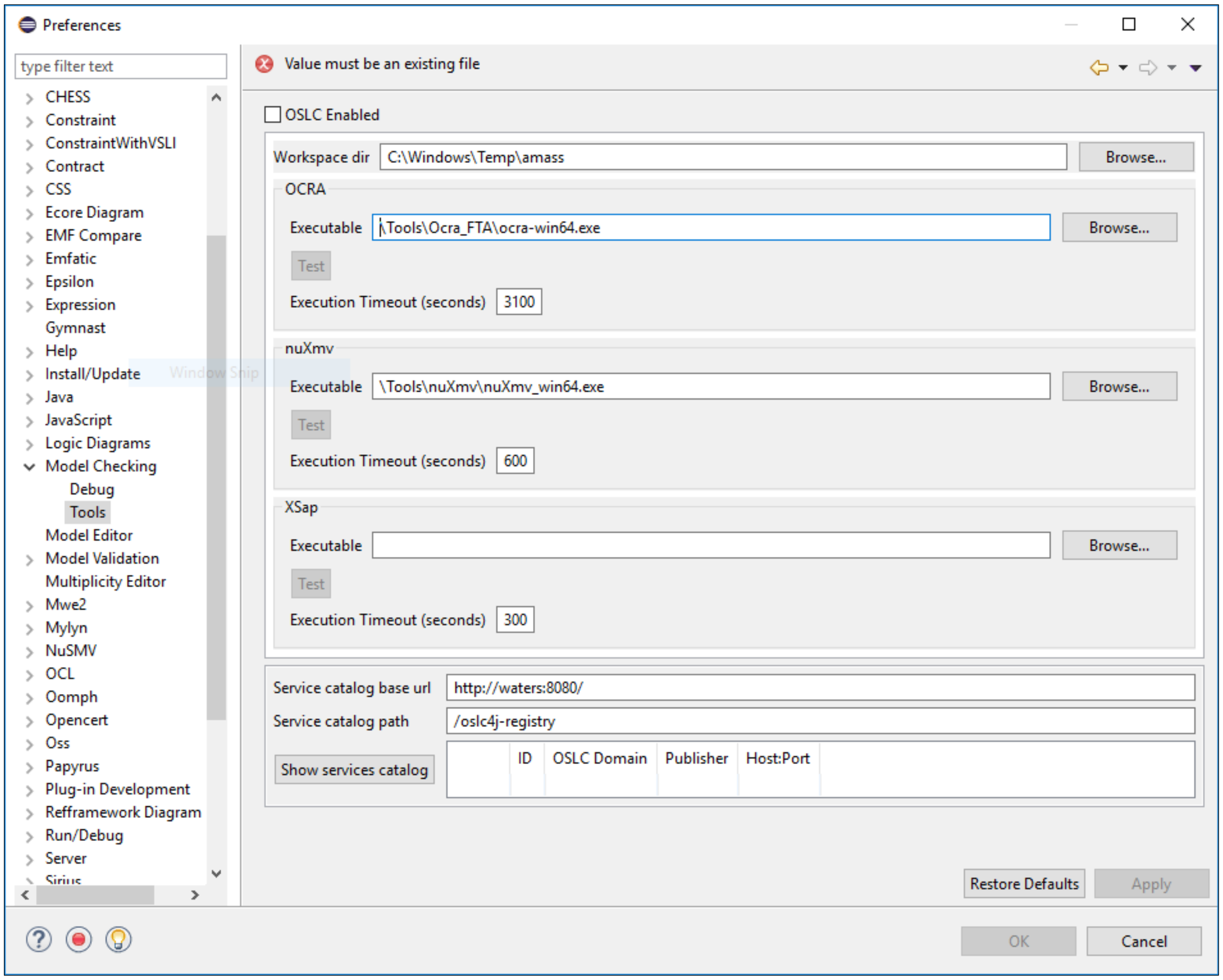
To enable the tools that are available locally:

- Go to "**Window → Preferences → Model Checking → Tools**", see Figure 1.
- For each V&V tool (OCRA, nuXmv, xSAP)
    - Click on "Browse…" and set the path of the executable

      e.g. CHESS_directory\V&VTools\OCRA\Ocra-win64.exe
- Click on "Test", to verify that the tool is compatible with CHESS.

To get access to V&V Tools that are exposed as web services via OSLC:

- Click on the check box "OSLC Enabled".
- Set the URL of the service catalogue.
- Set the path of the service catalogue.
- Click on "Show services catalogue" to verify that the web service exposes the V&V tools.



Value must be an existing file

OSLC Enabled

Workspace dir C:\Windows\Temp\amass

OCRA

Executable \Tools\Ocra_FTA\ocra-win64.exe

Execution Timeout (seconds) 3100

nuXmv

Executable \Tools\nuXmv\nuXmv_win64.exe

Execution Timeout (seconds) 600

XSap

Executable

Execution Timeout (seconds) 300

Service catalog base url http://waters:8080/

Service catalog path /oslc4j-registry

ID	OSLC Domain	Publisher	Host:Port

Figure 1. The preferences window to configure the V&V tools

3. Create a CHESS Project, Model and Diagrams

A CHESS project (i.e. the folder where the model and other artefact are stored) and a model can be created by using the CHESS dedicated Eclipse wizard (“**File → New → Others → CHESS…**”).

Please note that currently CHESS projects/models cannot be created in a CDO checkout but only as folder/file in the current workspace.

A CHESS project is basically a Papyrus project customized with the CHESS nature **(Make sure you use the Papyrus perspective in Eclipse while working with CHESS projects).**

It is possible to change the style of the CHESS diagrams going to the main menu “**Window → Preferences -> Papyrus → CSS Theme**”, and then selecting the current theme. It is recommended to use the “CHESS theme”.

A CHESS model is a Papyrus UML+SysML+MARTE model coming with a predefined set of packages/views and with the CHESS profile automatically applied on it, to allow modelling of contracts, dependability and real-time concerns. For details on the CHESS model see the “CHESS Modelling Language” guide.

A CHESS profile and tool can also be used to exploit different analysis at system level, and to apply the CHESS model driven methodology for the design, analysis and implementation of critical SW systems. More information about the CHESS methodology can be found at the CHESS web site[1]. See the “CHESS Toolset User Guide” as a general toolset use guide.

The CHESS editor is a customization of the Papyrus editor, offering a set of features on top of the CHESS model. To have an overall understanding of the Papyrus environment, for instance regarding the different views, commands, diagrams, please check the Papyrus user manual available online[2].

Below are some examples of SysML Block Definition Diagram (BDD in Figure 2) and Internal Block Diagram (IBD in Figure 3) which can be used to model the system hierarchical architecture, i.e. blocks, ports and connections. **Appendix A. CHESS Supported Basic Types** itemizes the supported basic types.

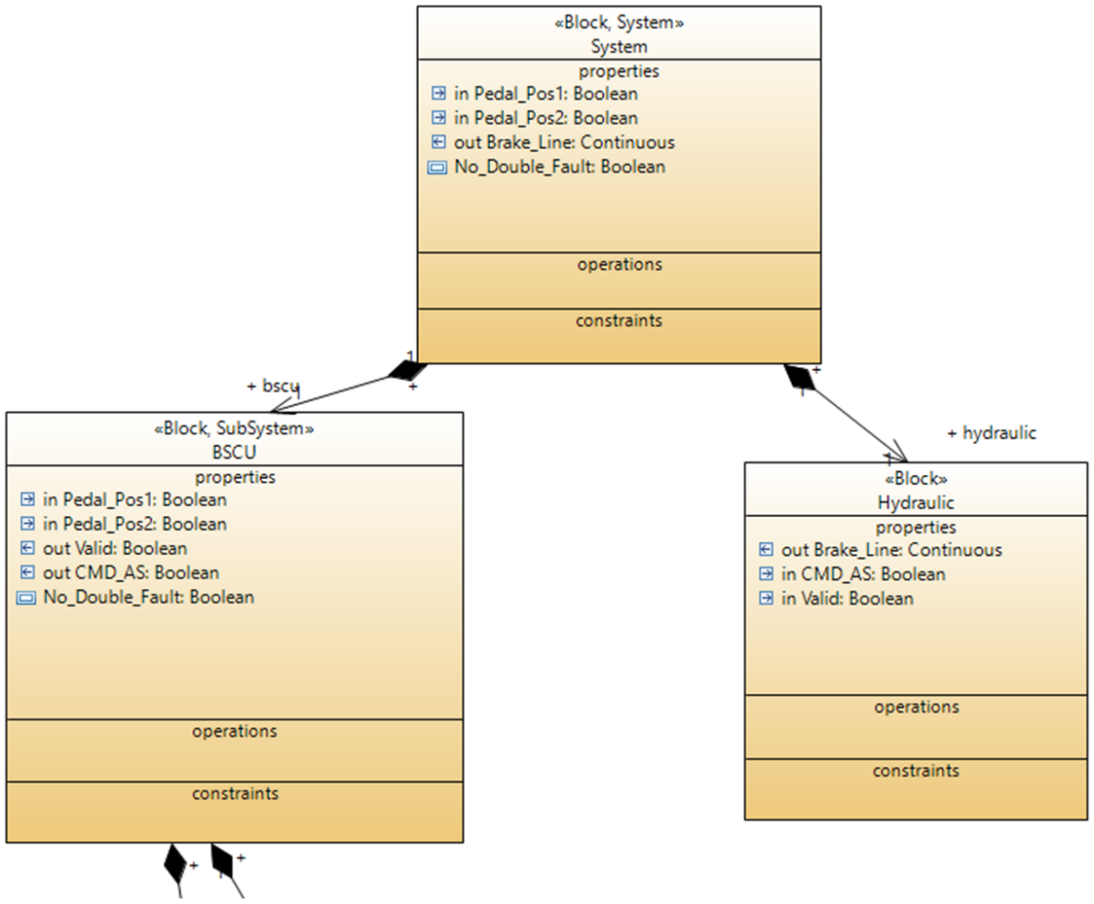


Figure 2. Block Definition Diagram example

[1] https://www.polarsys.org/chess/

[2] https://wiki.eclipse.org/Papyrus_User_Guide

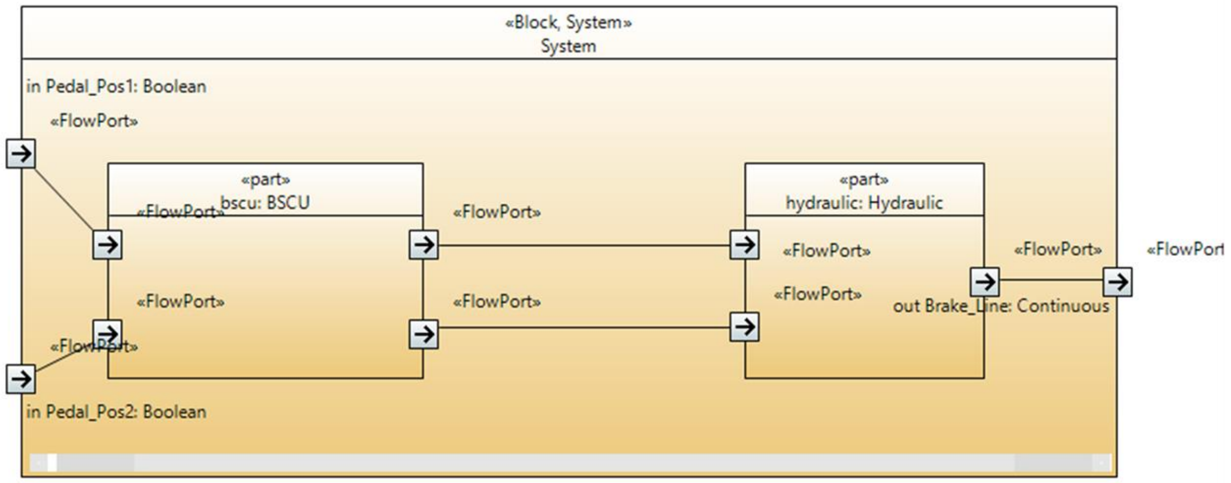


Figure 3. Internal Block Diagram example

3.1 Import CHESS project from a Git repository

To import an existing CHESS project from a Git repository:

- Select the “Git Perspective”
- In the Git Repositories View, chick on “Add an existing Git repository”
- Select the imported repository, right click and “import projects”.

Note: Be sure to have installed eGit[3]*.*

4. Create Requirements

System requirements can be created by using the SysML Requirement diagram. On the “Model Explorer” view, right click on “**CHESS RequirementView package**” and create a SysML Requirement diagram. Use the palette to create Requirement entities.

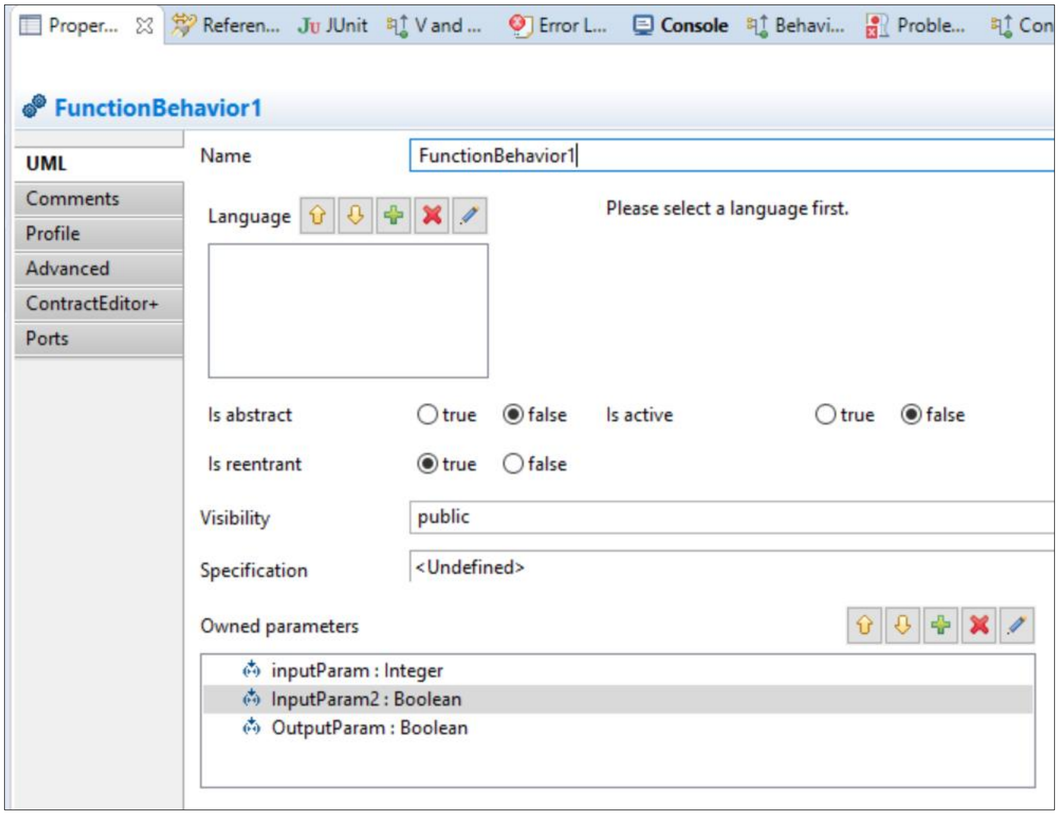
Papyrus also supports the import of requirements from different sources, e.g. Excel files or ReqIF model.

5. Create a FunctionBehavior

FunctionBehavior represents an uninterpreted function that can be used in formal properties, contracts, and state machines. To create a FunctionBehavior element:

- In the Model Explorer View, select the owner of the FunctionBehavior.
- Right click “**New Child → Function Behavior → as OwnedBehavior**”
- Select the created element and in the Property View - “UML” tab, set the name of the FunctionBehavior.
- In the “UML” tab, go to the owned parameters area and click on add elements to create the input/output parameters of the function (see Figure 4).

[3] http://download.eclipse.org/egit/updates


Proper... Referen... JUnit V and ... Error L... Console Behavi... Proble... Con

FunctionBehavior1

UML | Comments | Profile | Advanced | ContractEditor+ | Ports

Name: FunctionBehavior1

Language

Please select a language first.

Is abstract: true [✓ false] Is active: true [✓ false]

Is reentrant: [✓ true] false

Visibility: public

Specification: <Undefined>

Owned parameters
- inputParam : Integer
- InputParam2 : Boolean
- OutputParam : Boolean

Figure 4. "UML" tab of the selected FunctionBehavior element.

6. Create Formal Properties

Formal properties (i.e. the entities which play the role of Assumption and Guarantee in a Contract) can be created manually by using the dedicated tool in the "Contracts" palette.

It is also possible to package a FormalProperty in a Contract by creating the former directly in the latter.

To create a FormalProperty in a Block Definition Diagram (BDD)/Component diagram:

- Select FormalProperty from the Contract palette and click on the BDD
    - Give a proper name to the FormalProperty

To create a FormalProperty associated to a specific component (i.e. the owner of the FormalProperty), in a Block Definition Diagram (BDD)/Component diagram:

- Select FormalProperty from the Contract palette and click on the specific component in the BDD
    - Give a proper name to the FormalProperty

To create a FormalProperty packaged in a Contract:

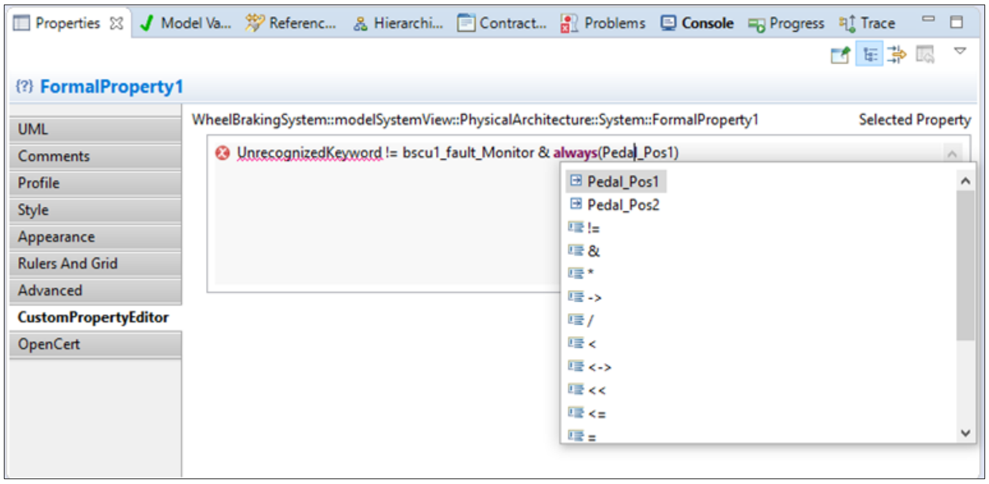
- Select FormalProperty from the Contract palette and click on the Contract
    - Give a proper name to the FormalProperty

If the FormalProperty is not used to define contracts, but to describe for example a possible scenario, it can refer to ports and parameters of sub-components of sub-components (e.g. the FormalProperty can refer to the port "subcomp.block.subBlock.port"). To enable this aspect, select the FormalProperty, then in the Property View – Advanced – Visibility - select "Private".

7. Edit a Formal Property

To edit a FormalProperty:

- Select the FormalProperty from the Project Explorer View or select its graphical element from the BDD.
- Open the tab "PropertyEditor+" in the Property View and write the formal property in OCRA Language **(Appendix B. OCRA Language to define Formal Properties and Contracts**). Pressing CRTL + SPACE it is possible to have a list of supported keywords of the OCRA Language to define the formal property. If the owner component is set correctly, the content assist can suggest also the input/output ports and the attributes of the owner component to type. Moreover, it notifies whether a typed word does not belong to a keyword/port/attribute, see Figure 5.



WheelBrakingSystem::modelSystemView::PhysicalArchitecture::System::FormalProperty1

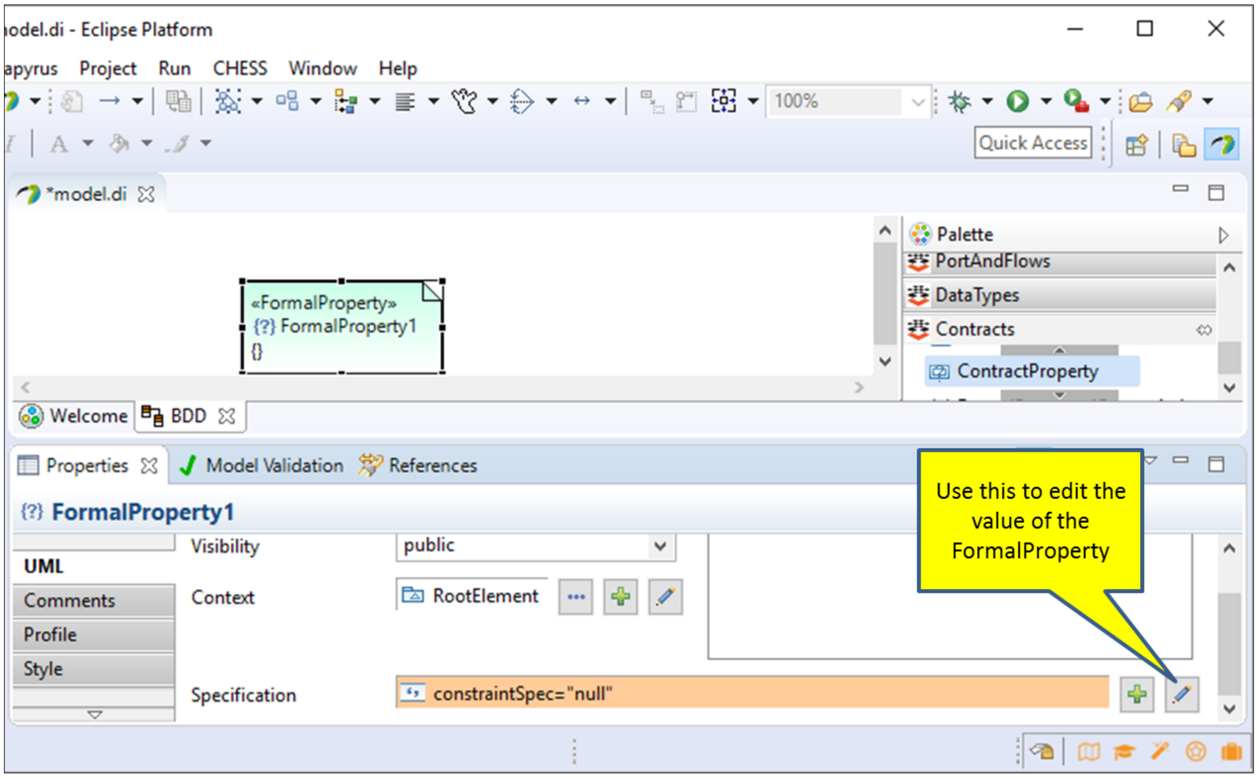
UnrecognizedKeyword != bscu1_fault_Monitor & always(Pedal_Pos1)

Pedal_Pos1
Pedal_Pos2
!=
&
*
->
/
<
<->
<<
<=
=

Figure 5. Property Editor with content assist

To associate an expression to the FormalProperty as shown in Figure 6:

- Select the FormalProperty.
- In the "UML" tab of the Properties view, edit the current value of the "Specification" field and provide the expression in the "Value" field.



«FormalProperty»
FormalProperty1

FormalProperty1

Visibility: public
Context: RootElement
Specification: constraintSpec="null"

Use this to edit the value of the FormalProperty

Figure 6. Creating a Formal Property

Use the formalize property of the FormalProperty stereotype, available in the “Profile” tab of the Property View, to link the requirements formalized by the current FormalProperty (see Figure 7).

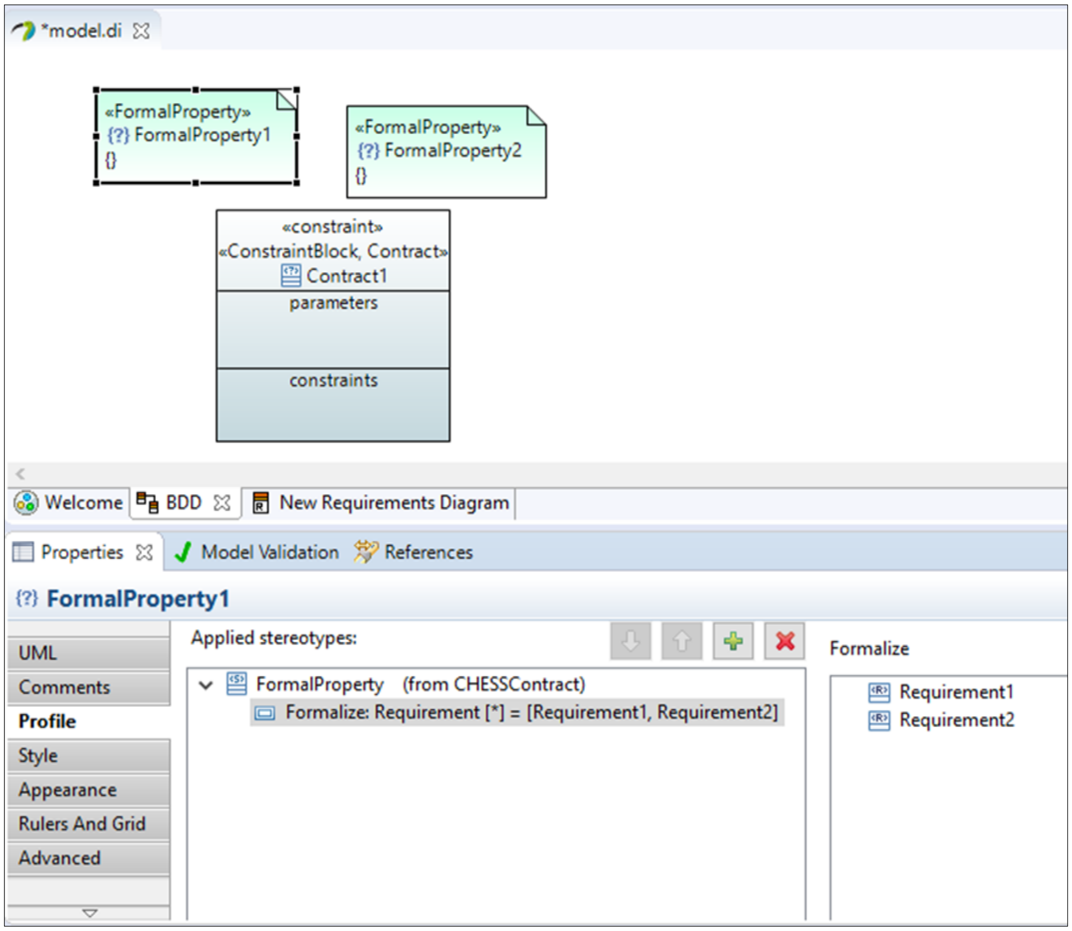


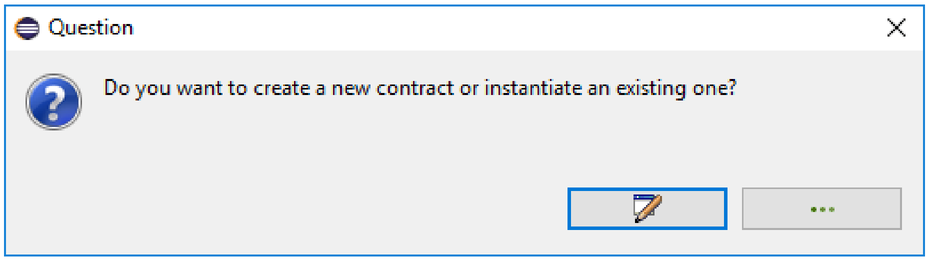
Figure 7. Formalizing Requirements

8. Create a Contract

Contracts can be created in a BDD and in a Class/Component Diagram.

To create a Contract:

- Open a BDD/Class diagram.
- Select “Contract” from the Contracts palette and click on the diagram.
  - A popup appears to choose if a new contract must be created or if an existing one must be instantiated (see Figure 8).
  - Give a proper name to the Contract.

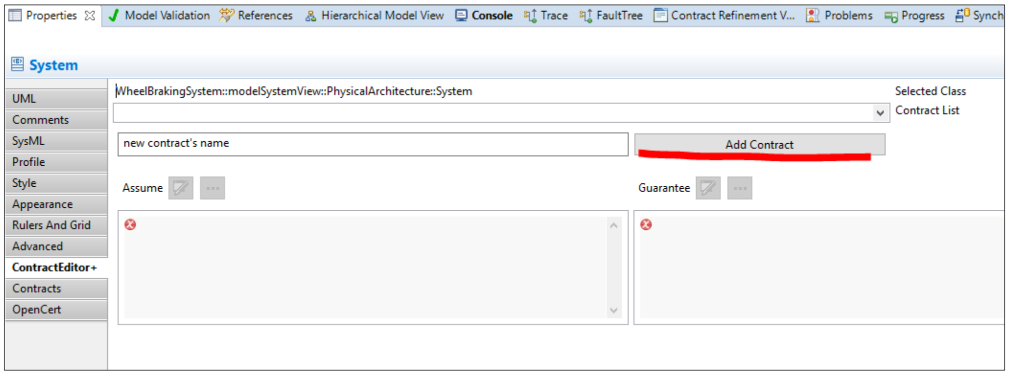


Do you want to create a new contract or instantiate an existing one?

Figure 8. Popup to create a new contract or instantiate an existing one

Alternatively, it is possible to create a Contract without the need to create its graphical representation from a BDD:

- Select the component to assign the contract. The element can be selected in the Model Explorer View or in the graphical editor.
- Open the "ContractEditor+" tab in the Property View, type the name of the contract in the "new contract's name" text field and click the "Add Contract" button (see Figure 9). A popup appears to choose if a new contract must be created or if an existing one must be instantiated.



WheelBrakingSystem::modelSystemView::PhysicalArchitecture::System

new contract's name

Figure 9. The Add Contract button in the ContractEditor

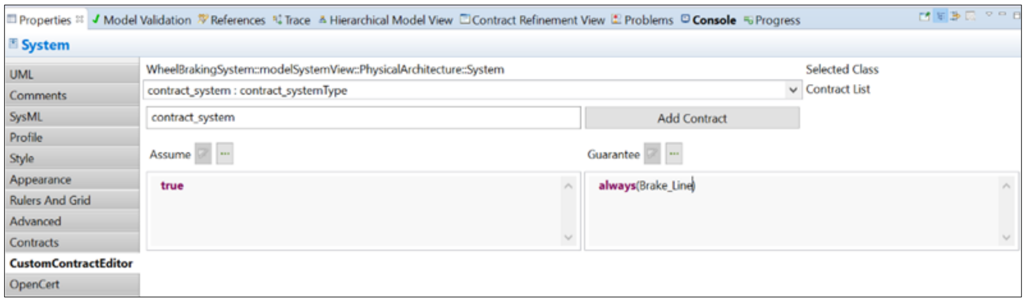
9. Specify Assumption and Guarantee for a Contract

To specify an Assume (or Guarantee) FormalProperty for a given Contract:

- Select the Contract, open the "Profile" tab in the Properties View and set the Assume (or Guarantee) attribute of the Contract stereotype to the (previously created) FormalProperty.

The Assumption and the Guarantee properties for a given Contract can also be specified through the dedicated "ContractEditor+" tab in the property editor (see Figure 10). To be able to use the "ContractEditor+" tab:

- Select the Contract to edit, or
- Select the Block/Component and then select the ContractProperty (see below) from the ContractList in the "ContractEditor+" tab.



WheelBrakingSystem::modelSystemView::PhysicalArchitecture::System

contract_system : contract_systemType

contract_system

true

always(Brake_Line)

Figure 10. Editing the Contract's Assume and Guarantee

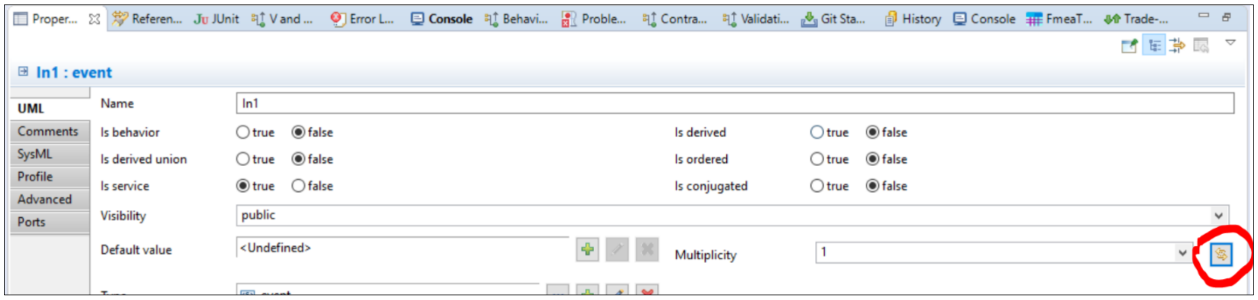
10. Parameterized Architectures

10.1 Set the multiplicity of the elements

It is possible to set the multiplicity of FlowPorts and sub-components to express a list of elements with the same type.

To set the multiplicity of the selected element:

- In the Property View – "UML" tab, in the multiplicity area, click on "Switch editors", see Figure 11.
- In the LowerValue Specification and in the UpperValue Specification, click on "Create new object" – "LiteralString", see Figure 12.
- In the "Value" text area, write the expression that expresses the number of elements (e.g. "5" or "numSubComp + 4" where numSubComp is a declared flowport).



In1 : event

Name: In1
Is behavior: true [✓ false]
Is derived union: true [✓ false]
Is service: [✓ true] false
Is derived: true [✓ false]
Is ordered: true [✓ false]
Is conjugated: true [✓ false]
Visibility: public
Default value: <Undefined>
Multiplicity: 1

Figure 11. Switch editors to set the multiplicity

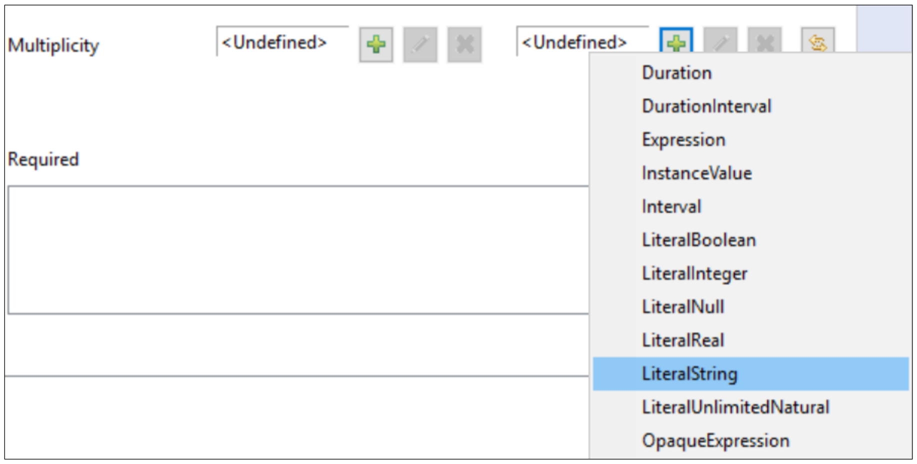


Figure 12. Select LiteralString for both LowerValue and UpperValue Specification

10.2 Modeling parameterized architecture

To parameterize an existing architecture, perform the following steps:

- Create the parameter: In the Model Explorer View or in the BDD Editor, create a static FlowPort. The static attribute can be set to "true" in the "SysML" tab of the Properties view.

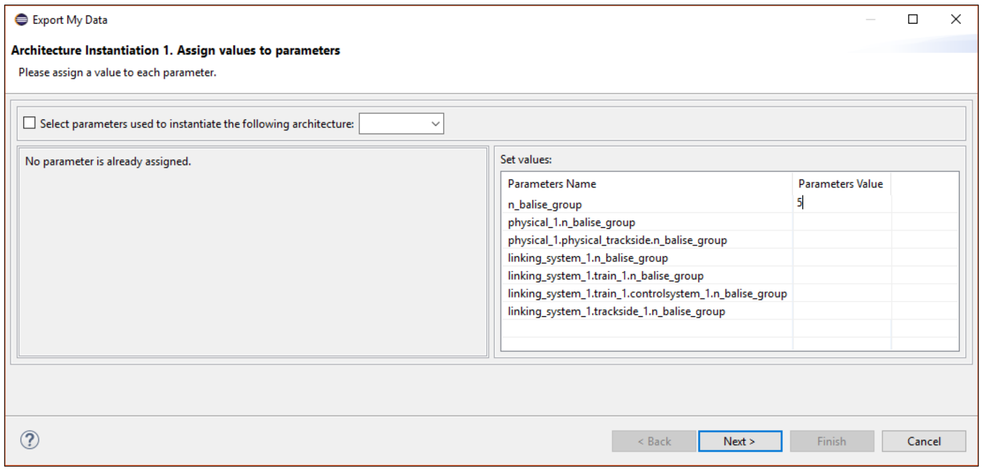
- Give an assignment to the parameter (optional step). In the Model Explorer View or in the IBD Editor, select the owner of the parameter, create the "DelegationConstraint" element and set a value or an expression to the parameter.
- Give a constraint to the parameter (optional step). In the Model Explorer View, select the owner of the parameter, create the "Constraint" element and write a boolean expression in the OSS language.

10.3 Instantiating the parameterized architecture

<u>Note</u>: for this functionality OCRA must be available, go to Section 1 to setup OCRA.

To instantiate a parameterized architecture, perform the following steps:

- Select the root component of the parameterized architecture: Select the root component (in the "Model Explorer" view) or the corresponding graphical representation (in the diagram editor) and open the menu using the right-button of the mouse. Select "**Instantiate the parameterized architecture**", then a wizard appears.
- Select the parameters used in existing instantiated architecture (optional step).
- Assign a value to each parameter, see Figure 13. This step may require more iteration; a parameter may depend on another parameter, so the latter needs to be set first.
- Import the instantiated architecture into the project, see Figure 14. Select the destination package on the right side of the wizard page.


Export My Data

Architecture Instantiation 1. Assign values to parameters

Please assign a value to each parameter.

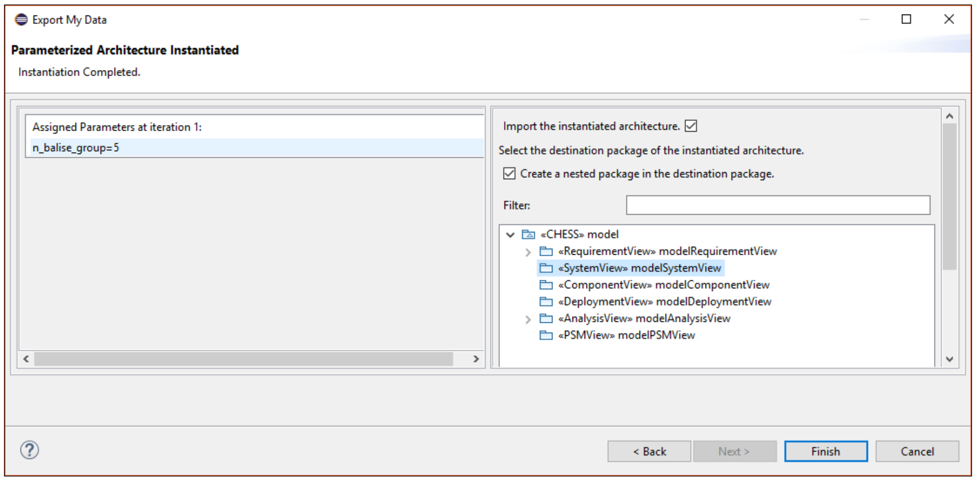
☐ Select parameters used to instantiate the following architecture:

No parameter is already assigned.

Set values:

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

Figure 13. Wizard to set the parameters of the parameterized architecture


Export My Data

Parameterized Architecture Instantiated

Instantiation Completed.

Assigned Parameters at iteration 1:
n_balise_group=5

Import the instantiated architecture. ☑

Select the destination package of the instantiated architecture.

☑ Create a nested package in the destination package.

Filter:

«CHESS» model
«RequirementView» modelRequirementView
«SystemView» modelSystemView
«ComponentView» modelComponentView
«DeploymentView» modelDeploymentView
«AnalysisView» modelAnalysisView
«PSMView» modelPSMView

Figure 14. Last page of the wizard to import the instantiated architecture into the current project

11. Perform Trade-off Analysis

After having instantiated some architectures as explained in Section 9, it will be possible to compute the Trade-off Analysis on the instantiated configurations.

This analysis allows to execute a certain check on all the selected configurations and get the results in a view that simplifies the comparison between them.

To run the Trade-off Analysis, perform the following steps:

- Select the root component of the parameterized architecture (in the Model Explorer View) or the corresponding graphical representation (in the diagram editor).
- Right click and go to "**CHESS → Trade-off Analysis**" (see Figure 15).
- A popup will appear, allowing to select the check to run on the selected configurations (see Figure 16). At least two configurations should be selected. At the moment, only the Check Contract Refinement check can be applied to the configurations.
- A popup will appear, allowing to select the time model of the architecture.

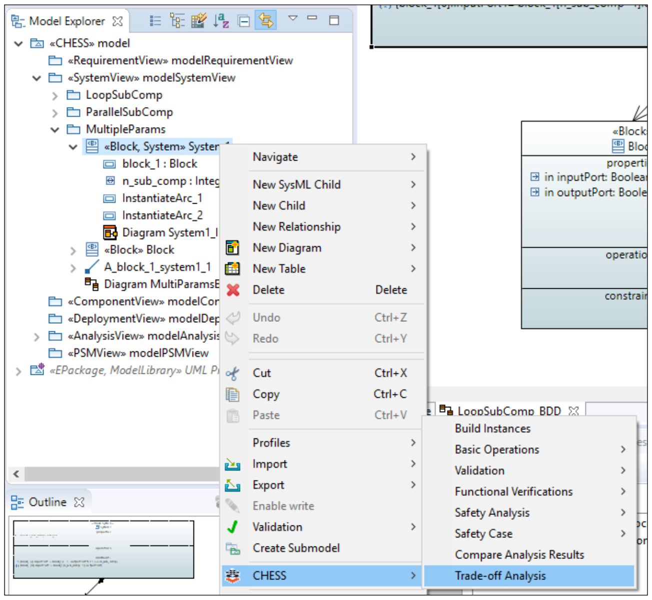
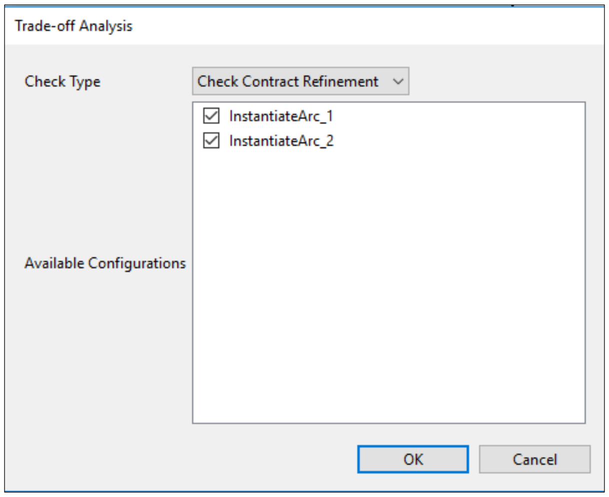


Figure 15. Trade-off Analysis command


Trade-off Analysis

Check Type: Check Contract Refinement

Available Configurations:
- [✓] InstantiateArc_1
- [✓] InstantiateArc_2

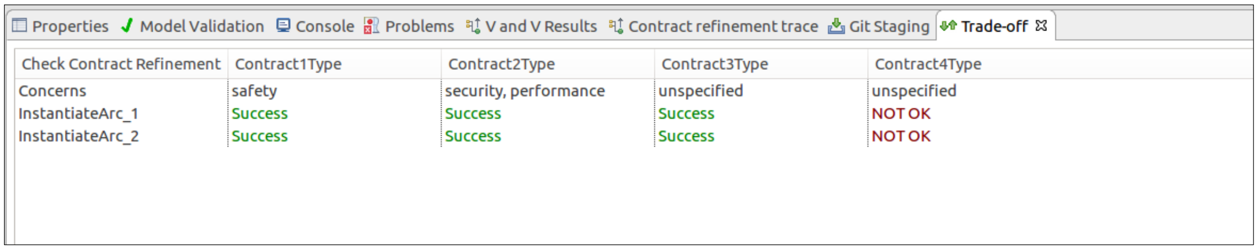
Figure 16. Parameters of the Trade-off Analysis

The results of the analysis will be displayed in a special view called “Trade-off”, as seen in Figure 17.

The type of executed check is displayed in the upper-left cell of the table. On the other columns the contracts found in the analyzed configurations are reported.

The first row of the table reports the concerns specified on the assumption/guarantee formal properties of each contract. In case of different concerns, they will be both reported.

Each following row of the table reports the results of the check on that specific configuration, making it easy to visually compare them. To get a detailed report of the check, double-click on a line and the view will be switched to the Contract Refinement trace for that configuration.



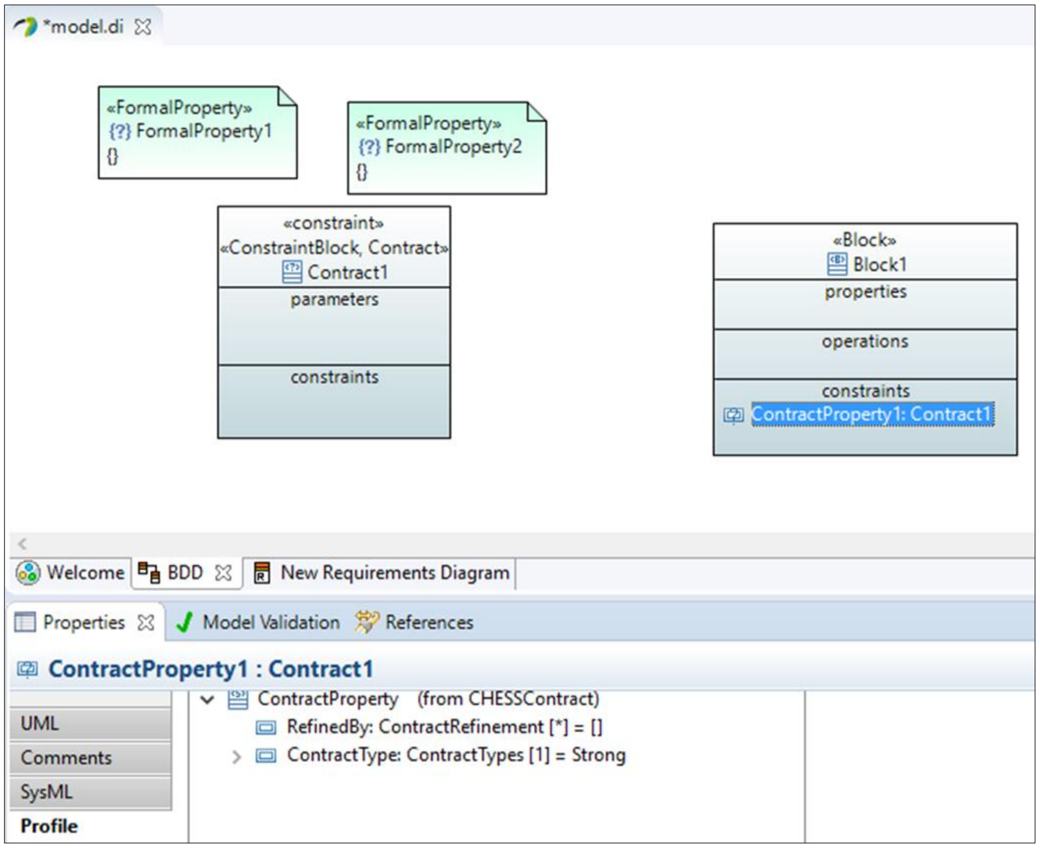
Check Contract Refinement	Contract1Type	Contract2Type	Contract3Type	Contract4Type
Concerns	safety	security, performance	unspecified	unspecified
InstantiateArc_1	Success	Success	Success	NOT OK
InstantiateArc_2	Success	Success	Success	NOT OK

Figure 17. Trade-off Analysis results

12. Associate a Contract to a Block/Component

To associate a Contract to a Block/Component the following actions need to be performed:

- Create a ContractProperty inside the Block/Component (see Figure 18). The ContractProperty acts as a special attribute of the Block/Property.
- Type the just created ContractProperty with the Contract by using the “UML” tab in the Properties View.



ContractProperty (from CHESSContract)
RefinedBy: ContractRefinement [*] = []
ContractType: ContractTypes [1] = Strong

Figure 18. ContractProperty

12.1 Contract Refinement

Once a model component that has a contract has been decomposed, it is possible to define the contract’s refinement. The refinement of a contract can be specified following these steps (see Figure 19):

- Select a ContractProperty of a Block in a BDD or in the Papyrus ModelExplorer View.

- Right click and select "**Contract → Set Contract Refinement**".

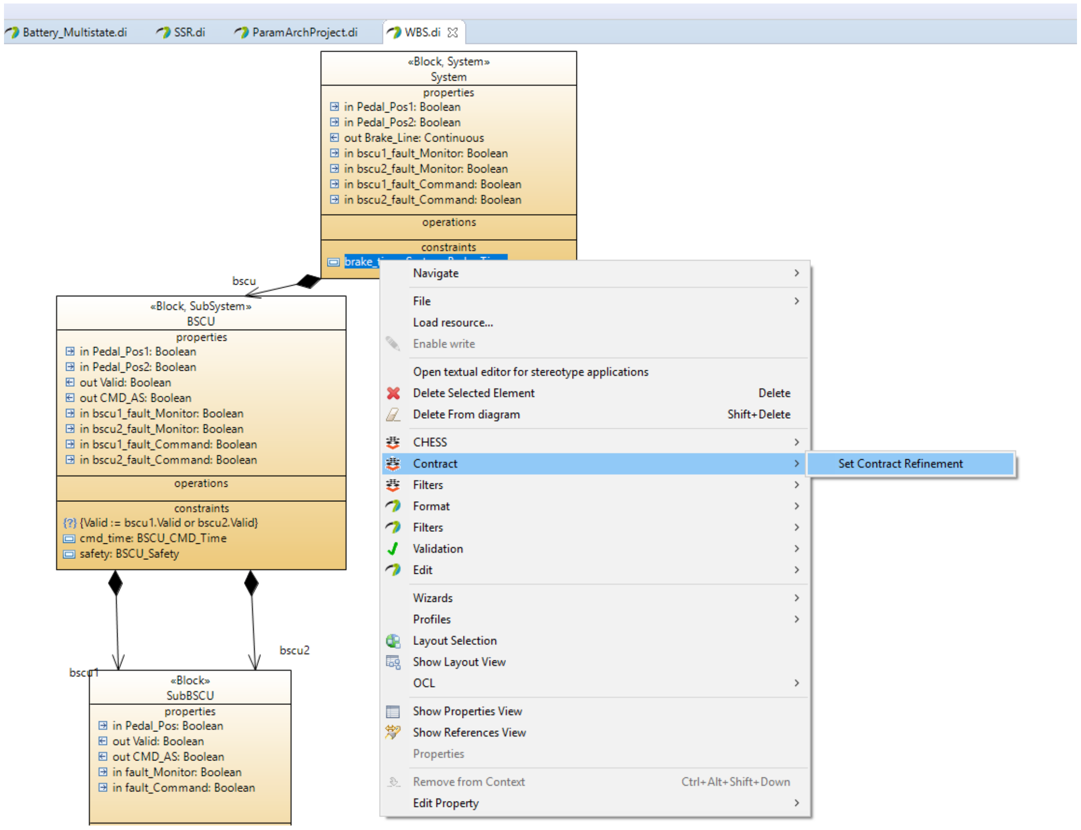
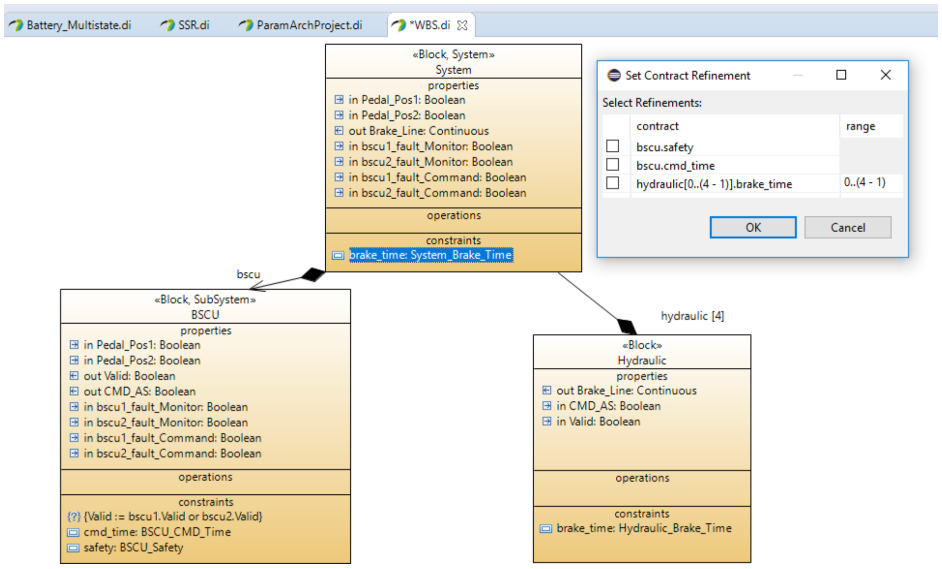


Figure 19. Set Contract Refinement Command

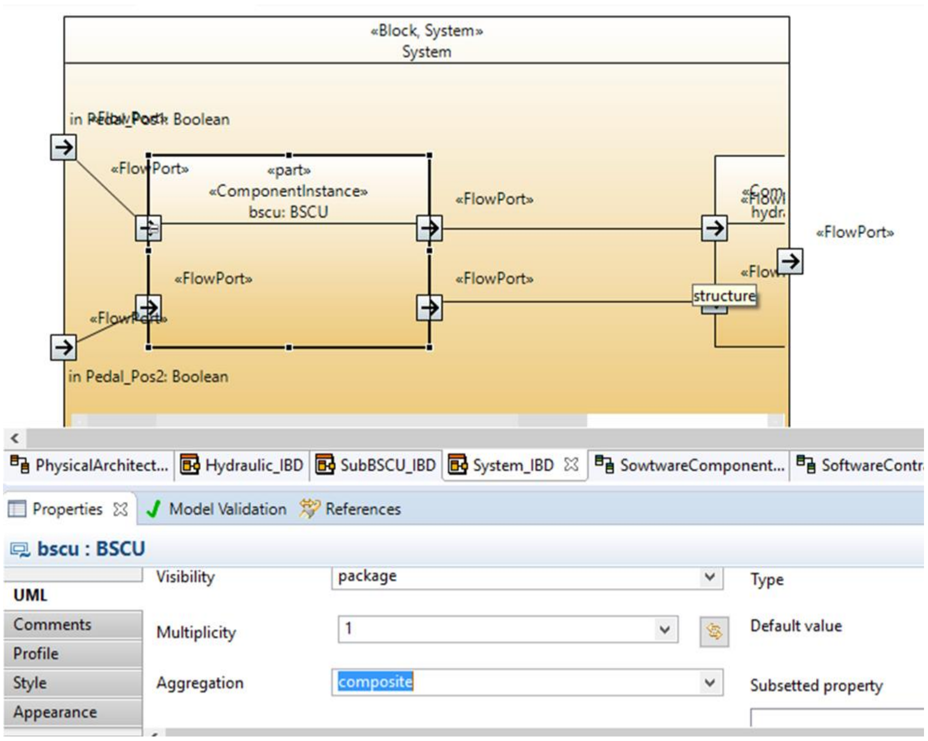
- In the new dialog window select the Contract Properties from the list (see Figure 20). In case the multiplicity of the *Aggregation* is defined, the "Range" field allows to specify which subcomponents refine the contract. This dialog window shows the Contract Properties in the format: *InstanceName.ContractProperty*. This allows the selection of instance-based Contract Properties (instead of type-based).



	contract	range
☐	bscu.safety	
☐	bscu.cmd_time	
☐	hydraulic[0..(4 - 1)].brake_time	0..(4 - 1)

Figure 20. Refinement Selection

Be sure that the *Aggregation* kind of the instances is set to *composite* as shown in the bottom-right part in the Figure 21. This is required to let "Set Contract Refinement" command to work properly.

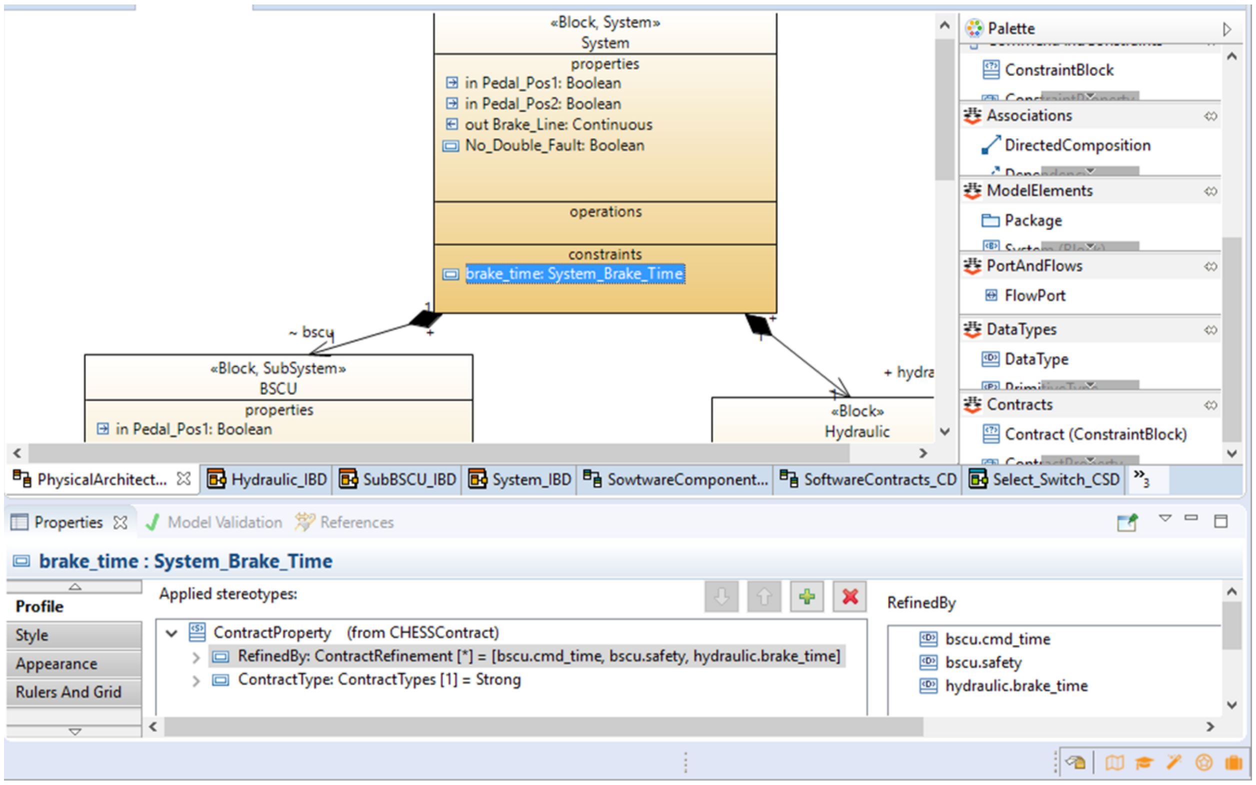


bscu : BSCU

Visibility: package
Multiplicity: 1
Aggregation: composite

Figure 21. Composite Aggregation

- The information about the refinement is set in the "RefinedBy" attribute of the "ContractProperty" stereotype of a Block and available in the "Profile" tab of the Property view (see Figure 22):



brake_time : System_Brake_Time

Applied stereotypes:

ContractProperty (from CHESSContract)
RefinedBy: ContractRefinement [*] = [bscu.cmd_time, bscu.safety, hydraulic.brake_time]
ContractType: ContractTypes [1] = Strong

RefinedBy
bscu.cmd_time
bscu.safety
hydraulic.brake_time

Figure 22. Contract Refinement

13. Specify Component Behavior

13.1 Nominal Behavior

Nominal behavior of a component can be provided by using UML State Machine diagrams. State Machines are usually expressed in terms of *discrete* time domain, CHESS supports also the *timed* time domain. In the case of a *timed* UML State Machine, the behavior is described by a timed automaton, a non-deterministic finite state machine extended by finitely many real-valued clocks, using timed and discrete transitions. A classic nominal behavior defined in CHESS using state machine comes with the following restricted elements:

- An initial state: The state representing the initial step of the behavior
- Basic states: As many states as necessary to represent the component behavior
- Transitions: A transition comes with a guard and an effect only. The guard is a boolean condition upon the values of the component properties. The effect must be expressed by using the CleanC language, to model component properties assignment. The transition coming from the initial node (i.e. the initial transition) has always its guard true and its effect represents the initial values of the component properties.
- State invariants: A boolean condition or restriction on the values of the component properties, expressed using the CleanC language as a state constraint. The property must be fulfilled in order to be able to remain in the state. In the case of a *timed* UML State Machine, the condition or

restriction on clocks should be convex, in the sense of a conjunction of inequalities on the clock (e.g. the condition “**x<0 && x>3**” is **not valid**).

Additional restrictions also apply: no UML concurrent regions, history states, choice or junction transition.

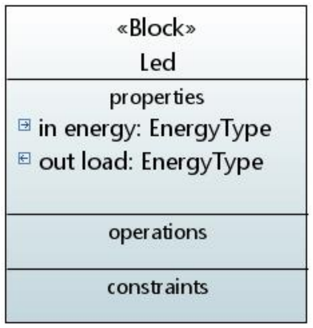


Figure 23. Example of a Led component

An example of a component is given **Figure 23**. It represents a component Led with two properties: energy and load.

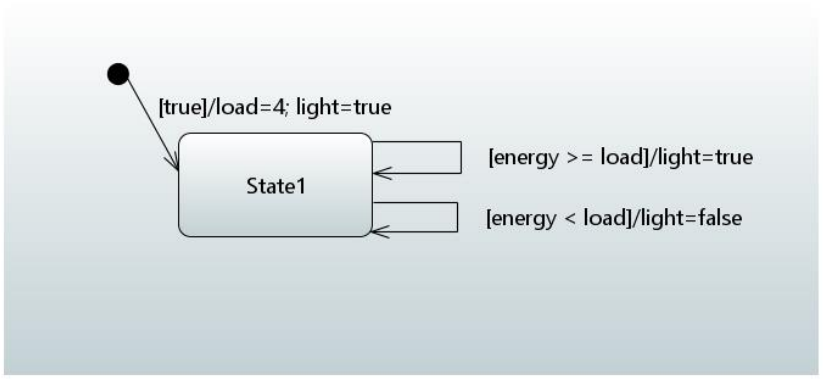

[true]/load=4; light=true
State1
[energy >= load]/light=true
[energy < load]/light=false

Figure 24. Example of nominal behavior for the Led component

Its nominal behavior is defined in the state machine described in **Figure 24**. At initialization, load is set to 4 and light is set true. There is only one state in this behavior with two transitions: if energy is greater or equal to load, the next assignment of light is true; if energy is lower than load, the next assignment of light is false.

To define the Guard and Effect on a transition in CHESS, the user needs to click on the transition and open the Properties view as described in **Figure 25**.

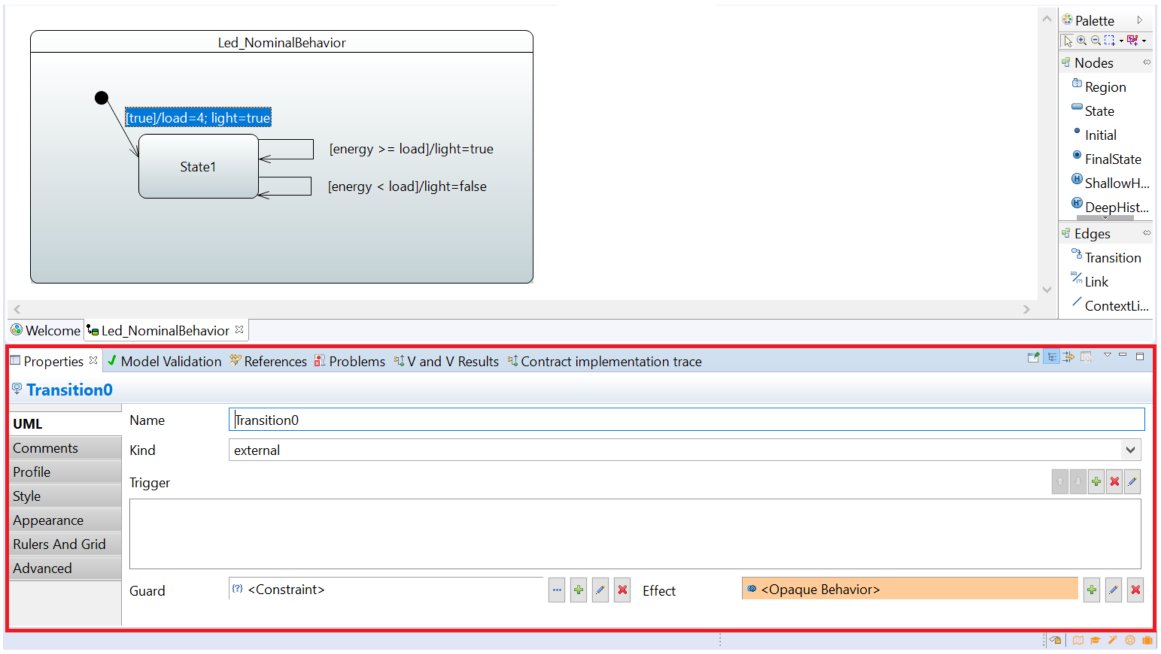


Figure 25. Properties view

For the Guard, the user needs to define a Constraint as described in **Figure 26**.

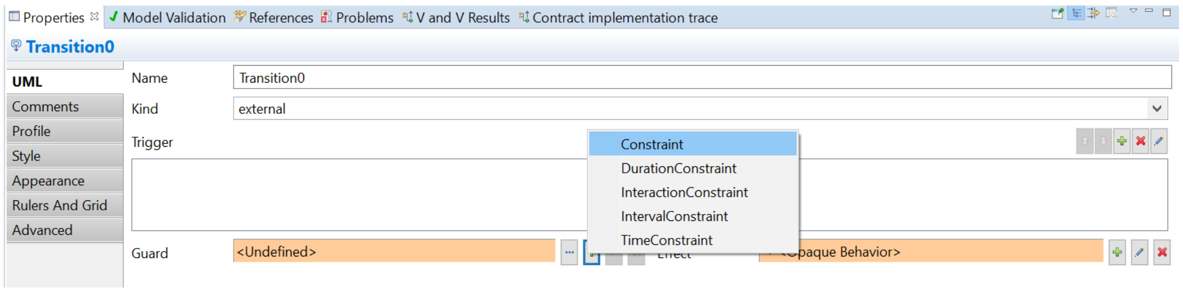


Figure 26. Constraint selection for Guard

The Constraint is defined using an OpaqueExpression, as shown in **Figure 27**.

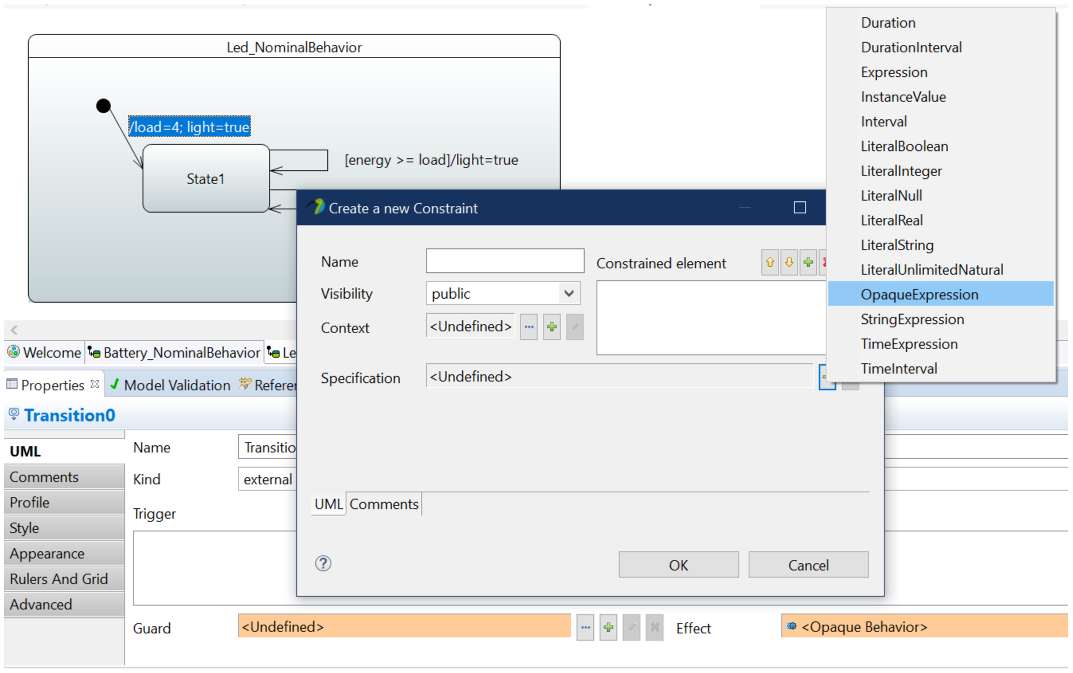


Figure 27. OpaqueExpression selection for Constraint

The OpaqueExpression must be defined in the **CleanC** language (The user has to manually set it) as described in **Figure 28**.

Appendix D. CleanC Language, the imperative language to define transition guards and effects. describes the rules of the language.

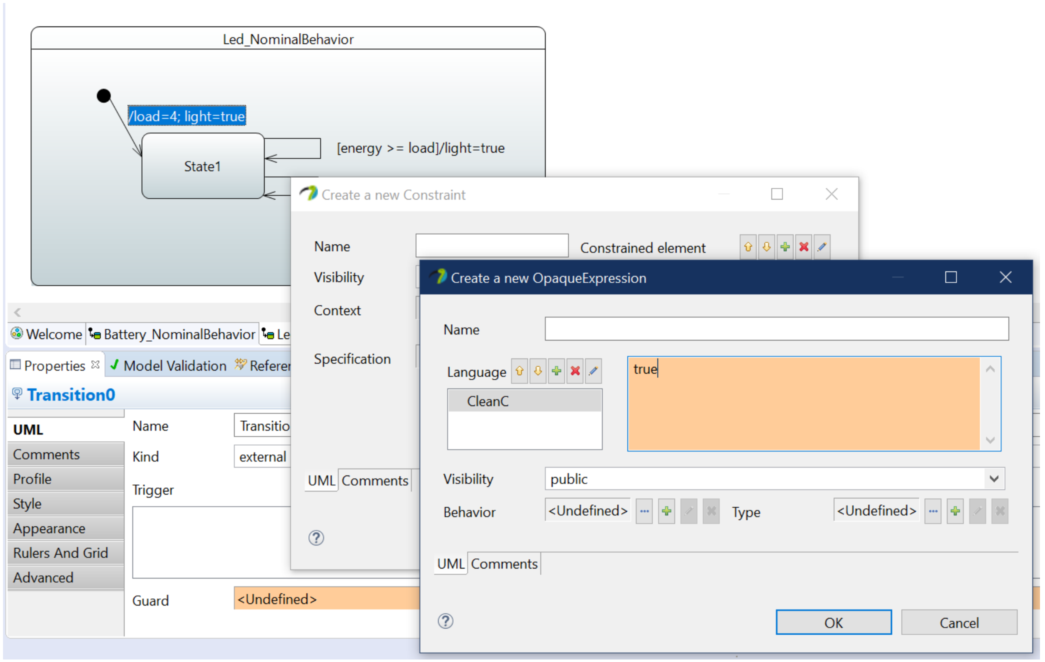
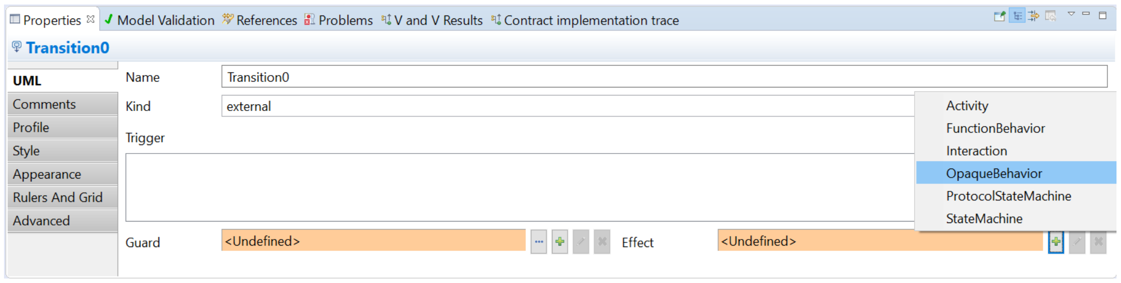


Figure 28. OpaqueExpression definition in CleanC language

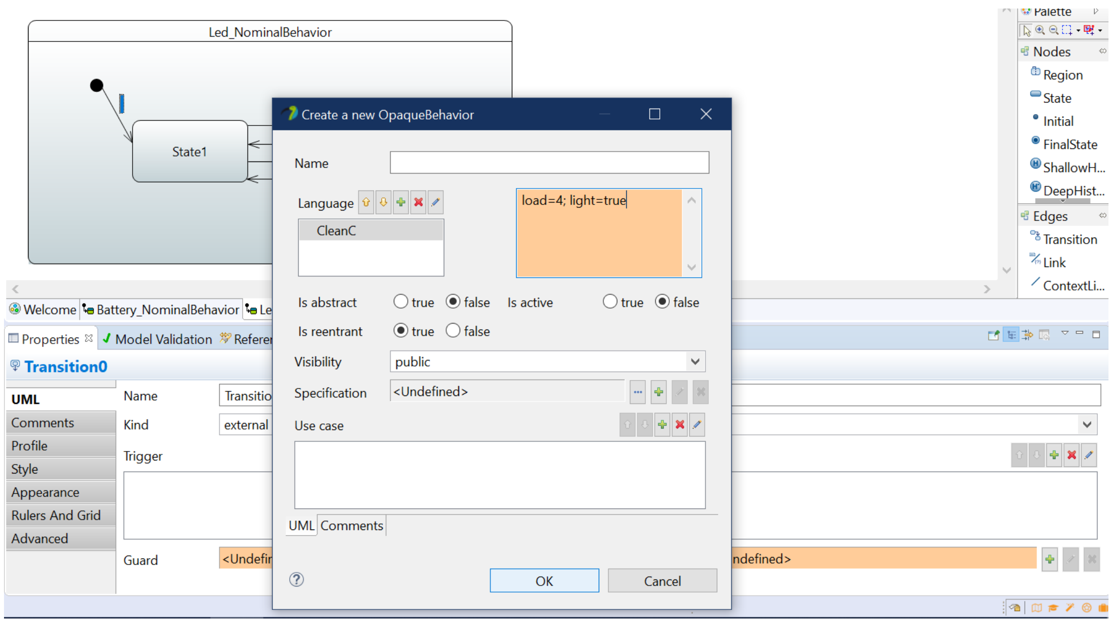
For the Effect, the user needs to define an Opaque Behavior as described in **Figure 29**.



Name: Transition0
Kind: external
Trigger
Guard: <Undefined>
Effect: <Undefined>

Figure 29. OpaqueBehavior selection for Effect

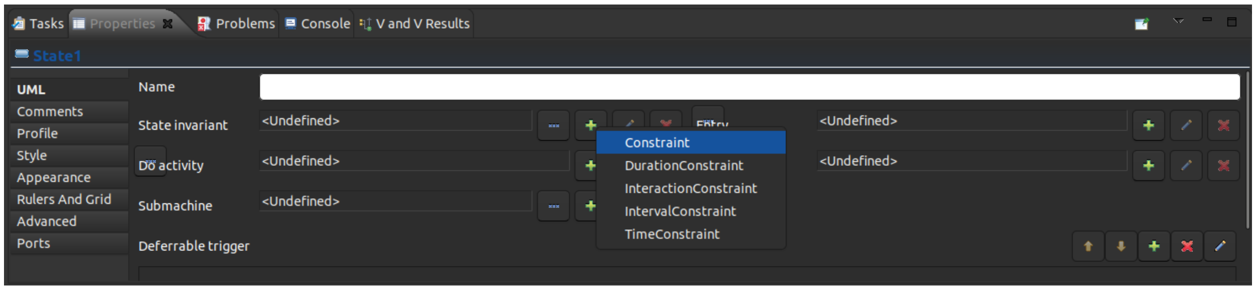
The OpaqueBehavior must be defined in the CleanC language as shown in **Figure 30**.



Name
Language: CleanC
load=4; light=true
Is abstract: true [✓ false]
Is active: true [✓ false]
Is reentrant: [✓ true] false
Visibility: public
Specification: <Undefined>
Use case

Figure 30. OpaqueBehavior definition in CleanC language

For the State Invariant, the user needs to define a Constraint as described in **Figure 31**.



Name
State invariant: <Undefined>
Do activity: <Undefined>
Submachine: <Undefined>
Deferrable trigger

Figure 31. Constraint selection for State invariant.

The Constraint is defined using an OpaqueExpression, as shown in **Figure 32**.

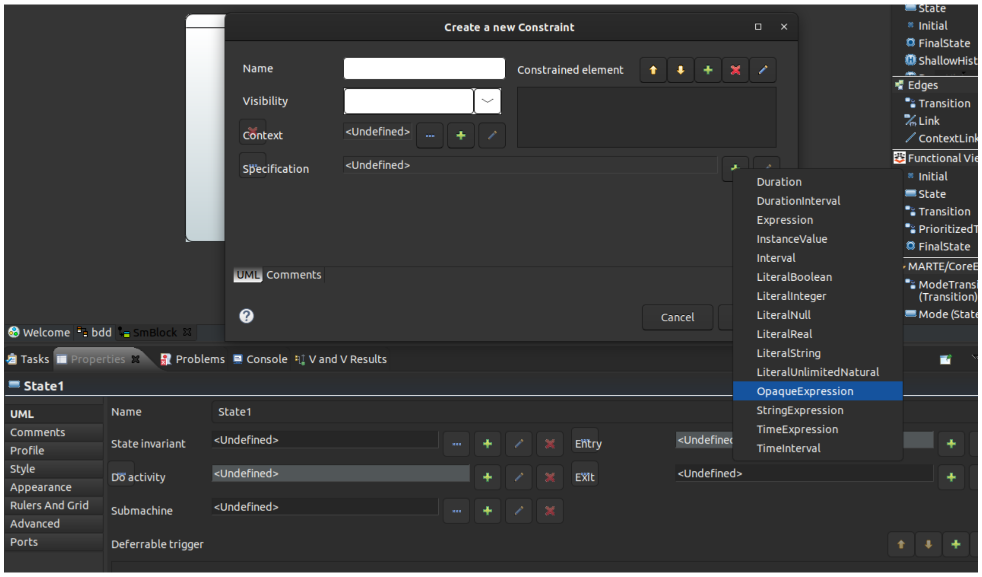


Figure 32. OpaqueExpression selection for constraint.

The OpaqueExpression must be defined in the **CleanC** language (The user has to manually set it) as described in **Figure 33.**

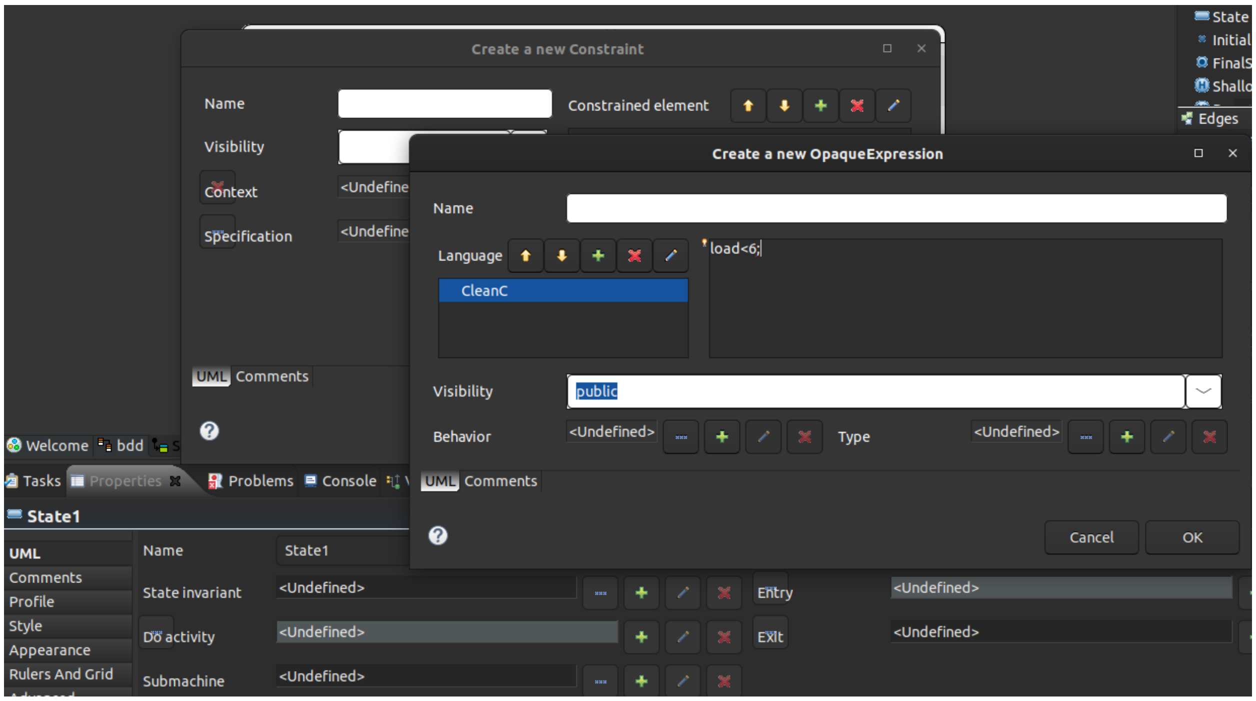


Figure 33. OpaqueExpression definition in CleanC language.

State Invariants can be displayed by dragging the constraint from the Model Explorer View into the state machine diagram as described in **Figure 34**.

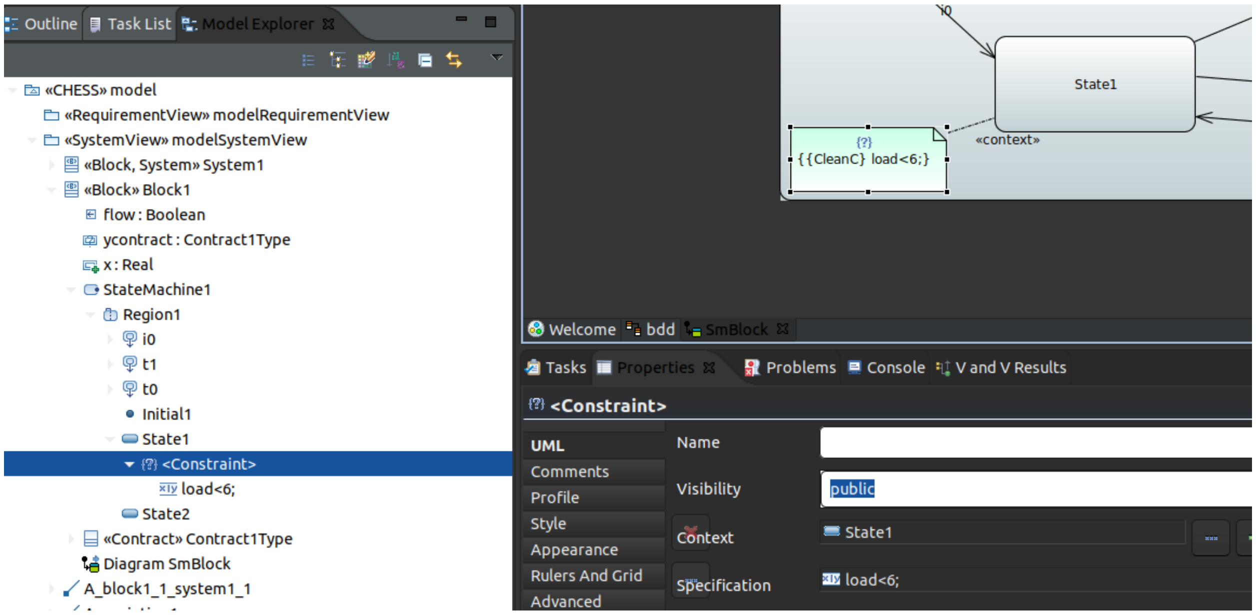


Figure 34. Displaying a state invariant.

The light attribute used in the state machine in **Figure 24** is not visible in the Led block in **Figure 23**. It is a local attribute of the Led component used only in the state machine. Local attributes to use only in UML state machine diagrams are created as follows:

1. In the Model Explorer View, select the owner component of the local attribute.
2. Right click "**New Child → Property**".
3. Select the Property (that represents the local attribute) and in the Property View – "UML" tab set the name and the Type of the element.

An example for the Led component is given **Figure 35**.

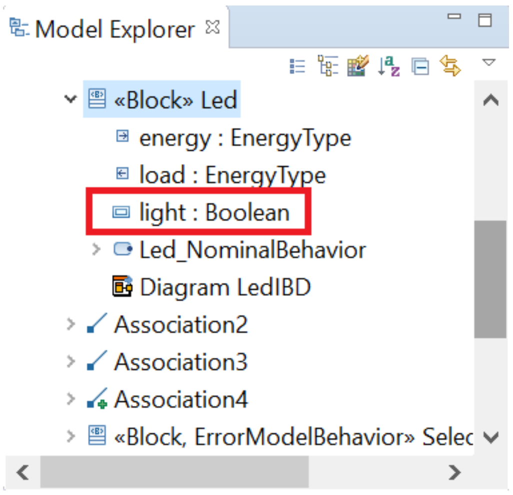


Figure 35. Example of local attribute

13.2 Faulty Behavior

Faulty behavior for a component can be provided by using the CHESS dependability profile.

Several possibilities exist in CHESS to model the faulty behavior of a component.

For instance, faulty behavior can be modelled in a dedicated state machine stereotyped with ErrorModel; the ErrorModel represents a particular kind of state machine containing information about faults, errors and failure modes and their propagation internal to a given component.

Once defined, the ErrorModel state machine can be attached to a given component by stereotyping the component with ErrorModelBehaviour.

The ErrorModelBehaviour stereotype comes with the errorModel attribute, which has to be used to reference the actual ErrorModel state machine to be attached to the component (see Figure 36).

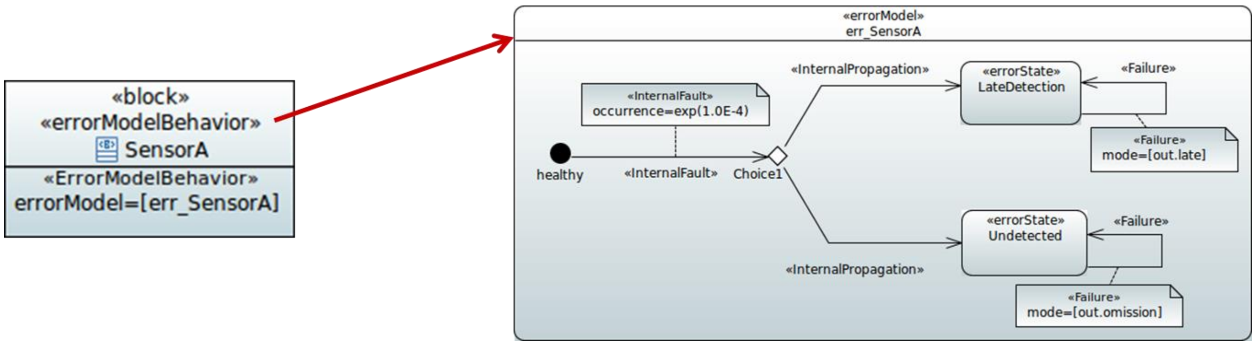


Figure 36. ErrorModel behavior

The following information can be provided through an ErrorModel state machine:

- Initial state
  - It represents the "healthy" state of the component
- Errors
  - UML State, with the «ErrorState» stereotype
- Internal faults
  - UML Transitions, with the «InternalFault» stereotype
    - connecting the initial state and an error state
    - occurrence – time to fault occurrence (time distribution)
- Internal propagations
  - UML Transitions, with the «InternalPropagation» stereotype
    - delay – time after which propagation occur
    - weight - relative probability of occurrence
    - externalFaults – Boolean expression on the occurrence of external faults (i.e., failures incoming on input ports of the component)
- Failures
  - UML Transition, with the «Failure» stereotype
    - mode – the failure mode(s) under which the failure manifest itself on the port(s) of the component

13.3 Fault injection with probability

As seen before, the faulty behavior for a component is provided through a state machine, the latter tagged with the *«ErrorModel»* stereotype defined in the CHESS dependability profile.

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

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

Configuring the *«ErrorState»* stereotype it is possible to assign a probability for the fault to happen.

As an example, the state machine shown in Figure 37 represents an error model. In case of an internal fault, the component enters in an error state where the property "energy" is fixed at 0 value. The probability for this fault to happen is set to 0.05.

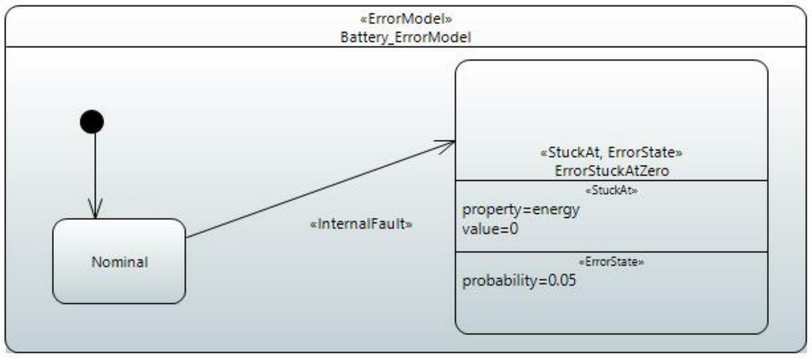


Figure 37. An ErrorModel state machine with probability

14. Manage Analysis Contexts

Analysis contexts are used in CHESS to collect information about a given analysis to be run; an analysis context is used at least to refer to the set of entities that must be considered for the analysis. If needed, the analysis context can be used to provide analysis specific configuration parameters.

Analysis context can be created in the CHESS Dependability View by using class diagrams.

15. Perform Fault Tree Analysis

Once the system architecture has been provided, by mean of components definition and their nominal and error models, the Fault Tree Analysis (FTA) can be obtained by invoking the xSAP symbolic model checker through the CHESS environment.

To obtain the fault tree, perform the following steps:

- Select a package from the Model Explorer View.
- Right click on the package and go to "**CHESS → Architecture Verification → Fault Tree Analysis (FTA)**" (see Figure 38).
- A popup will appear. From the list, select the Analysis Context from which the analysis has to be started and click "OK".
- A popup will appear to select the time model of the architecture. Only *discrete* is supported at the moment.

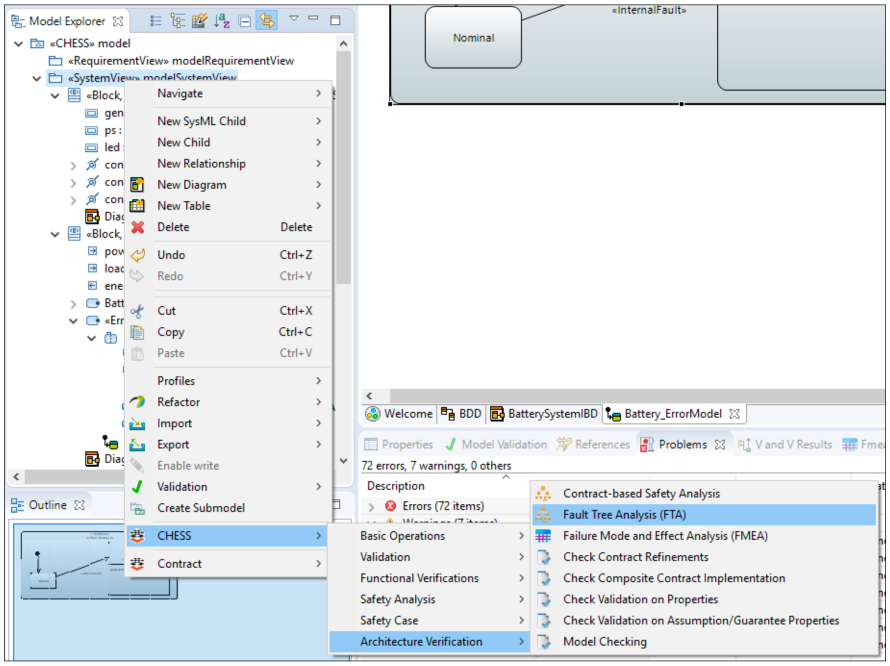


Figure 38. FTA command

Once the analysis is executed, the fault tree is automatically shown in a dedicated panel in the frontend; see Figure 39 as an example of resulting fault tree. Then the fault tree can be examined, in particular the minimal cut-set and so the basic fault conditions which can lead to the top-level failure. If probabilities were specified for the faults to happen, they will be computed and reported in the resulting tree.

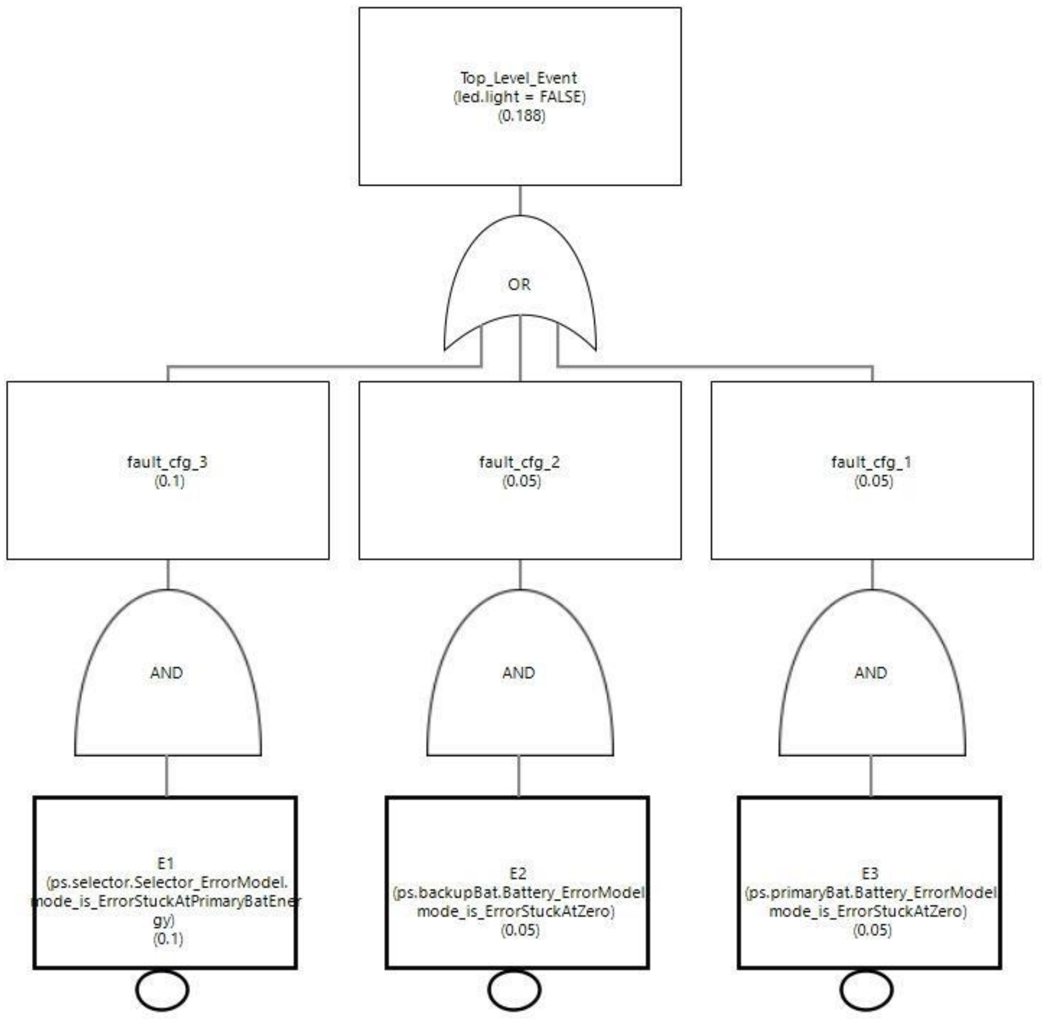

Top_Level_Event
(led.light = FALSE)
(0.188)
OR
fault_cfg_3
(0.1)
fault_cfg_2
(0.05)
fault_cfg_1
(0.05)
AND
AND
AND
E1
(ps.selector.Selector_ErrorModel.
mode_is_ErrorStuckAtPrimaryBatEnergy)
(0.1)
E2
(ps.backupBat.Battery_ErrorModel
mode_is_ErrorStuckAtZero)
(0.05)
E3
(ps.primaryBat.Battery_ErrorModel
mode_is_ErrorStuckAtZero)
(0.05)

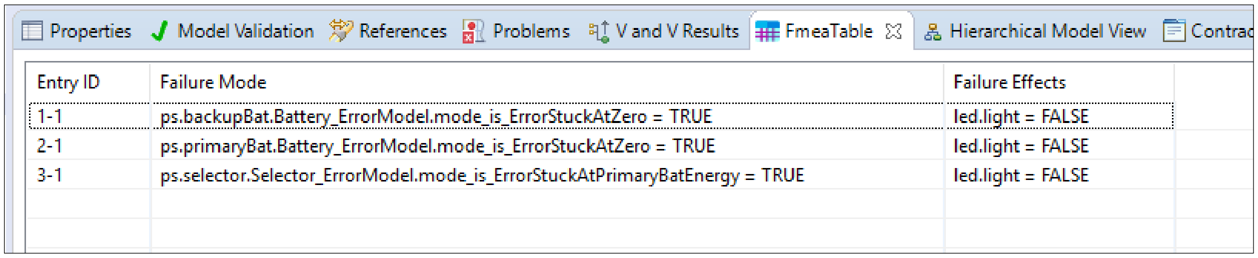
Figure 39. Fault Tree Analysis result

16. Perform Failure Mode and Effect Analysis

Along with the fault tree generation (see Section 14), it is possible to generate the Failure Model and Effect Analysis (FMEA) table. FMEA is obtained in a similar mode as the fault tree analysis:

- Select a package from the Model Explorer View.
- Right click on the package and go to "**CHESS → Architecture Verification → Failure Mode and Effect Analysis (FMEA)**".
- A popup will appear. From the list, select the Analysis Context from which the analysis has to be started and click OK.
- A popup will appear to select the time model of the architecture. Only *discrete* is supported at the moment.

Once run, the resulting table is visualized in a specific view inside CHESS, see Figure 40.



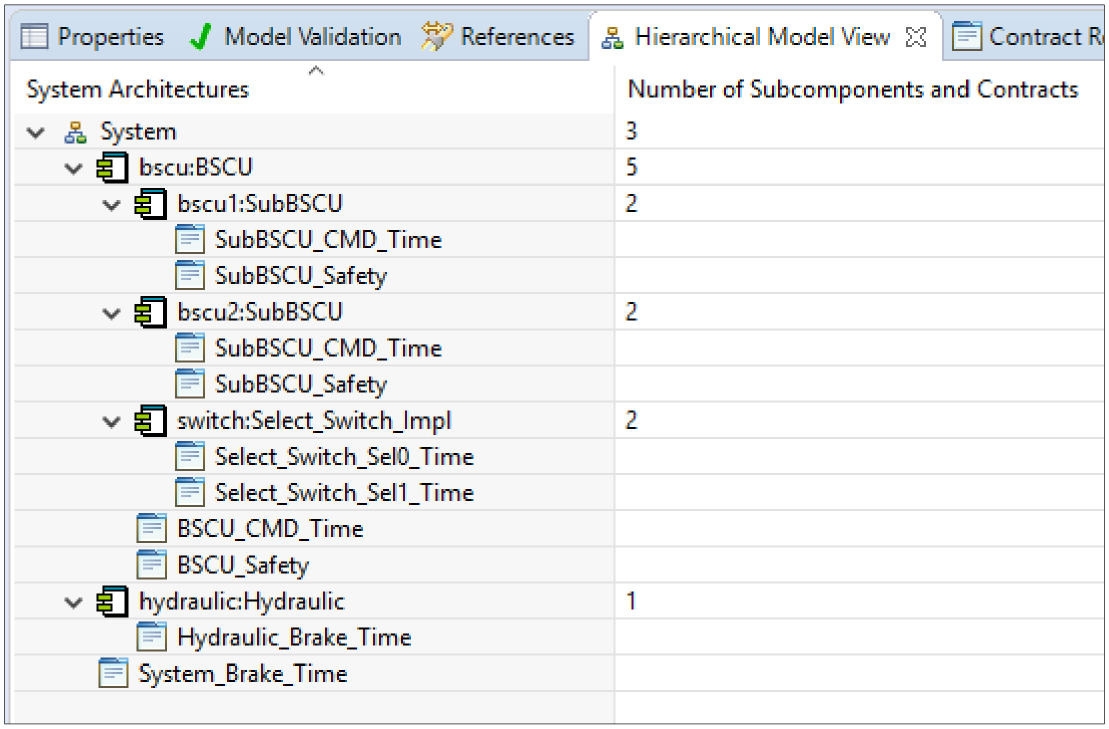
Entry ID	Failure Mode	Failure Effects
1-1	ps.backupBat.Battery_ErrorModel.mode_is_ErrorStuckAtZero = TRUE	led.light = FALSE
2-1	ps.primaryBat.Battery_ErrorModel.mode_is_ErrorStuckAtZero = TRUE	led.light = FALSE
3-1	ps.selector.Selector_ErrorModel.mode_is_ErrorStuckAtPrimaryBatEnergy = TRUE	led.light = FALSE

Figure 40. The FMEA table

17. View Status of System Architecture

CHESS provides a hierarchical view that shows the decomposition of the system component into sub-components. It shows also the contracts assigned for each component. The system is graphically represented as the top element of the view, see Figure 41.

To show the view, go to "**Window → Show View → Hierarchical Model View**".

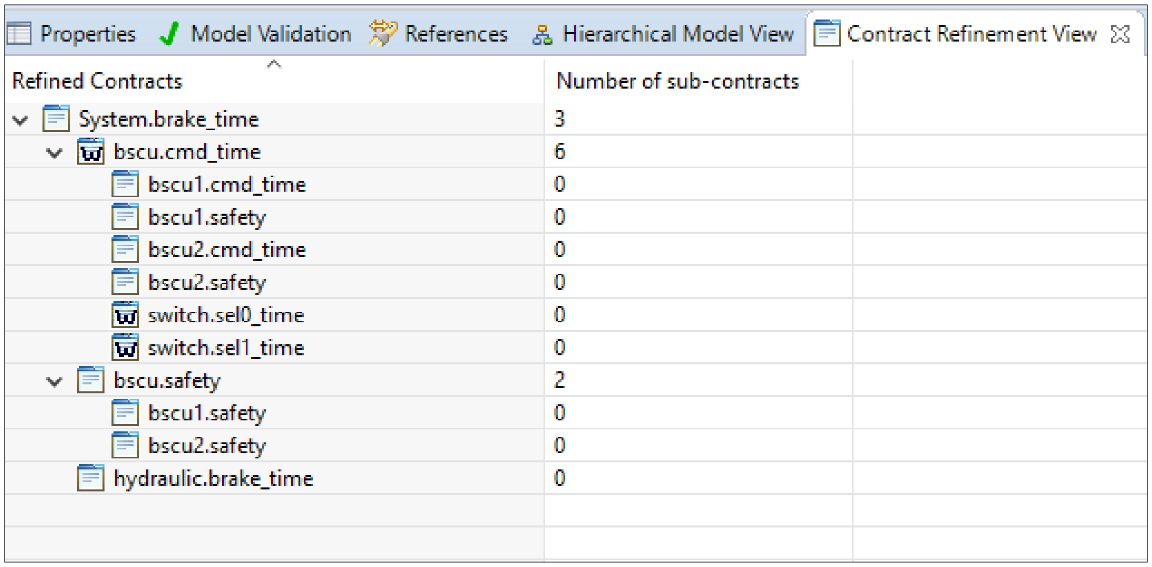


System Architectures	Number of Subcomponents and Contracts
System	3
bscu:BSCU	5
bscu1:SubBSCU	2
SubBSCU_CMD_Time	
SubBSCU_Safety	
bscu2:SubBSCU	2
SubBSCU_CMD_Time	
SubBSCU_Safety	
switch:Select_Switch_Impl	2
Select_Switch_Sel0_Time	
Select_Switch_Sel1_Time	
BSCU_CMD_Time	
BSCU_Safety	
hydraulic:Hydraulic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

Figure 41. 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. The weak contracts are graphically represented as a document with a "W" on top, see Figure 42.

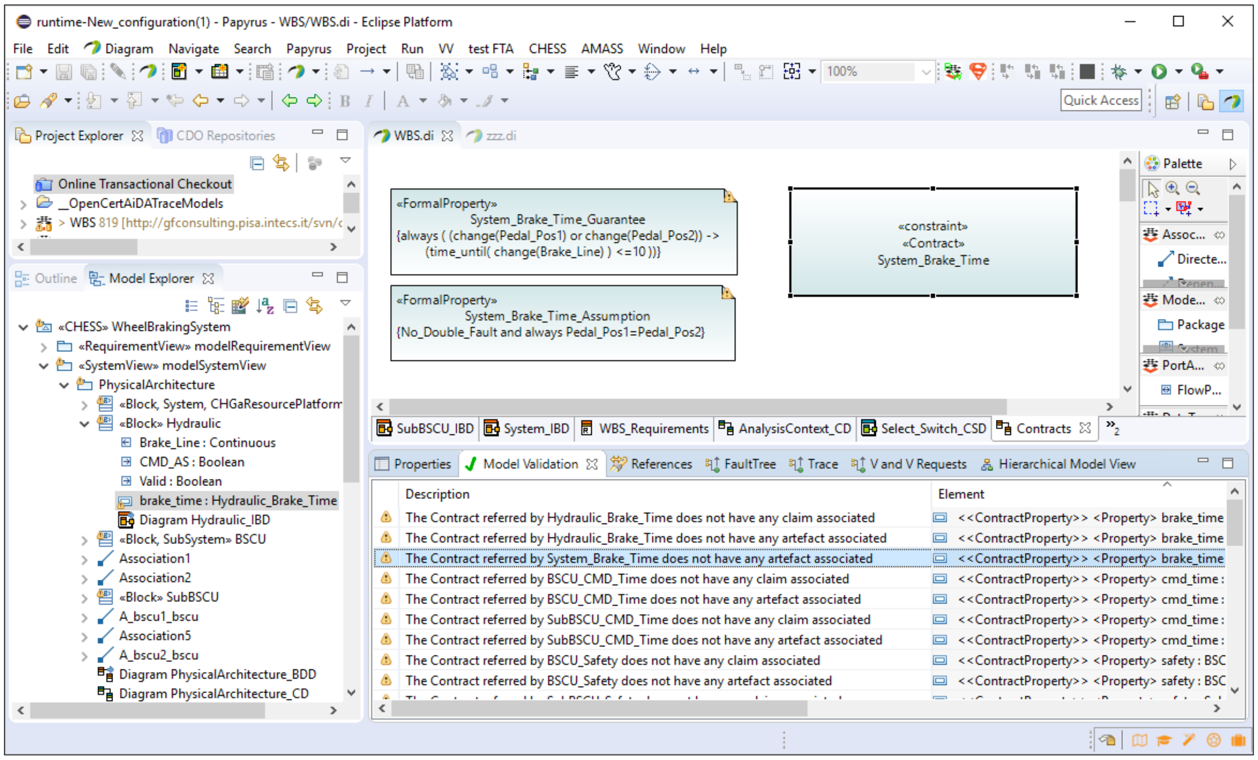
To show the view, go to "**Window → Show View → Contract Refinement View**".



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 42. Contract Refinement View

A report about contracts status with respect to the assurance can be also generated through CHESS; from the Model Explorer View coming with Papyrus/CHESS, right click and select "**Validation → CHESS → Validate Contracts for Assurance**", then issues with the defined contracts are reported in the Model Validation view as shown in Figure 43.



«FormalProperty»
System_Brake_Time_Guarantee
{always ((change(Pedal_Pos1) or change(Pedal_Pos2)) ->
(time_until(change(Brake_Line)) <=10))}

«FormalProperty»
System_Brake_Time_Assumption
{No_Double_Fault and always Pedal_Pos1=Pedal_Pos2}

«constraint»
«Contract»
System_Brake_Time

Description	Element
The Contract referred by Hydraulic_Brake_Time does not have any claim associated	<<ContractProperty>> <Property> brake_time
The Contract referred by Hydraulic_Brake_Time does not have any artefact associated	<<ContractProperty>> <Property> brake_time
The Contract referred by System_Brake_Time does not have any artefact associated	<<ContractProperty>> <Property> brake_time
The Contract referred by BSCU_CMD_Time does not have any claim associated	<<ContractProperty>> <Property> cmd_time :
The Contract referred by BSCU_CMD_Time does not have any artefact associated	<<ContractProperty>> <Property> cmd_time :
The Contract referred by SubBSCU_CMD_Time does not have any claim associated	<<ContractProperty>> <Property> cmd_time :
The Contract referred by SubBSCU_CMD_Time does not have any artefact associated	<<ContractProperty>> <Property> cmd_time :
The Contract referred by BSCU_Safety does not have any claim associated	<<ContractProperty>> <Property> safety : BSC
The Contract referred by BSCU_Safety does not have any artefact associated	<<ContractProperty>> <Property> safety : BSC

Figure 43. Validating contracts status

18. Perform Check of Contract Refinement

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

- Select a component (in the Model Explorer View) or the corresponding graphical representation (in the diagram editor). The contract refinements considered will be the ones associated to the selected component and the ones associated to its sub components. This operation includes recursively all the contracts along the subcomponents, from the root to the leaves of the system.
- Perform the check contract refinement: right click on the selected component, then go to "**CHESS → Functional Verification → Check Contract Refinement on selected component**" (see Figure 44).
- A popup will appear to select the time model of the architecture. Options supported are *discrete, hybrid* and *timed*.

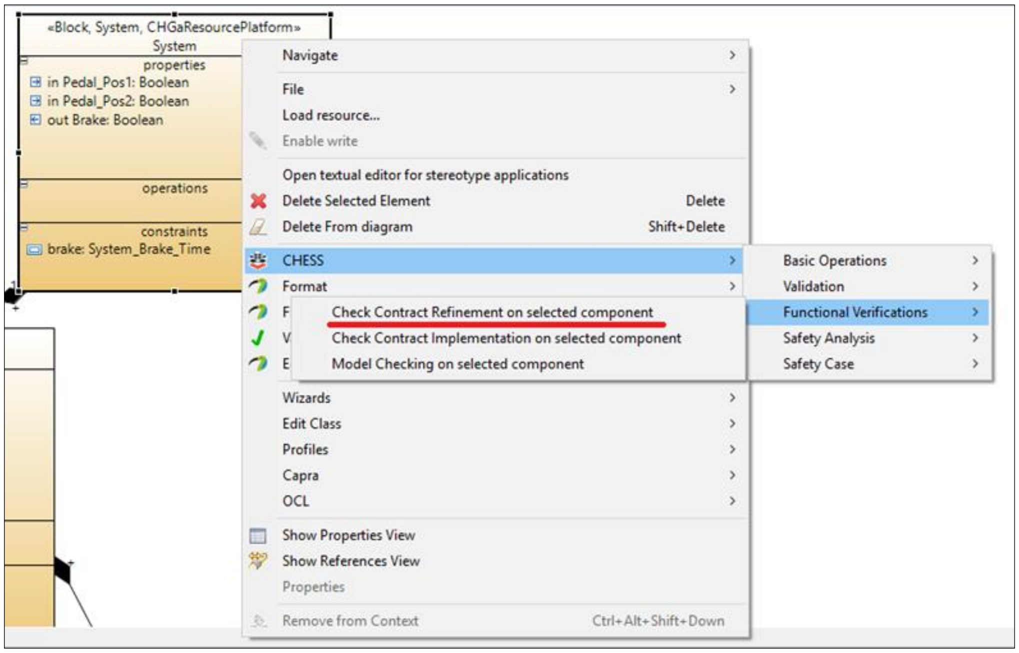


Figure 44. Perform the check of the contract refinements

19. Perform Check of Component Implementation on Contracts

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

- Select a component (in the Model Explorer View) or the corresponding graphical representation (in the diagram editor). The contracts and state machines considered will be the ones associated to the selected component and the ones associated to its sub components. This operation includes recursively all the contracts and state machines along the subcomponents, from the root to the leaves of the system.
- Right click on the selected component, then go to "**CHESS → Functional Verifications → Check Contract Implementation on selected component**" (see Figure 45).

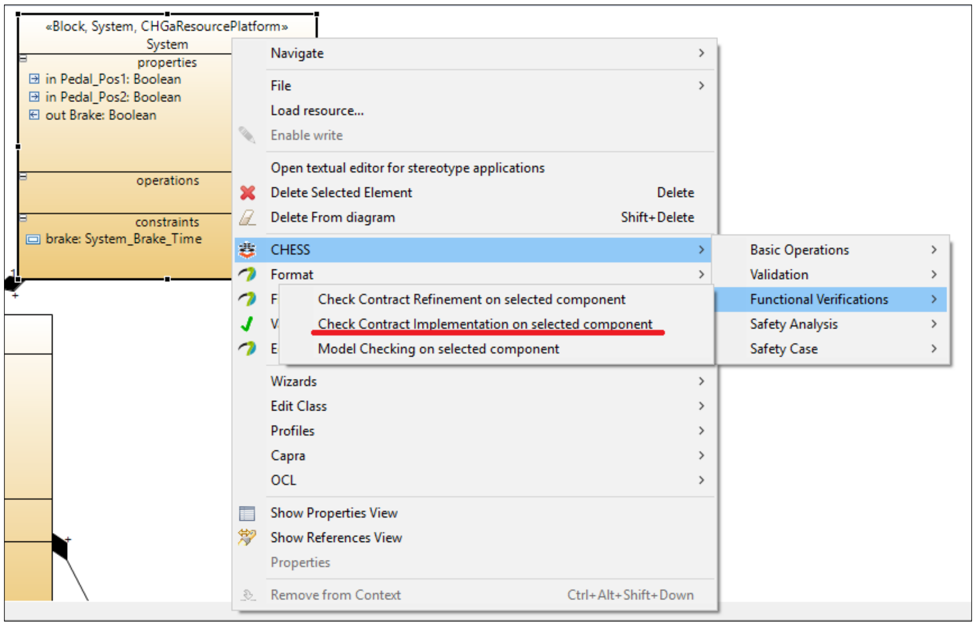


Figure 45. Perform the check implementation based on contracts

To verify that the behavior of the entire system defined in the model satisfies all the contracts, perform the following steps:

- Select a root component (in the Model Explorer View) or the corresponding graphical representation (in the diagram editor). The contracts and state machines considered will be the ones associated to the selected component and the ones associated to its sub components. This operation includes recursively all the contracts and state machines along the subcomponents, from the root to the leaves of the system.
- Right click on the selected component, then go to "**CHESS → Functional Verifications → Check Composite Contract Implementation on selected component**".
- A popup will appear to select the time model of the architecture. Options supported are *discrete* and *timed*.

20. Perform Consistency Check of Assumption/Guarantee Formal Properties

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

- Select a component (in the Model Explorer View) or the corresponding graphical representation (in the diagram editor). The properties available to check will be the assumptions and guarantees of contracts belonging to the selected component and to its sub components. This operation includes recursively all the assumptions and guarantee properties from the root to the leaves of the selected component.
- Right click on the selected component, then go to "**CHESS → Validation → Check Validation on Assumption/Guarantee Properties on selected component**" (see Figure 46).
- A popup will appear to select the time model of the architecture. Options supported are *discrete, hybrid* and *timed*.

- A popup appears to set the parameters of the command (see Figure 47, see OCRA user manual[4] for more details).

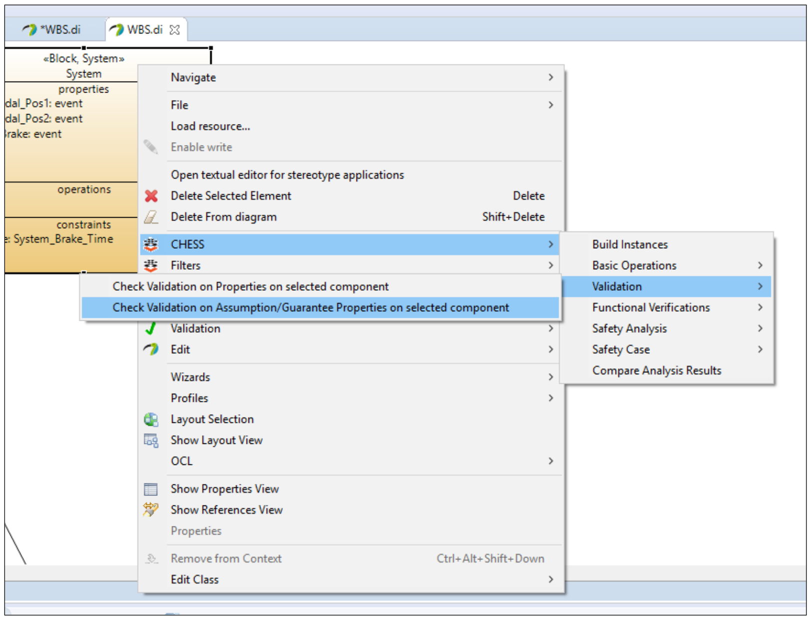
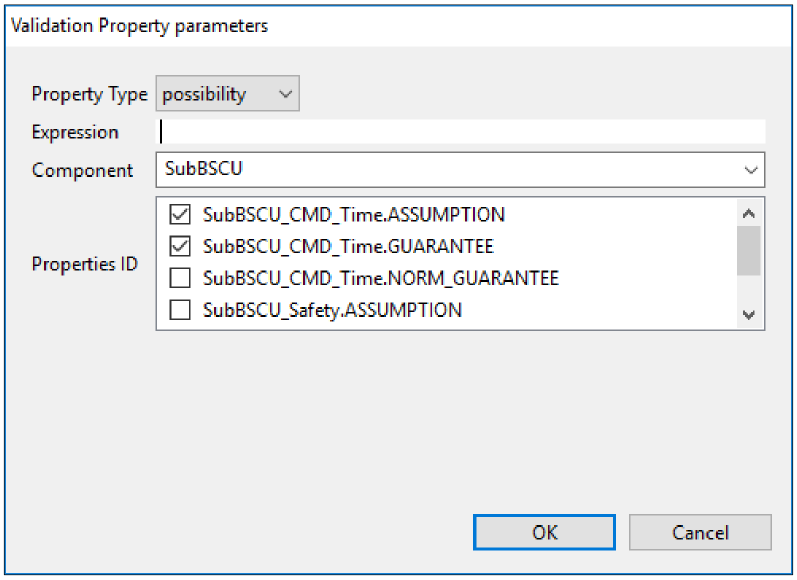


Figure 46. Perform the consistency check of assumption/guarantee formal properties


Validation Property parameters

Property Type: possibility

Expression:

Component: SubBSCU

Properties ID:
- [x] SubBSCU_CMD_Time.ASSUMPTION
- [x] SubBSCU_CMD_Time.GUARANTEE
- [ ] SubBSCU_CMD_Time.NORM_GUARANTEE
- [ ] SubBSCU_Safety.ASSUMPTION

Figure 47. Validation Property parameters

21. Perform Consistency Check of Formal Properties

In Section 19 it is described how to check assumptions and guarantees formal properties. Similarly, a validation can be done to all the formal properties of a component. To verify the formal property, perform the following steps:

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

- Select a component (in the Model Explorer View) or the corresponding graphical representation (in the diagram editor). The properties available to check will be the formal properties belonging to the selected component and to its sub components. This operation includes recursively all the properties from the root to the leaves of the selected component.
- Right click on the selected component, then go to "**CHESS → Validation → Check Validation on Assumption/Guarantee Properties on selected component**" (see Figure 48).
- A popup will appear to select the time model of the architecture. Options supported are *discrete, hybrid* and *timed*.
- A popup appears to set the parameters of the command (see Figure 47, see OCRA user manual[5] for more details).

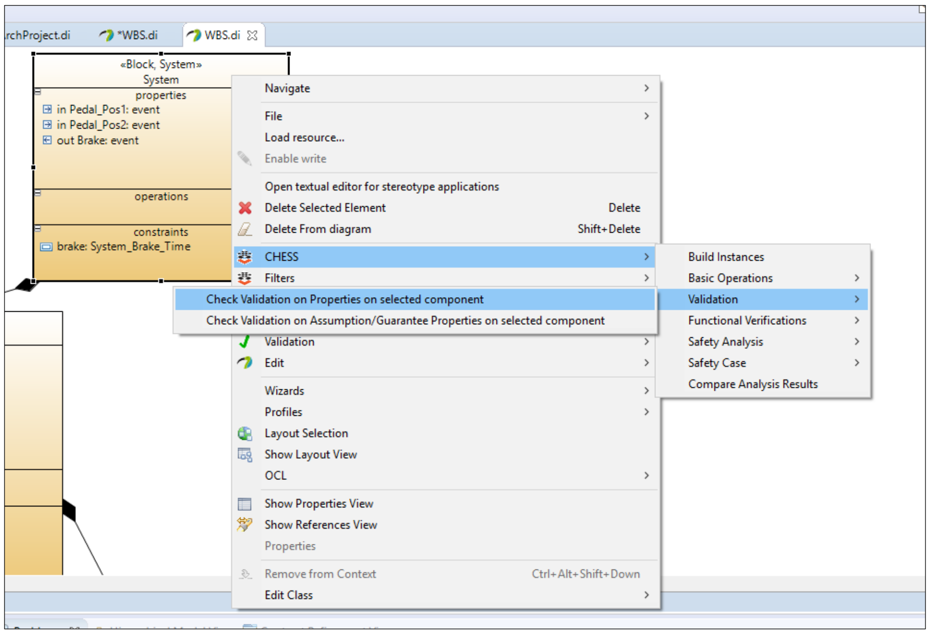


Figure 48. Perform the consistency check of formal properties

22. Perform Model Checking on Component Behavior

To execute the model checking, perform the following steps:

- Select a component (in the Model Explorer View) or the corresponding graphical representation (in the diagram editor). The components behavior to check will be the behavior of the selected component and the behavior of its sub components. This operation includes recursively all the behaviors from the root to the leaves of the selected component.
- Right click on the selected component, then go to "**CHESS → Functional Verification → Model Checking on selected component**" (see Figure 49).
- A popup appears to set the parameters of the command (see Figure 50, see nuXmv user manual[6] for more details).

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

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

- A popup will appear to select the time model of the architecture. Options supported are *discrete* and *timed*.

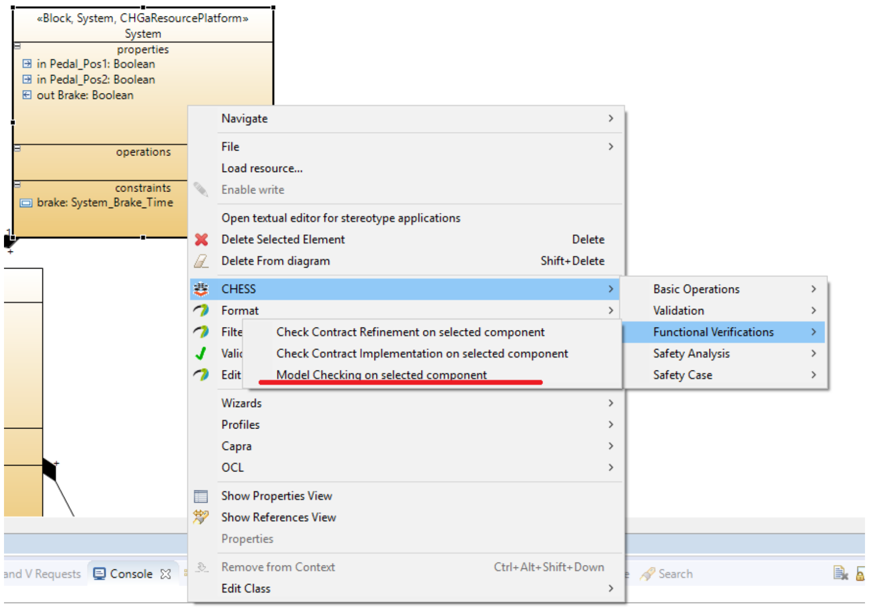
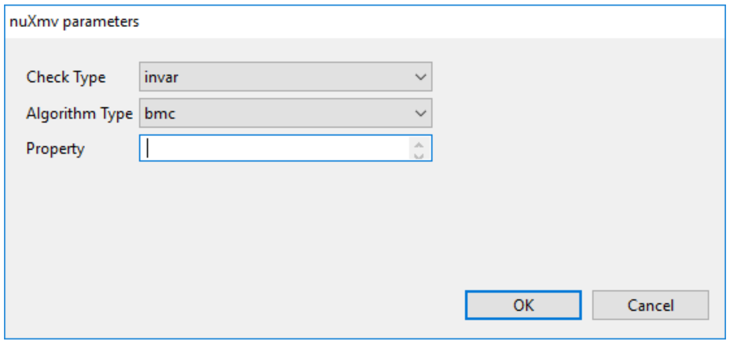


Figure 49. Perform the model checking of system behavior


nuXmv parameters

Check Type: invar
Algorithm Type: bmc
Property:

OK Cancel

Figure 50. Model Checking parameters

23. Generate Contract-based Fault Tree

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

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

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

- Select a component (in the Model Explorer View) or the corresponding graphical representation (in the diagram editor). The contracts considered will be the ones associated to the selected component and the ones associated to its sub components. This operation includes recursively all the contracts along the subcomponents, from the root to the leaves of the system.
- Right click on the selected component, then go to "**CHESS → Safety Analysis → Contract-based Safety Analysis on selected component**" (see Figure 51).

When the analysis is completed, the fault tree is shown. The representation is the same as the ones used for the Model-based Safety Analysis, see Figure 39, but with the probabilities set to 0.

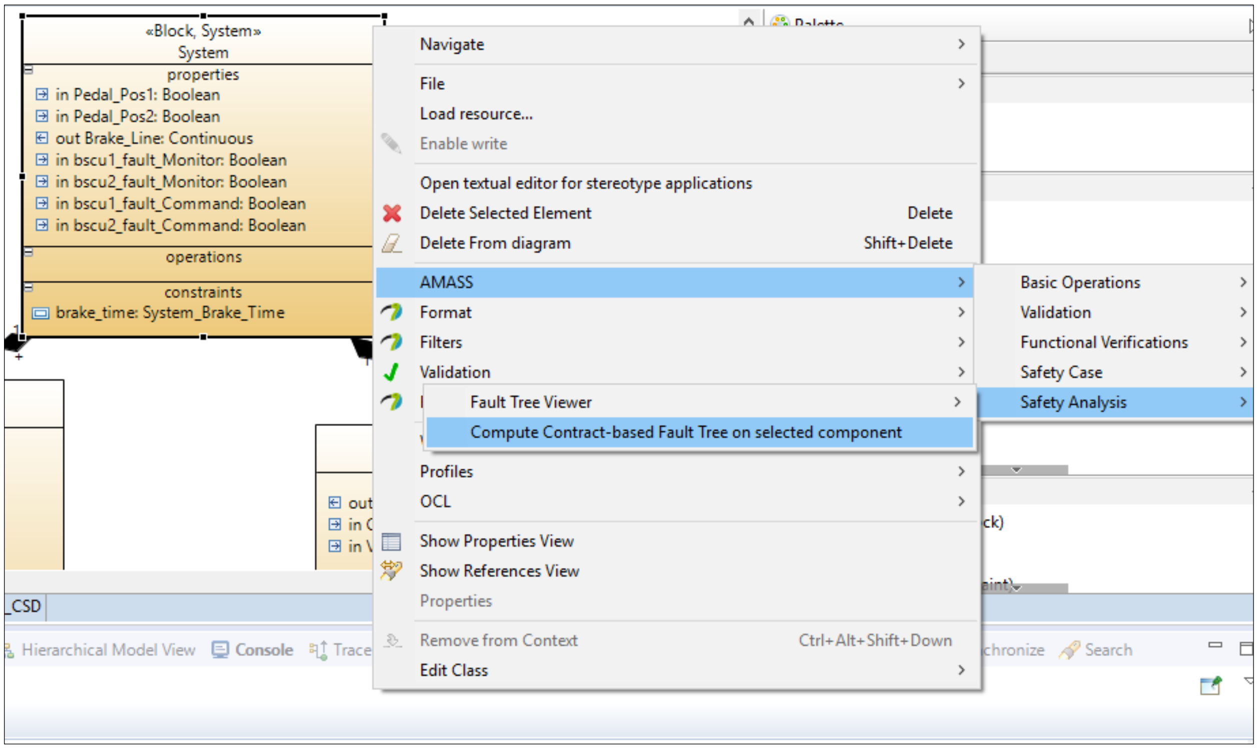
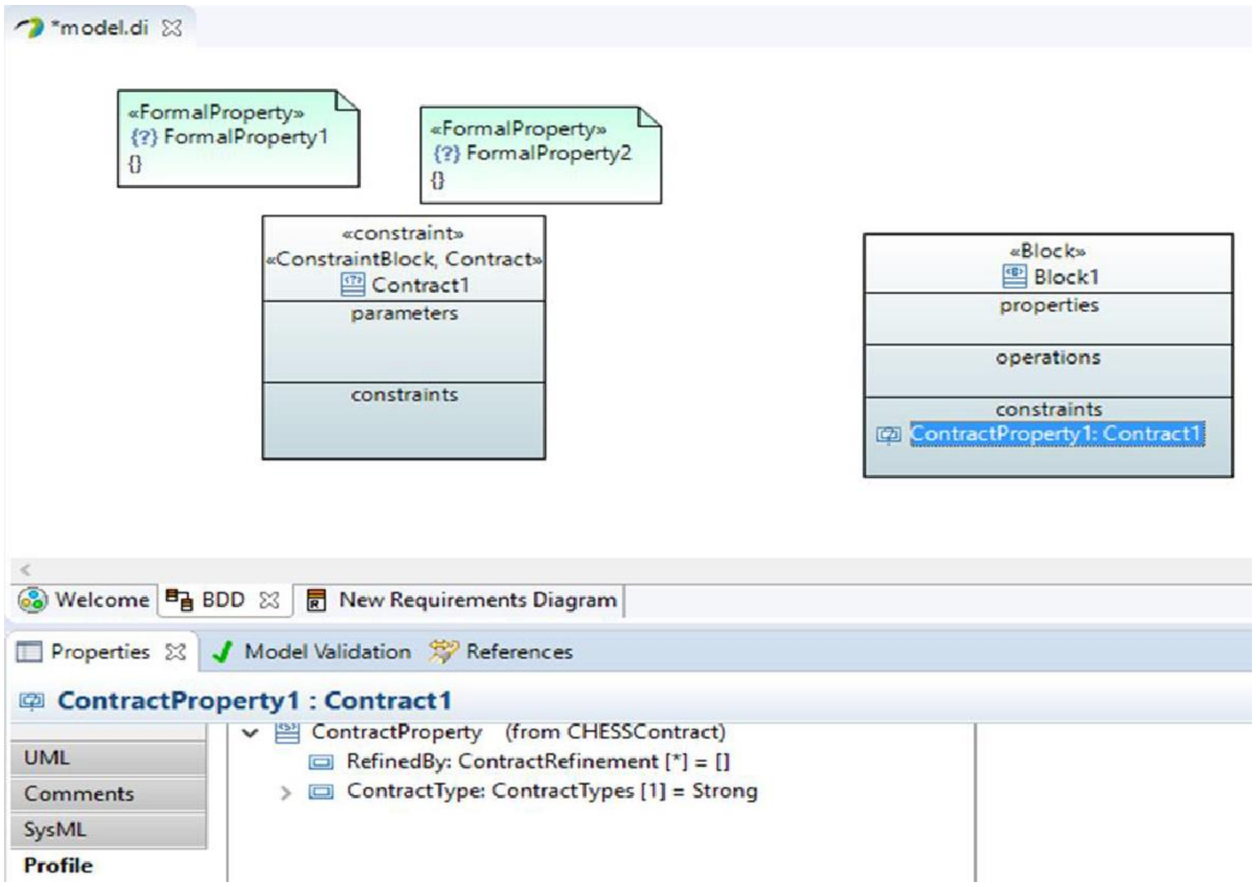

«Block, System»
System
properties
in Pedal_Pos1: Boolean
in Pedal_Pos2: Boolean
out Brake_Line: Continuous
in bscu1_fault_Monitor: Boolean
in bscu2_fault_Monitor: Boolean
in bscu1_fault_Command: Boolean
in bscu2_fault_Command: Boolean
operations
constraints
brake_time: System_Brake_Time
Navigate
File
Load resource...
Enable write
Open textual editor for stereotype applications
Delete Selected Element Delete
Delete From diagram Shift+Delete
AMASS
Format
Filters
Validation
Fault Tree Viewer
Compute Contract-based Fault Tree on selected component
Profiles
OCL
Show Properties View
Show References View
Properties
Remove from Context Ctrl+Alt+Shift+Down
Edit Class
Basic Operations
Validation
Functional Verifications
Safety Case
Safety Analysis
CSD
Hierarchical Model View
Console
Trace
Search

Figure 51. Dedicated menu to perform the compute the contract-based fault tree

24. Modelling Support and Contract-based Verification for Strong and Weak Contracts

The ContractProperty stereotype comes with the attribute "ContractType" which can be used to set the ***Strong*** or ***Weak*** property of the Contract associated to the architectural entity; While the strong assumptions and guarantees must be satisfied always in order for component to be used, the weak pairs offer additional information in some specific contexts where besides the strong assumptions, the weak assumptions are to be met as well.

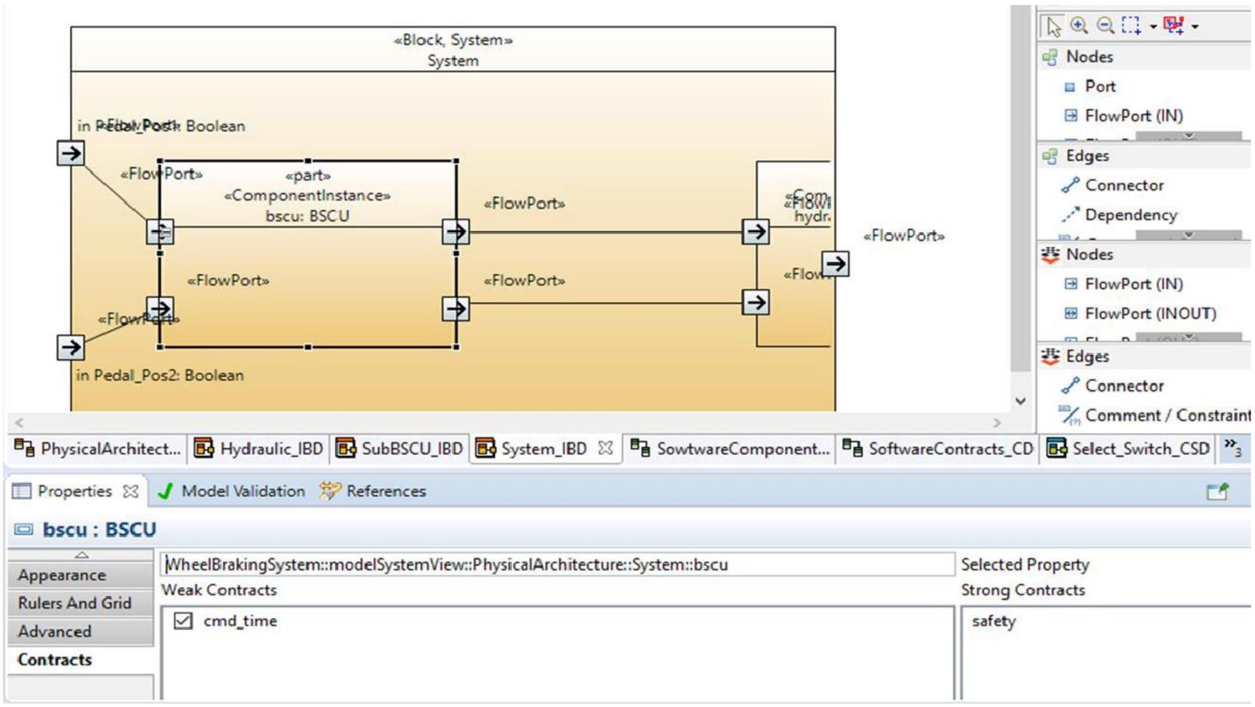


ContractProperty (from CHESSContract)
RefinedBy: ContractRefinement [*] = []
ContractType: ContractTypes [1] = Strong

Figure 52: set Strong/Weak property for contract

While a Strong contract associated to a Block/Component (must) hold for all the instances of the Block/Component, a Weak contract associated to a Block/Component can hold for a given instance of the Block/Component if the environment where the instance is placed met the assumption. So, for a given instance it must be possible to specify the set of weak contracts specified for the typing Block/Component (if any) which holds for the instance. To do so, the following steps must be performed:

- Select the Block/Component instance in the SysML Internal Block Diagram / UML Composite Structure Diagram.
- Apply the ComponentInstance stereotype to the instance.
- Select the “Contract” tab in the Properties editor (see figure below): The “Contract” tab shows the strong and weak contract inherited by the classifier typing the instance. In particular the Weak Contract area can be used to check the weak contracts that hold for the current instance.



WheelBrakingSystem::modelSystemView::PhysicalArchitecture::System::bscu

Weak Contracts	Strong Contracts
☑ cmd_time	safety

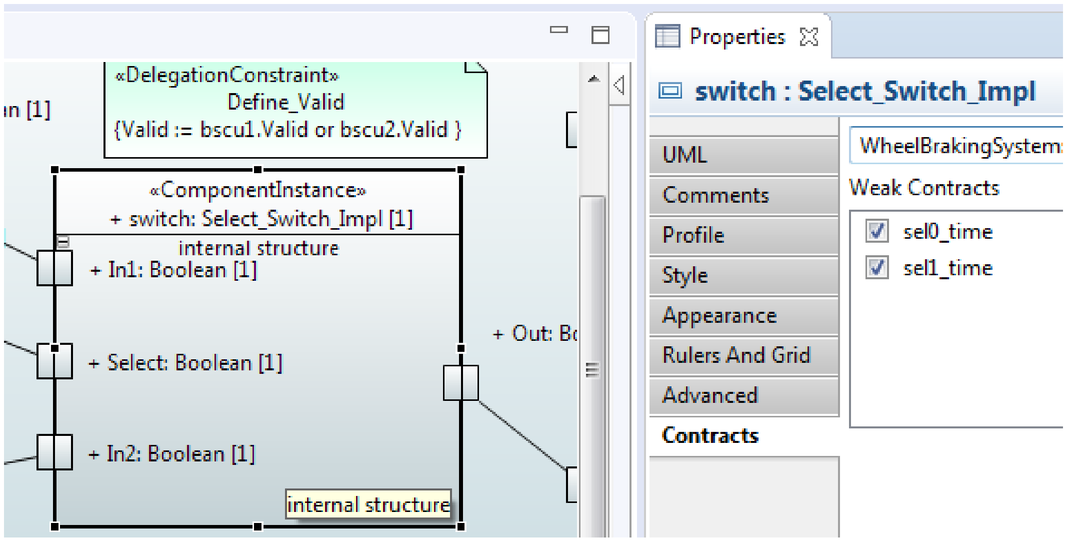
Selected Property

Figure 53. “Contract” tab for instances

The information about the weak contracts which hold for the given instance is automatically set in the “WeakGuarantees” attribute of the “ComponentInstance” stereotype.

Strong and weak contracts are introduced to support out-of-context reasoning and component reuse across variety of environments. While strong contracts must hold in all environments, the weak ones are environment-specific. Prior to performing the refinement check using strong and weak contracts, contracts must be created and allocated to the component types, which represent out-of-context components. At the component type level, the user indicates if a contract is strong or weak. When the component type is instantiated in a particular system to a component instance, all the strong, and a subset of weak contracts can be identified as relevant in the particular system in which the component is instantiated. As discussed previoulsy, identifying which are those relevant weak contracts can be done manually on the component instance level. For example, Figure 101 depicts the selected weak contracts for the Select_Switch_Impl component instance.

To perform the refinement check with strong and weak contracts, the user first creates a class diagram in the “*DependabilityAnalysisView*”, which is a part of the “AnalysisView”. Then, the user creates a new “*ContractRefinementAnalysisContext*” by selecting it from the Palette. She selects the newly created “*ContractRefinementAnalysisContext*” and go to the properties view, in particular the Profile tab. There, the user can select the platform that should be analysed (e.g., the system block), and set the attribute “*checkAllWeakContracts*” to true or false (Figure 98): during the analysis, all weak contracts will be included in the generated .oss file such that they will be transformed to implications within the strong guarantees. To identify which contract is relevant in the particular system context, the user needs to run the command “*CHESS->FunctionalVerification->Automatic Selection of Weak Contract*”. This command will check validity of each weak contract assumption and identify which weak contracts are relevant in the given system. Upon running the weak contract assumption validity check, the contract status is updated accordingly. It should be noted that the "*Automatic Selection of Weak Contract*" command can be run only with discrete-time specification, hence the usage of continuous variables or operators in the contracts disables the validity property check.



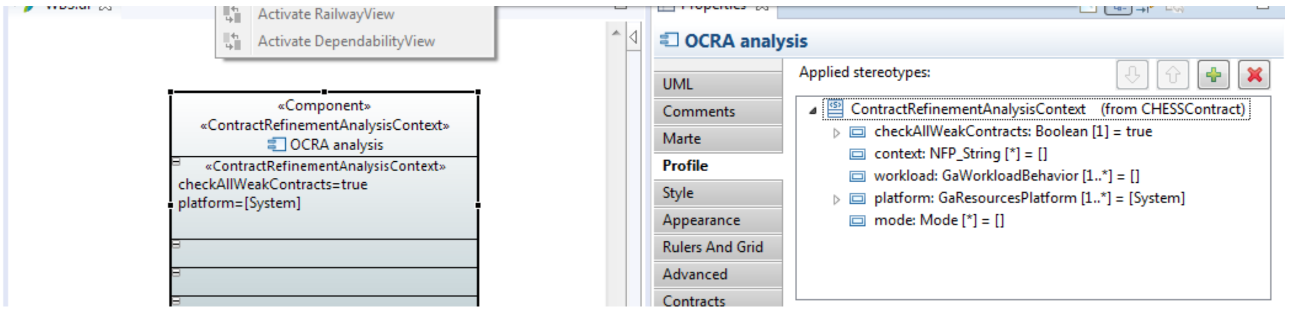
switch : Select_Switch_Impl

WheelBrakingSystem

Weak Contracts

- [✓] sel0_time
- [✓] sel1_time

Figure 54. Weak contract selection for a component instance



OCRA analysis

Applied stereotypes:

```
ContractRefinementAnalysisContext   (from CHESSContract)
  checkAllWeakContracts: Boolean [1] = true
  context: NFP_String [*] = []
  workload: GaWorkloadBehavior [1..*] = []
  platform: GaResourcesPlatform [1..*] = [System]
  mode: Mode [*] = []
```

Figure 55. Aanalysis context for strong and weak contracts

25. Import an OCRA File

CHESS is able to import an architecture described in the OCRA language **Appendix B. OCRA Language to define Formal Properties and Contracts**, contained in a *.oss* file. Components will be generated, along with their structural description (ports, contracts, refinements, connections, etc.) and imported in the selected package. In Section 24 it will be described how to automatically generate diagrams for the imported components.

To import a OCRA file, perform the following steps:

- From the Model Explorer View, select the SystemView package or create a sub package below it.
- Right click on the package, then go to "**CHESS → Basic Operations → Import <<SystemView>> components from .oss file**" (see Figure 52).
- Select a file with *.oss* extension and click "OK".

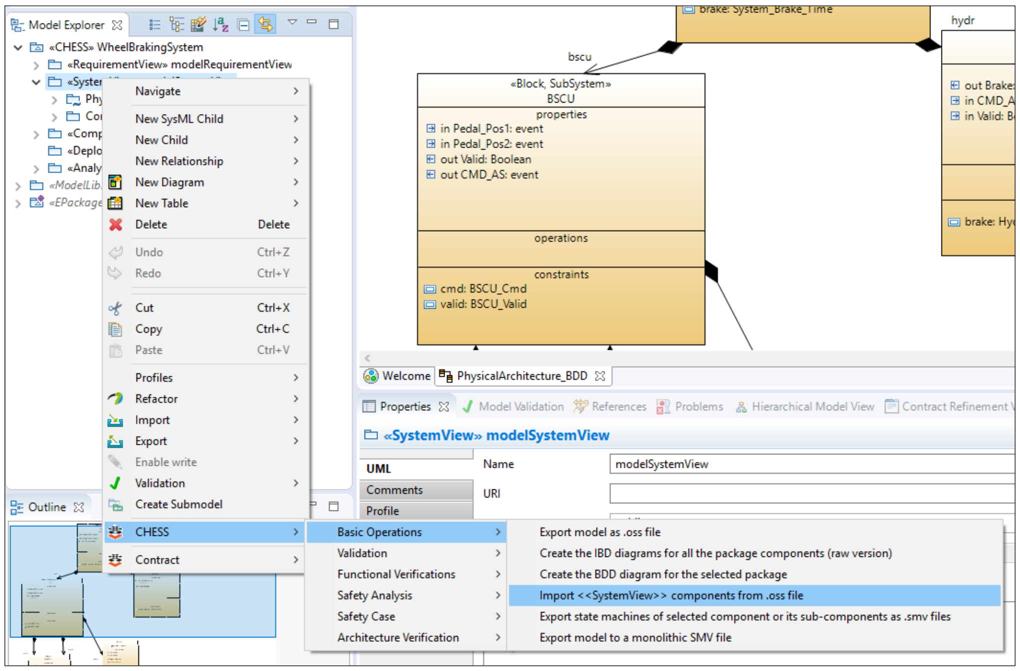

Model Explorer
«CHESS» WheelBrakingSystem
«RequirementView» modelRequirementView
Navigate
New SysML Child
New Child
New Relationship
New Diagram
New Table
Delete
Undo
Redo
Cut
Copy
Paste
Profiles
Refactor
Import
Export
Enable write
Validation
Create Submodel
CHESS
Contract
Basic Operations
Validation
Functional Verifications
Safety Analysis
Safety Case
Architecture Verification
Export model as .oss file
Create the IBD diagrams for all the package components (raw version)
Create the BDD diagram for the selected package
Import <<SystemView>> components from .oss file
Export state machines of selected component or its sub-components as .smv files
Export model to a monolithic SMV file
«Block, SubSystem» BSCU
«SystemView» modelSystemView
Name
modelSystemView

Figure 56. Import of an OCRA file

The import feature allows to reimport an OCRA file: if the selected package already contains some components, they will be updated to reflect the content of the new OCRA definition. BDD and IBD diagrams will be updated accordingly. See Section 24.1 to get more details on diagram updates.

26. Automatic generation of Block Definition and Internal Block Diagrams

The canonical way to add elements to a model is by editing diagrams, e.g., creating a Block Definition Diagram and adding components and relations graphically using the Palette. But it is also possible to edit the model from the Model Explorer View, right clicking and adding children. In the latter case, no diagrams are created or updated. In general, changes in the diagrams are reflected in the model, but changes in the model are not reflected in the diagrams.

CHESS offers the possibility to automatically generate BDDs and IBDs starting from the model.

To create a BDD, perform the following steps:

- Select a package from the Model Explorer View. The package should be the SystemView package or one of its sub packages.
- Right click on the package and go to "**CHESS → Basic Operations → Create the BDD diagram for the selected package**" (see Figure 53).
- A new BDD diagram will be created and added to the model. Graphical components will be automatically arranged.

Similarly, right clicking the package, it is possible to create the IBDs for all the components of that package. To obtain the diagrams, select "**CHESS → Basic Operations → Create the IBD diagrams for all the package**

components" (raw version). Differently from the BDD creation, no automatic arrange will run on the diagrams. See Section 24.2 to run the automatic layout on a specific diagram.

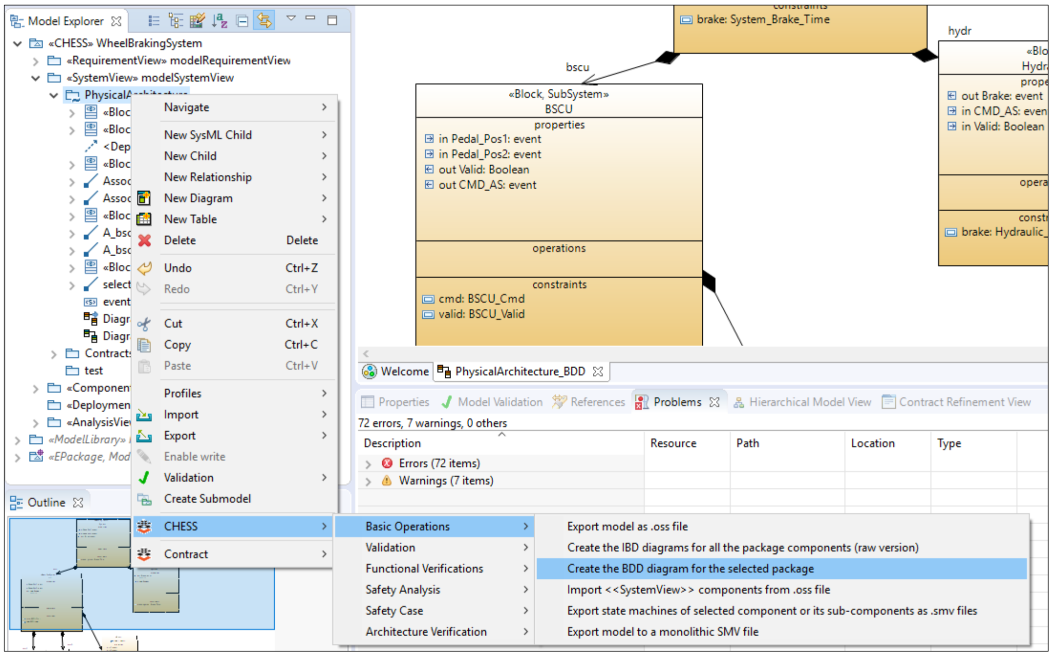


Figure 57. Creation of a BDD from the Model Explorer View

To create a single IBD for a single component, perform the following steps:

- Select the component from the Model Explorer View or the corresponding graphical representation (in the BDD diagram editor).
- Right click on the component and go to "**CHESS → Basic Operations → Create the IBD diagram for the selected component**" (see Figure 54).
- A new IBD diagram will be created and added to the component. Graphical elements will be automatically arranged.

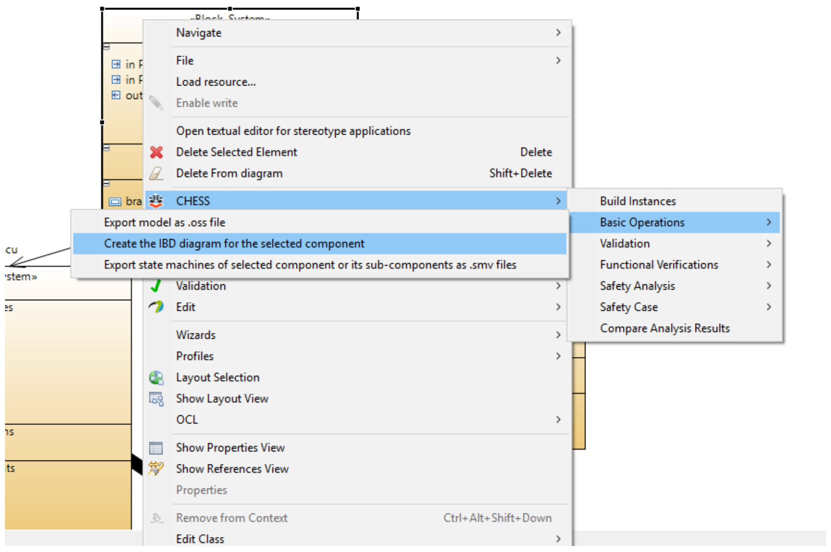

«Block, System»
Navigate
File
Load resource...
Enable write
Open textual editor for stereotype applications
Delete Selected Element Delete
Delete From diagram Shift+Delete
CHESS
Export model as .oss file
Create the IBD diagram for the selected component
Export state machines of selected component or its sub-components as .smv files
Validation
Edit
Wizards
Profiles
Layout Selection
Show Layout View
OCL
Show Properties View
Show References View
Properties
Remove from Context Ctrl+Alt+Shift+Down
Edit Class
Build Instances
Basic Operations
Validation
Functional Verifications
Safety Analysis
Safety Case
Compare Analysis Results

Figure 58. Creation of a IBD from a BDD diagram

It should be noted that multiple diagrams could be created for a single component or package. They do not substitute the already existing diagrams.

26.1 Update of diagrams

As stated before, it could happen that the diagrams and the model are not aligned. As an example, the creation of a port in a BDD is reflected in the model inside the Model Explorer View, but the IBD diagram of that component will not be affected.

To improve the usability, CHESS allows to update BDD and IBD diagrams. Perform the following steps:

- Select the diagram to be updated and click anywhere on the background. Do not select a graphic element.
- Right click and go to "**CHESS → Basic Operations → Update the current BDD (or IBD) diagram**" (see Figure 55).

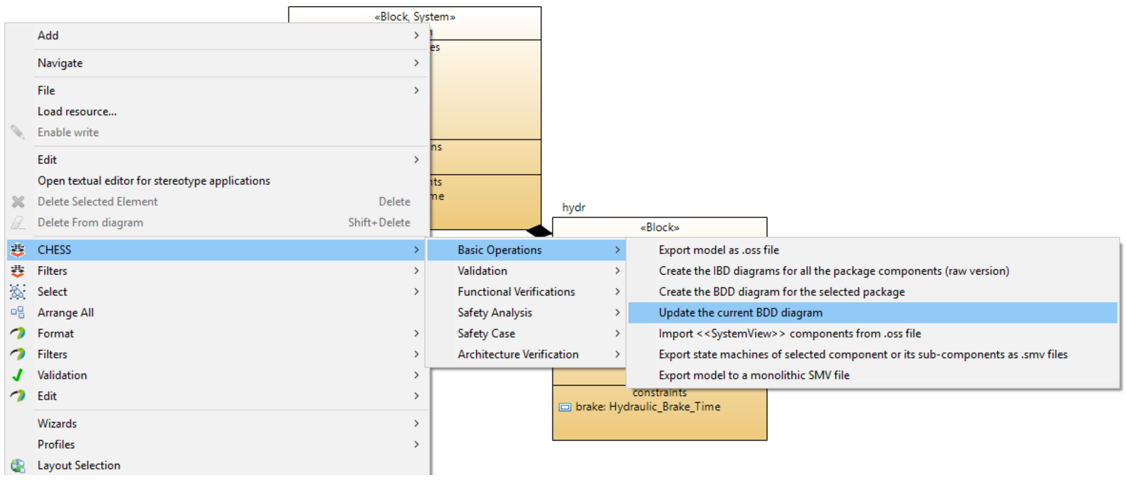

«Block, System»
Add
Navigate
File
Load resource...
Enable write
Edit
Open textual editor for stereotype applications
Delete Selected Element Delete
Delete From diagram Shift+Delete
CHESS
Filters
Select
Arrange All
Format
Filters
Validation
Edit
Wizards
Profiles
Layout Selection
Basic Operations
Validation
Functional Verifications
Safety Analysis
Safety Case
Architecture Verification
hydr
«Block»
Export model as .oss file
Create the IBD diagrams for all the package components (raw version)
Create the BDD diagram for the selected package
Update the current BDD diagram
Import <<SystemView>> components from .oss file
Export state machines of selected component or its sub-components as .smv files
Export model to a monolithic SMV file
constraints
brake: Hydraulic_Brake_Time

Figure 59. Update of a BDD diagram

The diagram will be updated according to the model present in the Model Explorer View. To give emphasis on elements that have been added to the diagrams, they will be depicted in particular positions. New components will be displayed in the upper-left corner while new ports will be displayed in the lower-left corner. The diagram can be later automatically arranged as explained in the next sub section. In Figure 52 it is possible to see an IBD diagram with two new added elements.

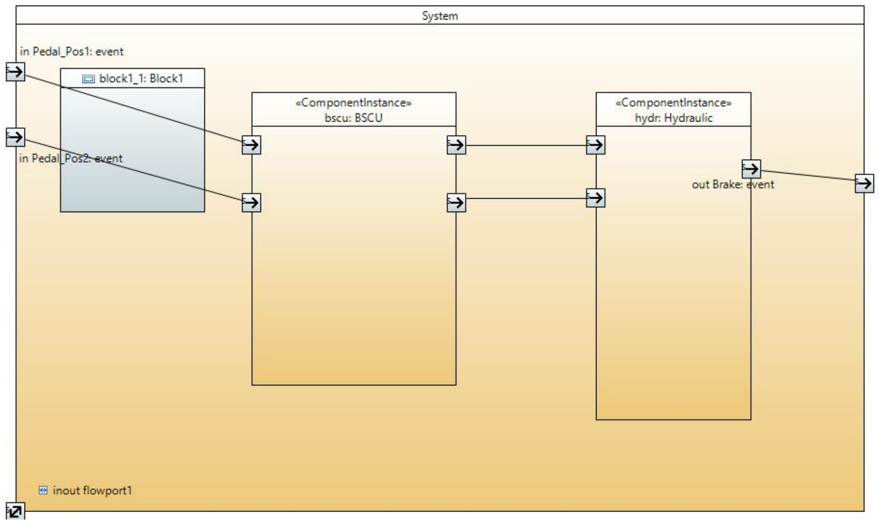


Figure 60. An IBD diagram after the update

26.2 Auto layout of diagrams

When the BDD and IBD diagrams are generated, a routine will automatically rearrange the graphic elements. If a user does not like the layout, he/she can manually move the elements.

There are cases where the user may need to force the auto layout to run again, i.e., after the update of the diagram. It can be done right clicking on the diagram and selecting "**Layout Selection**" or clicking on the icon in the upper-left corner, as seen in Figure 53.

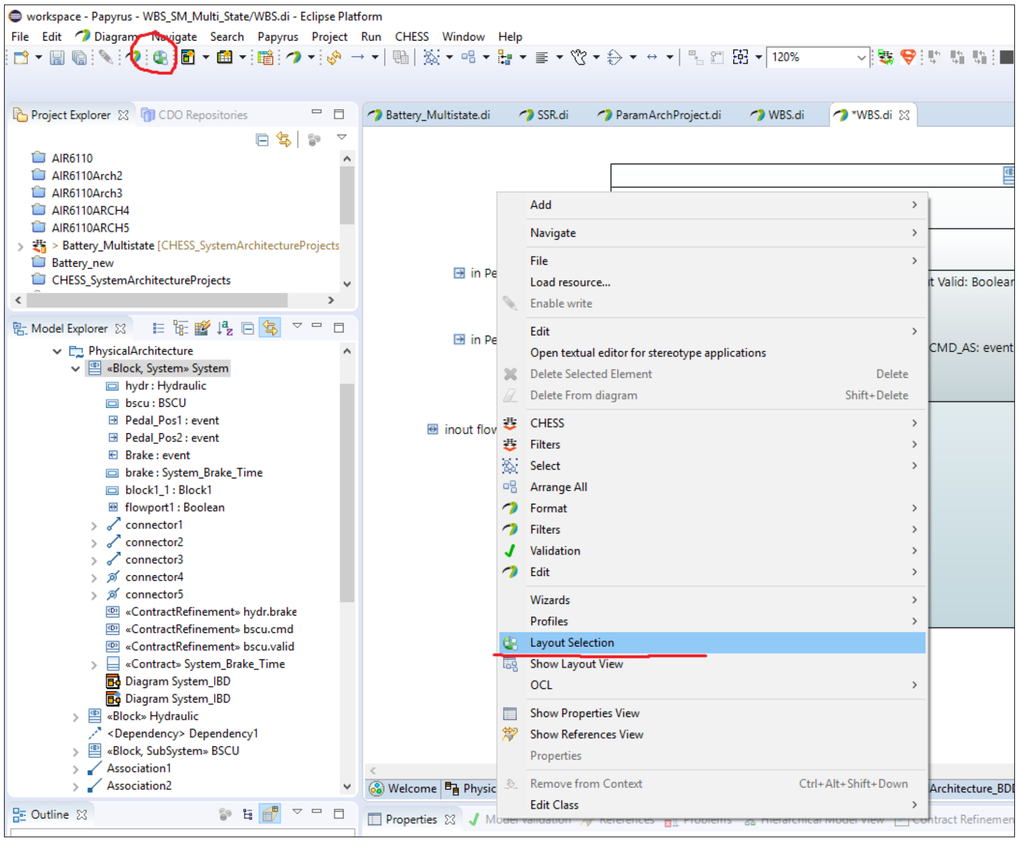


Figure 61. Auto layout of a diagram

27. Generate Documentation

CHESS can generate a document summarizing the model architecture and the results of various analyses executed on the model. Output format is an HTML document or a LaTeX source code.

The generated document is composed by two main sections: *Components* and *V&V Results*; an example can be seen in Figure 58.

The first section describes the structure of the model: it includes the diagrams that are not associated to a specific component, such as the Block Definition Diagram, and a subsection for each of the components. Each subsection contains:

- Name of the component and its type
- Diagrams (Internal Block Diagrams, nominal and error State Machine Diagrams)
- Table of input ports with name and type
- Table of output ports with name and type
- Table of subcomponents with instance name and type name
- Table of interface assertions
- Table of refinement assertions
- Table of connectors between ports
- Table of contracts

- Table of contract refinements
- Table of uninterpreted functions
- Table of parameters
- Table of parameters assumptions

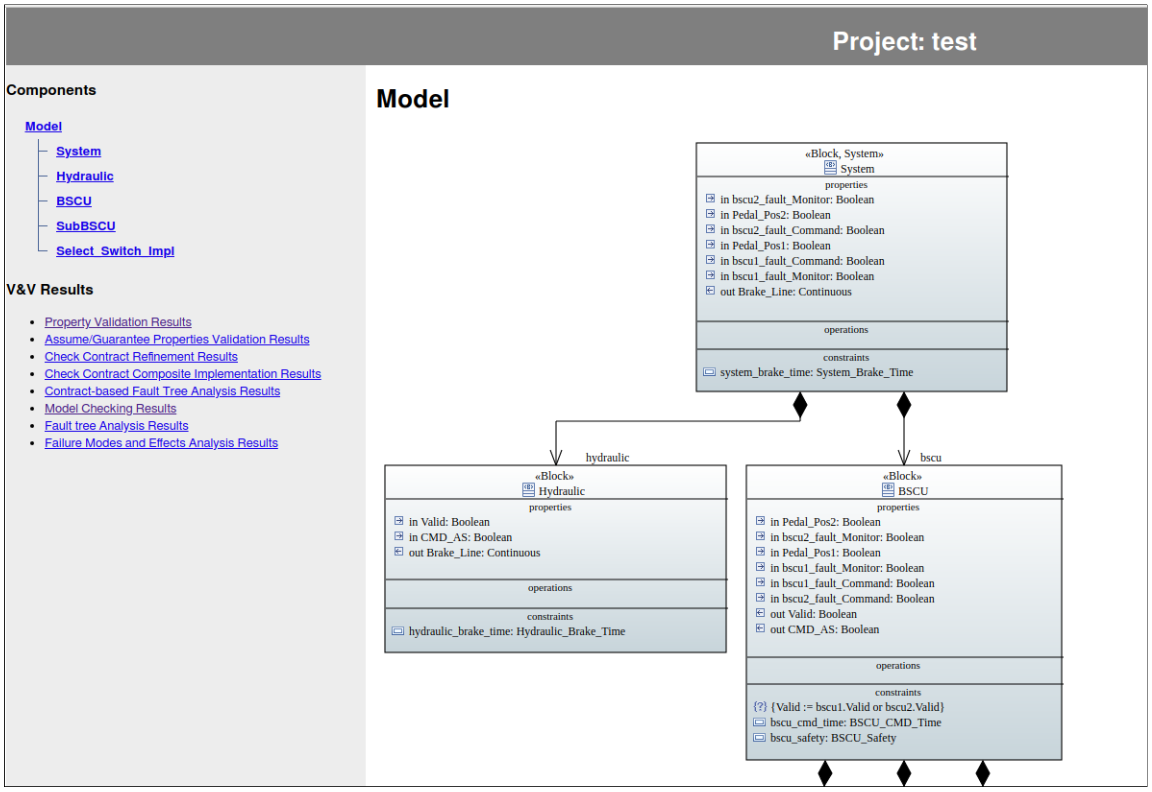
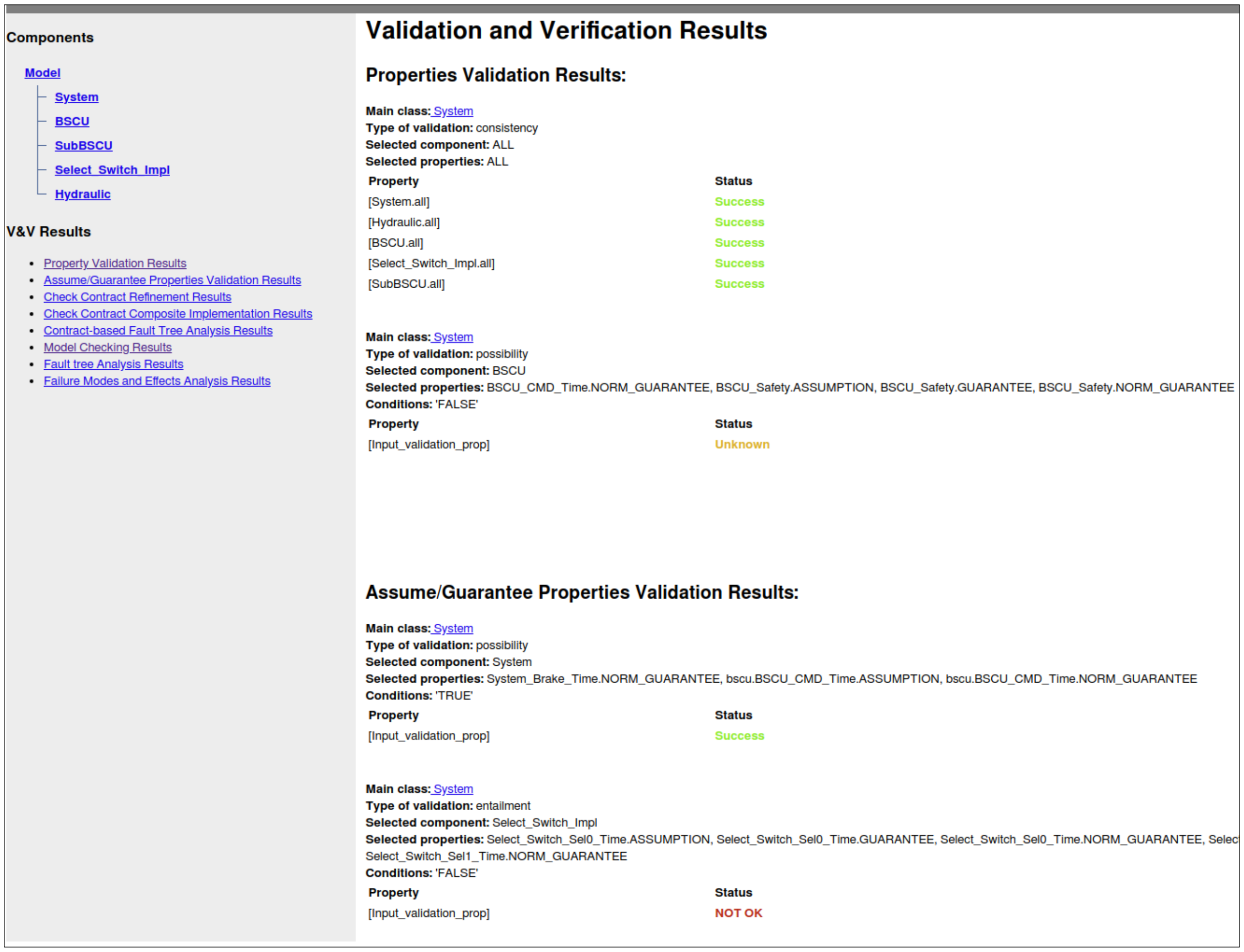

Project: test
Components
Model
System
Hydraulic
BSCU
SubBSCU
Select_Switch_Impl
V&V Results
Property Validation Results
Assume/Guarantee Properties Validation Results
Check Contract Refinement Results
Check Contract Composite Implementation Results
Contract-based Fault Tree Analysis Results
Model Checking Results
Fault tree Analysis Results
Failure Modes and Effects Analysis Results
Model
«Block, System»
System
properties
in bscu2_fault_Monitor: Boolean
in Pedal_Pos2: Boolean
in bscu2_fault_Command: Boolean
in Pedal_Pos1: Boolean
in bscu1_fault_Command: Boolean
in bscu1_fault_Monitor: Boolean
out Brake_Line: Continuous
operations
constraints
system_brake_time: System_Brake_Time
hydraulic
bscu
«Block»
Hydraulic
properties
in Valid: Boolean
in CMD_AS: Boolean
out Brake_Line: Continuous
operations
constraints
hydraulic_brake_time: Hydraulic_Brake_Time
«Block»
BSCU
properties
in Pedal_Pos2: Boolean
in bscu2_fault_Monitor: Boolean
in Pedal_Pos1: Boolean
in bscu1_fault_Monitor: Boolean
in bscu1_fault_Command: Boolean
in bscu2_fault_Command: Boolean
out Valid: Boolean
out CMD_AS: Boolean
operations
constraints
{Valid := bscu1.Valid or bscu2.Valid}
bscu_cmd_time: BSCU_CMD_Time
bscu_safety: BSCU_Safety

Figure 62. Part of a generated report

The second section of the report lists the results of the checks and analyses run on the model. An example can be seen in Figure 59. Reported analyses are the ones executed on the totality of the model, i.e., on the root component (stereotyped with *«System»*). To run an analysis that should be reported in the documentation, it should be run from a specific menu at package level. Details will be explained later in this subsection.

Reported analyses are the following:

- Properties Validation
- Assume/Guarantee Properties Validation
- Check Contract Refinement
- Check Contract Composite Implementation
- Model Checking
- Fault Tree Analysis
- Failure Modes and Effects Analysis



Components

- Model
  - System
  - BSCU
  - SubBSCU
  - Select_Switch_Impl
  - Hydraulic

V&V Results

- Property Validation Results
- Assume/Guarantee Properties Validation Results
- Check Contract Refinement Results
- Check Contract Composite Implementation Results
- Contract-based Fault Tree Analysis Results
- Model Checking Results
- Fault tree Analysis Results
- Failure Modes and Effects Analysis Results

Validation and Verification Results

Properties Validation Results:

Main class: System
Type of validation: consistency
Selected component: ALL
Selected properties: ALL

Property	Status
[System.all]	Success
[Hydraulic.all]	Success
[BSCU.all]	Success
[Select_Switch_Impl.all]	Success
[SubBSCU.all]	Success

Main class: System
Type of validation: possibility
Selected component: BSCU
Selected properties: BSCU_CMD_Time.NORM_GUARANTEE, BSCU_Safety.ASSUMPTION, BSCU_Safety.GUARANTEE, BSCU_Safety.NORM_GUARANTEE
Conditions: 'FALSE'

Property	Status
[Input_validation_prop]	Unknown

Assume/Guarantee Properties Validation Results:

Main class: System
Type of validation: possibility
Selected component: System
Selected properties: System_Brake_Time.NORM_GUARANTEE, bscu.BSCU_CMD_Time.ASSUMPTION, bscu.BSCU_CMD_Time.NORM_GUARANTEE
Conditions: 'TRUE'

Property	Status
[Input_validation_prop]	Success

Main class: System
Type of validation: entailment
Selected component: Select_Switch_Impl
Selected properties: Select_Switch_Sel0_Time.ASSUMPTION, Select_Switch_Sel0_Time.GUARANTEE, Select_Switch_Sel0_Time.NORM_GUARANTEE, Selec
Select_Switch_Sel1_Time.NORM_GUARANTEE
Conditions: 'FALSE'

Property	Status
[Input_validation_prop]	NOT OK

Figure 63. Analysis results section of the generated report

To automatically generate the documentation, perform the following steps:

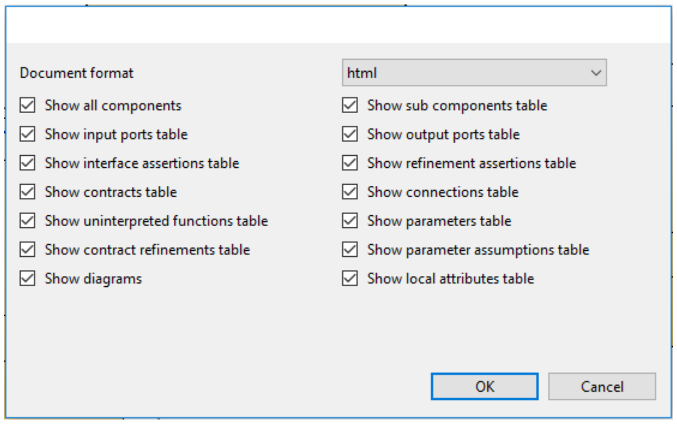
- Select the root component (in the Model Explorer View) or the corresponding graphical representation (in the Diagram editor). It is also possible to select the whole package in the Model Explorer View. The information used to generate the documentation will be related to the selected component and to its sub components. This operation includes recursively the information from the root to the leaves of the selected component.
- Right click on the selected component, then go to "**CHESS → Safety Case → Document Generation → Generate documentation on selected component**".
- A popup appears to select the folder that will contain the document and the diagrams.
- A popup appears to set the options related to the format of the document and to the style of the diagrams (see Figure 60).
- A popup will appear to select the time model of the architecture. Options supported are *discrete, hybrid* and *timed*.

Analyses that should be reported in the documentation, i.e., executed on the root component, are grouped in a specific menu. To execute them, perform the following steps:

- Select the package containing the architecture from the Model Explorer View.

- Right click and go to "**CHESS ->Architecture Verification**".
- Select one of the available analyses to be run on the root component of that package.

It is possible to select any component from the model and generate the report. In this case, no results from the root-level analyses will be reported in the documentation as they are valid for the whole model only. Analyses executed on subcomponents will not be reported either.



Document format: html

- [✓] Show all components
- [✓] Show sub components table
- [✓] Show input ports table
- [✓] Show output ports table
- [✓] Show interface assertions table
- [✓] Show refinement assertions table
- [✓] Show contracts table
- [✓] Show connections table
- [✓] Show uninterpreted functions table
- [✓] Show parameters table
- [✓] Show contract refinements table
- [✓] Show parameter assumptions table
- [✓] Show diagrams
- [✓] Show local attributes table

Figure 64. Popup to set the preferences of the generated document

Appendix A. CHESS Supported Basic Types

The basic types that can be assigned to ports, local attributes, parameters, functions behavior are:

- Primitive types: the following types are available on the package “UML Primitive types”
  - Integer
  - Boolean
  - Real
- Continuous: this type is available on the package “CHESS Contract - DataTypes”. If this type is not available, in the Model Explorer View, select the model – right click – import – import registered profile, and select “CHESS Contract”. Continuous type is available only in the **hybrid** time domain.
- IdealClock: this type is available in the package “MARTE_Library - TimeLibrary”. If the type is not available, in the Model Explorer View, select the model – right click – import – import registered package, and select “MARTE_Library”. IdealClock type is available only in the **timed** time domain.
- Range: To create this type:
  - In the Model Explorer View, select the owner of the type to define.
  - Right click – New Child – DataType
  - Select the new DataType, in the Property View – Profile tab, Add Profile – BoundedSubtype
  - in the Property View – Profile tab, select the BoundedSubtype and set the minValue and maxValue
- Enumerative: To create this type:
  - In the Model Explorer View, select the owner of the type to define.
  - Right click – New Child – Enumeration
  - Select the new DataType, in the Property View – UML tab – Owned literals, add the enumerative values
- Event
  - In the Model Explorer View, select the owner of the type to define
  - Right click – New Child – Signal

Appendix B. OCRA Language to define Formal Properties and Contracts

OCRA constraint language can be interpreted over discrete traces (in this case, it coincides with LTL) or over hybrid traces. The relevant syntax of the language has been summarized in Figure 61 together with the corresponding mathematical formulation in HRELTL.

constraint	:=	*atom* |	ϕ	:=	a |
		not *constraint* |			$\neg\phi$ |
		constraint **and** *constraint* |			$\phi \wedge \phi$ |
		constraint **or** *constraint* |			$\phi \vee \phi$ |
		constraint **implies** *constraint* |			$\phi \rightarrow \phi$ |
		always *constraint* |			$G\phi$ |
		never *constraint* |			$G\neg\phi$ |
		in the future *constraint* |			$F\phi$ |
		constraint **until** *constraint*;			$\phi\, U\phi$;
		then *constraint* |			$X\phi$ |
		historically *constraint* |			$H\phi$ |
		in the past *constraint* |			$P\phi$ |
		constraint **since** *constraint*;			$\phi\, S\phi$;
		term **at next** *constraint*;			$t\ @F\phi$;
		term **at last** *constraint*;			$t\ @P\phi$;
atom	:=	**true** |	a	:=	$\top$ |
		false |			$\bot$ |
		term relation term |			$t \bowtie t$ |
		time_until(*term* **)** *relation term* |			$\rhd_{\bowtie t} t$ |
		time_since(*term* **)** *relation term* |			$\lhd_{\bowtie t} t$ |
		change(*term* **)** |			$v' \neq v$ |
		fall(*boolean_term* **)** |			$\phi \wedge \neg X\phi$ |
		rise(*boolean_term* **)** |			$\neg\phi \wedge X\phi$ |
		boolean_term ;			t;
term	:=	*port* |	t	:=	v |
		constant |			c |
		term function term |			$t \star t$ |
		der(*port* **)** |			$\dot{v}$ |
		next(*port* **)** ;			v';

Figure 65. The OCRA language grammar

In case of discrete time, *der*, *time_until*, and *time_since* are not allowed.

Basic formulas are defined with linear arithmetic predicates over the variables or their derivatives. For examples, *x-e<limit* and *der(x)<0* are well-defined formulas. Predicates can be combined with Boolean and temporal operators. For example, *x-e<limit* and *der(x)<0* and *always x-e<limit* are well-defined formulas.

In temporal logic, a formula without temporal operators is interpreted in the initial state. Thus, *x=0* characterizes all traces that start with a state evaluating x to 0, and then x can evolve arbitrarily. Instead, to express that a predicate holds along the whole evolution, one may use the *always* operator as in *always x=0*.

Another classical example of properties is the response to a certain event. The formula *always (p implies in the future q)* defines the set of traces where every occurrence of *p* is followed by an occurrence of *q*. Note that *q* may happen with a certain delay (although there is no bound on such delay). The formula always (p implies *q*) instead forces *q* to happen at the instant of *p*.

The above formulas do not constrain the time model of the traces. Therefore, they can be interpreted either as discrete traces or as hybrid traces. However, the logic is suitable to characterize specific sets of hybrid traces, constraining when there should be discrete events and how the continuous variables should evolve along continuous evolutions.

The *der(.)* operator is used to specify constraints on the derivative of the continuous evolution of continuous variables. For example, the following OCRA constraint:

always (train.location<=target implies der(train.location)>=0)

characterizes the set of hybrid traces where in all states, if the train has not yet reached the target location, its speed (expressed as the derivative of the location) is greater than or equal to zero.

The *next(.)* operator is used to specify functional properties requiring discrete changes to variables. For example, we can express the property that the warning variable will change value after the train's speed passes the limit with the following constraint:

always (speed>limit implies in the future next(warning)!=warning)

The expression *change(x)* can be used instead of *next(x)!=x*.

The expression *fall(x)* is an abbreviation for *x and then(not x)*, i.e. a Boolean term *x* becomes false.

The expression *rise(x)* is an abbreviation for *(not x) and then x*, i.e. a Boolean term *x* becomes true.

In order to constrain the delay between two events, we use the *time_until(.)* and *time_since(.)* operators, which denote respectively the time that will elapse until the next occurrence of an event and the time that elapsed since the last occurrence of an event. For example, the formula *always (p implies time_until(q)<max_delay)* defines the set of hybrid traces where *p* is always followed by *q in less than max_delay time units*.

The operator *at next* has a similar but more general purpose. It denotes the value of the left expression, known as the sample, at the next step in which the right expression, known as the trigger, will be true. For example, the formula *always (p implies (((time at next q)- time)< max_delay))* is the discrete time equivalent of the previous example, using an explicit user defined time variable. There is also an *at last.* operator, which denotes the value of the sample at the last step in which the trigger was true.

For further information see OCRA user manual[7].

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

Appendix C. Note about the usage of the <<Contract>> ConstraintBlock in the context of the SysML language

The SysML ConstraintBlock is a package for Constraints that can be used to bind the latter to a given block. One of the main features of the ConstraintBlock construct is that it allows the reuse of constraints in different blocks; in particular this is possible in SysML:

1. by using the ConstraintBlock parameters to represent the parameters addressed by the packaged constraints,
2. by using the Parametric diagram which allows to bind the ConstraintBlock parameters to the Block parameters, so as to specify which are the Block parameters that are interested by the constraints packaged in the ConstraintBlock.

In FoReVer the aforementioned SysML support for ConstraintBlock can be applied to Contracts also, so the design and then the reuse of pattern of Contract are available.

However, in case the support of Contract-pattern is not needed, the SysML procedure regarding the binding of the ConstraintBlock to a given Block can be simplified by allowing the modeler to omit the parameters of the Contract itself and so to omit the modeling of the Parametric diagram for the given Contract usage.

So the following semantic regarding the binding of Contract to Block is adopted in the FoReVer profile:

- If the Parametric diagram is not provided for the given instantiation of the Contract inside the Block, then an implicit binding is assumed between the parameters considered in the Assume and Guarantee of the Contract and the properties (e.g. ports) of the Block where the Contract has been instantiated. The implicit mapping is performed by comparing the string of the Assume/ Guarantee parameters with the name of the Block properties.
- If instead the Parametric diagram is provided for the given instantiation of the Contract inside the Block, then the mapping of the Assume/ Guarantee parameters to the Block properties are derived from the mapping provided through the Parametric diagram itself.

Appendix D. CleanC Language, the imperative language to define transition guards and effects.

CleanC is a sub-set of the C language that is enriched with additional rules.

Each transition guard can be expressed in CleanC with a boolean expression (e.g. 'port1 < 5').

Each transition effect can be expressed in CleanC with a sequence of assignments (e.g. 'port1=5; port2=true;').

CleanC rules:

- Subset of C:
  - Supported types: boolean, integer, real, enum.
  - Assignment Statement, the supported operator is '='.

- Logical Expressions.
  - The supported binary operators are: '||', '&&', '==', '!='.
  - The supported unary operators is: '!'.
- Arithmetic Expressions, the supported operators are: '*', '/', '%', '+', '-'.
- Relational Expressions, the supported operators are: '<', '>', '<=', '>='.
- Bitwise Expressions.
  - The supported binary operators are '&','|', '^','<<', '>>'.
  - The supported unary operators is: '~'.

- Additional rules:
  - Boolean Literals: 'true','false'.
  - Specific Function Call Expressions, the supported operation calls are abs(___), count(___, ...), min(___,___), max(___,___).