ModelicaML – Tutorial

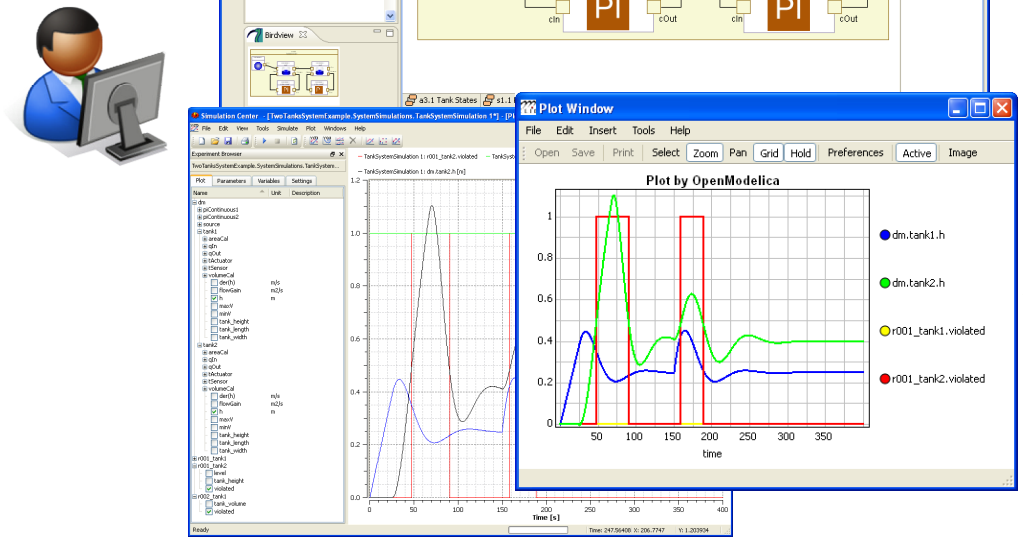
Getting Started

Wladimir Schamai
EADS Innovation Works
Systems Engineering

20.03.2011

ModelicaML: Technology

1 System Modeling with ModelicaML



2 Modelica Code Generation

```
within TwoTanks;

function limitValue

  input Real pMin;
  input Real pMax;
  output Real pLim;
  input Real p;

  algorithm
  // code generated from the Activity "algorithm (diagram)"
  // Activity.name: "algorithm (diagram)"
    if p < pMin then
      pLim := pMin; // OpaqueAction.name: "pLim := pMin;"
    elseif p > pMax then
```

```
within DesignModels;

model Tank

  TwoTanksSystemExample.DesignModels.ReadSignal tSensor; //Connector, sensor reading tank le
  TwoTanksSystemExample.DesignModels.ActSignal tActuator; //Connector, actuator controlling
  TwoTanksSystemExample.DesignModels.LiquidFlow qIn; //Connector, flow (m3/s) through input
  TwoTanksSystemExample.DesignModels.LiquidFlow qOut; //Connector, flow (m3/s) through outpu
  parameter Real flowGain(unit = "m2/s") = 0.05;
  parameter Real minV = 0; //Limits for output valve flow
  parameter Real maxV = 10; //Limits for output valve flow
  Real h(start = 0.0, unit = "m"); //Tank level
  parameter Real tank_height = 0.6;
  parameter Real tank_width = 1;
  parameter Real tank_length = 1.3;
```

```
within DesignModels;

model TanksConnectedPI

  TwoTanksSystemExample.DesignModels.LiquidSource source(flowLevel = 0.02);
  TwoTanksSystemExample.DesignModels.Tank tank1;
  TwoTanksSystemExample.DesignModels.Tank tank2(tank_length = 1);
  TwoTanksSystemExample.DesignModels.PIcontinuousController piContinuous1(ref = 0.25);
  TwoTanksSystemExample.DesignModels.PIcontinuousController piContinuous2(ref = 0.4);

  equation
    connect(source.qOut, tank1.qIn);
    connect(piContinuous1.cIn, tank1.tSensor);
    connect(piContinuous1.cOut, tank1.tActuator);
    connect(tank1.qOut, tank2.qIn);
    connect(piContinuous2.cIn, tank2.tSensor);
    connect(piContinuous2.cOut, tank2.tActuator);

end TanksConnectedPI;
```

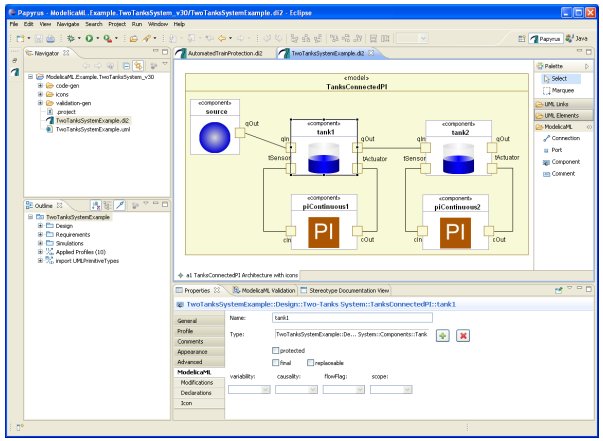
3 System Simulation with Modelica Tools

EADS

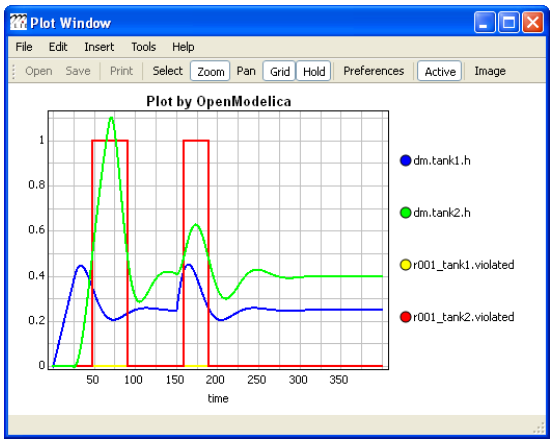
ModelicaML: Technology

Papyrus UML

ModelicaML Profile (Eclipse Plug-In)



Any Modelica Simulation Tool



Model to Text Transformation

Acceleo

ModelicaML Code Generator
(Eclipse Plug-In)

.mo

Generated Modelica Code
(.mo files)

Required Installations

- If not installed yet: JavaSE-1.6 from http://www.oracle.com/technetwork/java/javase/overview/index.html, required for Eclipse.
- Eclipse 3.6 (Helios) Modeling from http://www.eclipse.org/downloads/
- Papyrus MDT Eclipse plug-in (find the update site on http://www.eclipse.org/modeling/mdt/papyrus/updates/index.php) for modeling with UML
- Acceleo Eclipse plug-in from http://www.acceleo.org/pages/download-acceleo/en for Modelica code generation from ModelicaML models
- ModelicaML Eclipse plug-ins from http://www.openmodelica.org/index.php/developer/tools/134 for modeling with ModelicaML (ModelicaML plug-ins that are available for Papyrus MDT)
- OMC from http://www.openmodelica.org/index.php/download. OpenModelica Compiler used for simulation.
- Modelica Development Tooling (MDT) Eclipse plug-in (find the update site on http://www.openmodelica.org/index.php/developer/tools/133) for viewing the generated Modelica code if needed

For details see: www.openmodelica.org/modelicaml

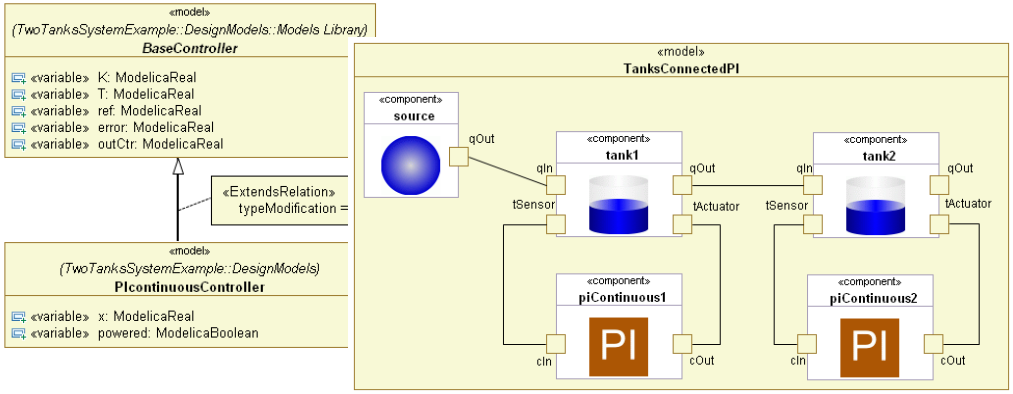
ModelicaML

General Description of the

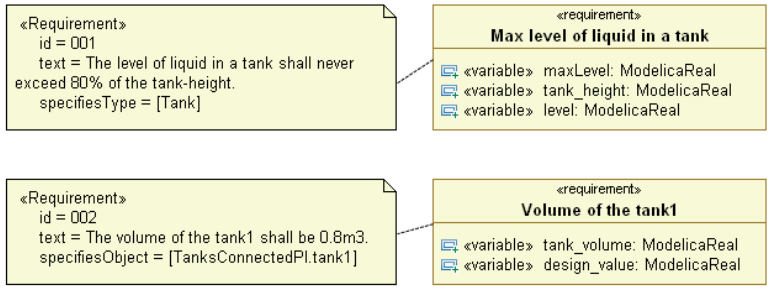
UML-Based Graphical Notation

ModelicaML: Graphical Notation

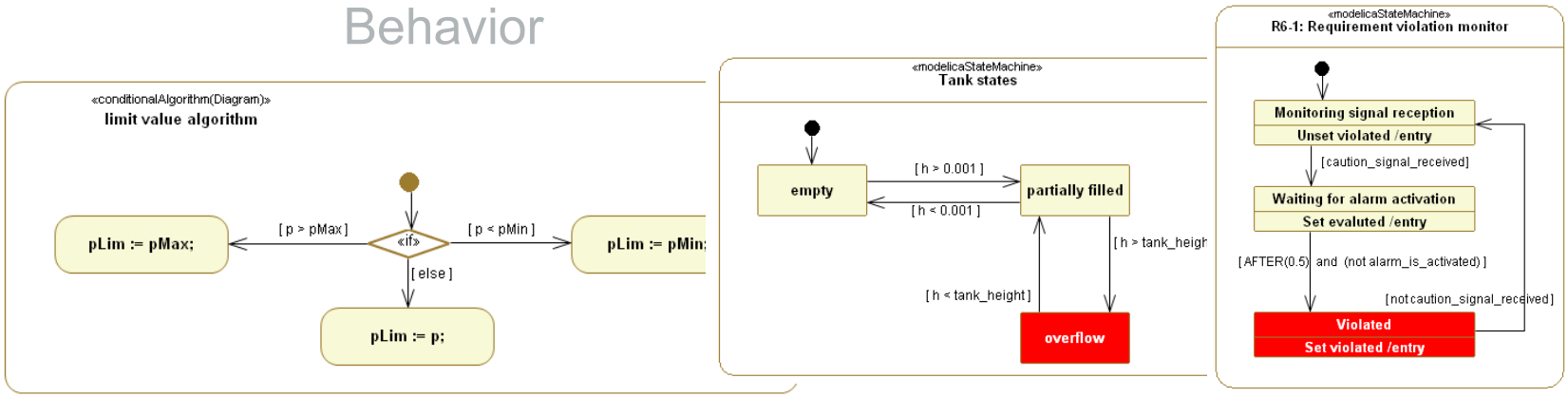
Structure



Requirements



Behavior



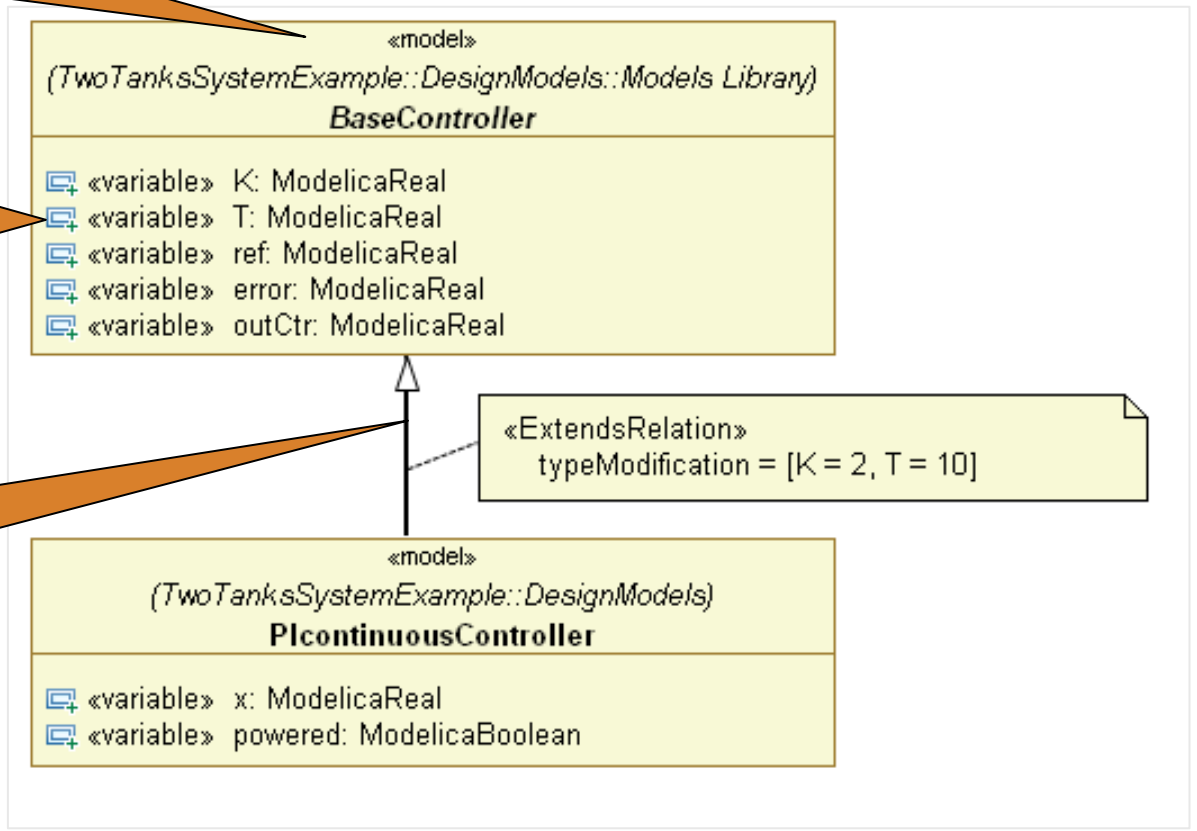
EADS

ModelicaML: Class Diagram

Class (model, block, record, connector)

variables (of primitive type) or **components** (of composite type)

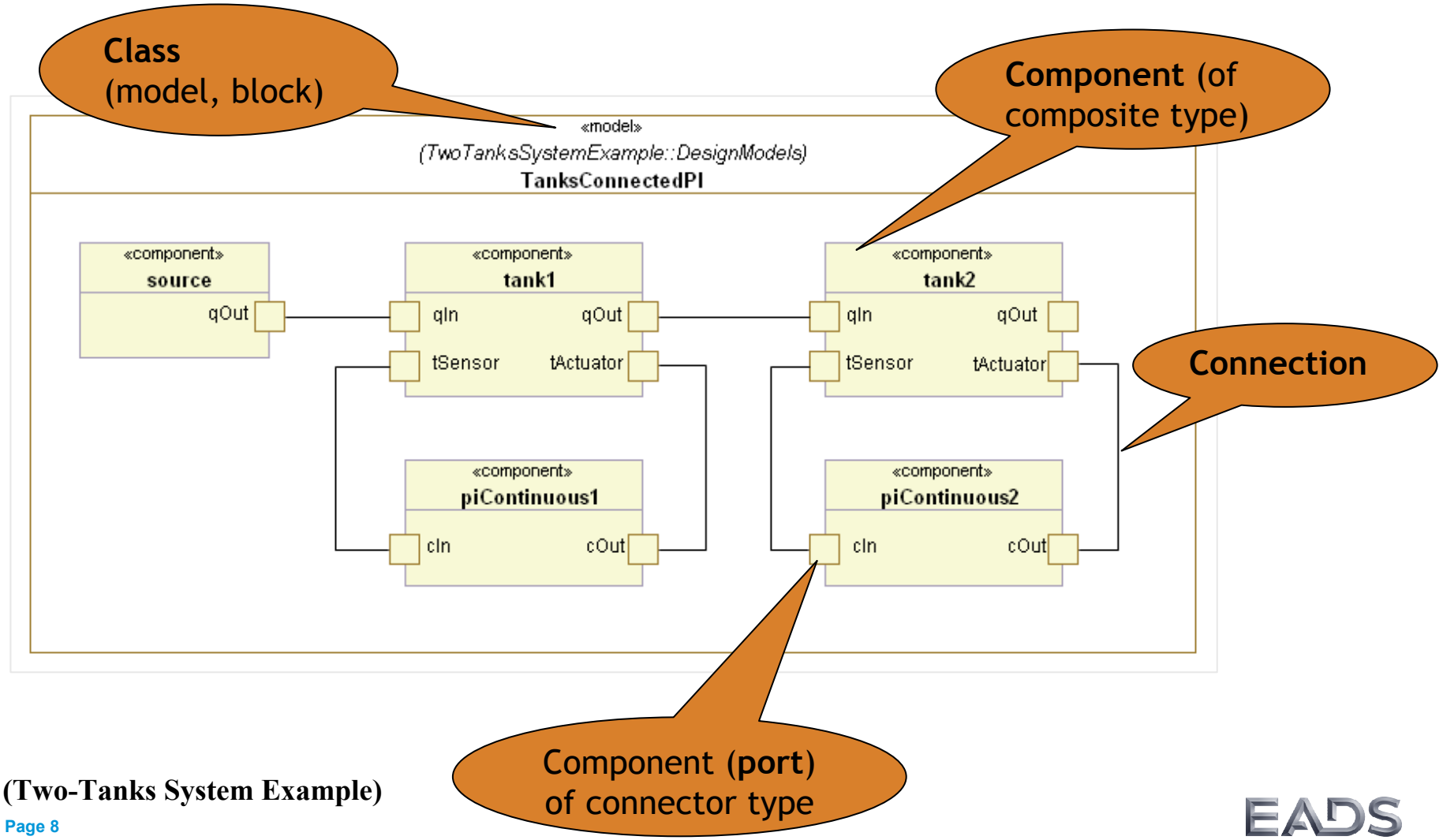
extends relation (with type modifications)



(Two-Tanks System Example)

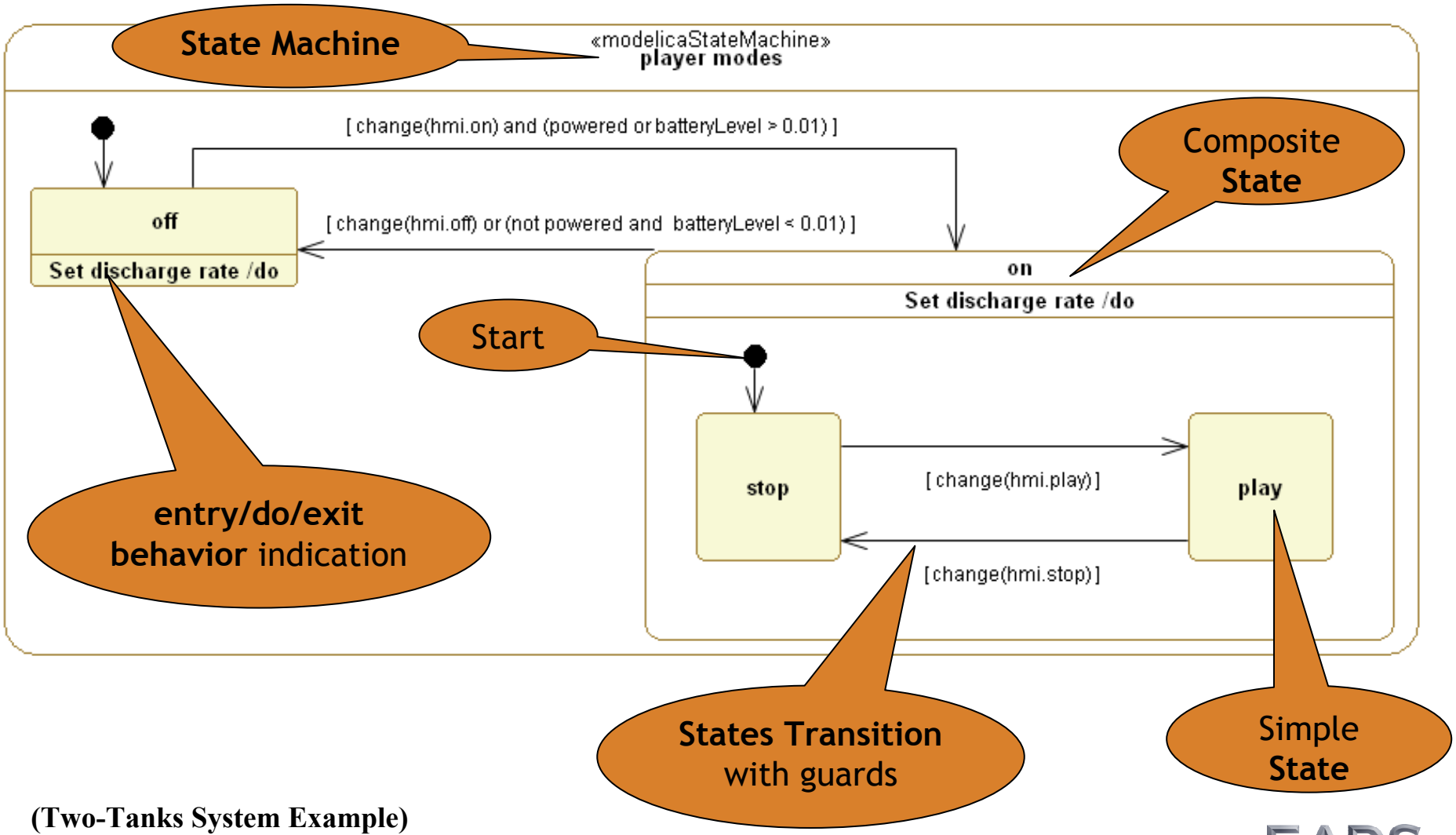
EADS

ModelicaML: Connection Diagram



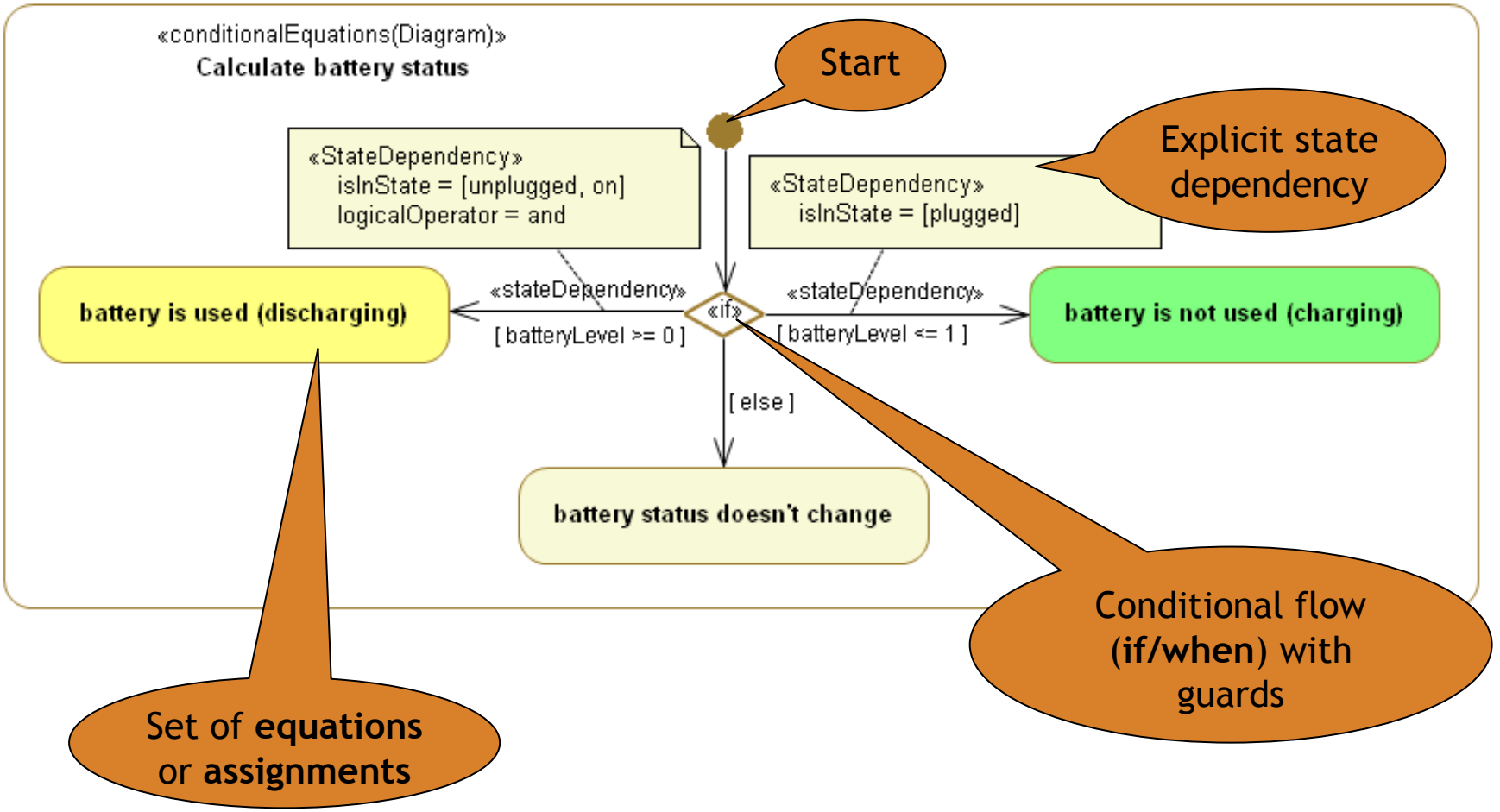
(Two-Tanks System Example)

ModelicaML: State Machine Diagram



(Two-Tanks System Example)

ModelicaML: Conditional Eq./Alg. Diagram



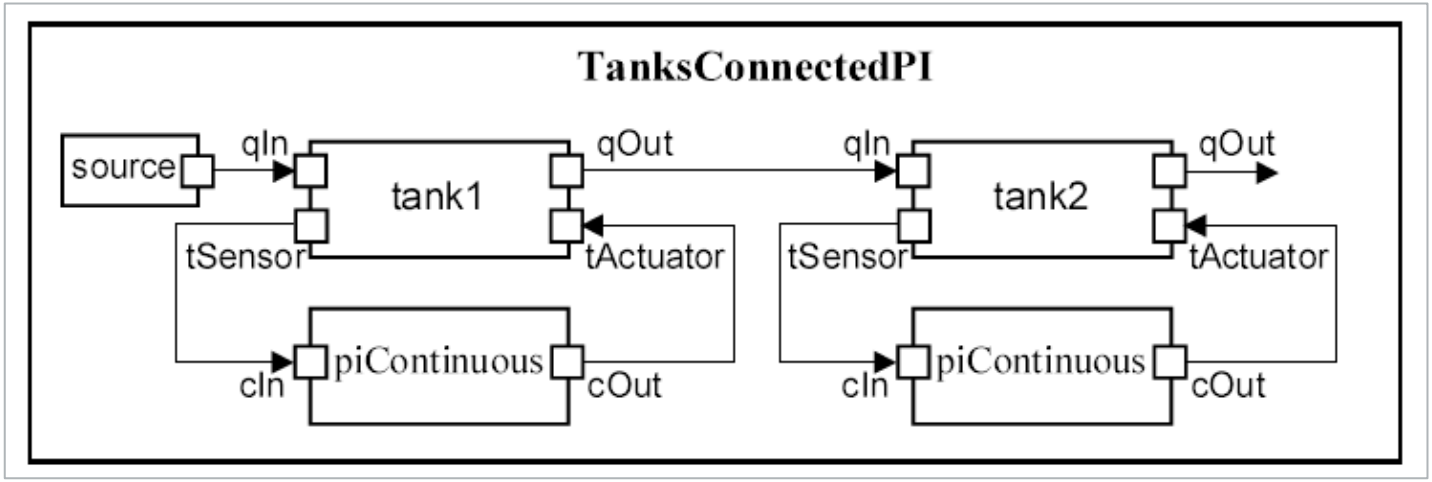
(Player Example)

ModelicaML

Hands-on Modeling Tutorial

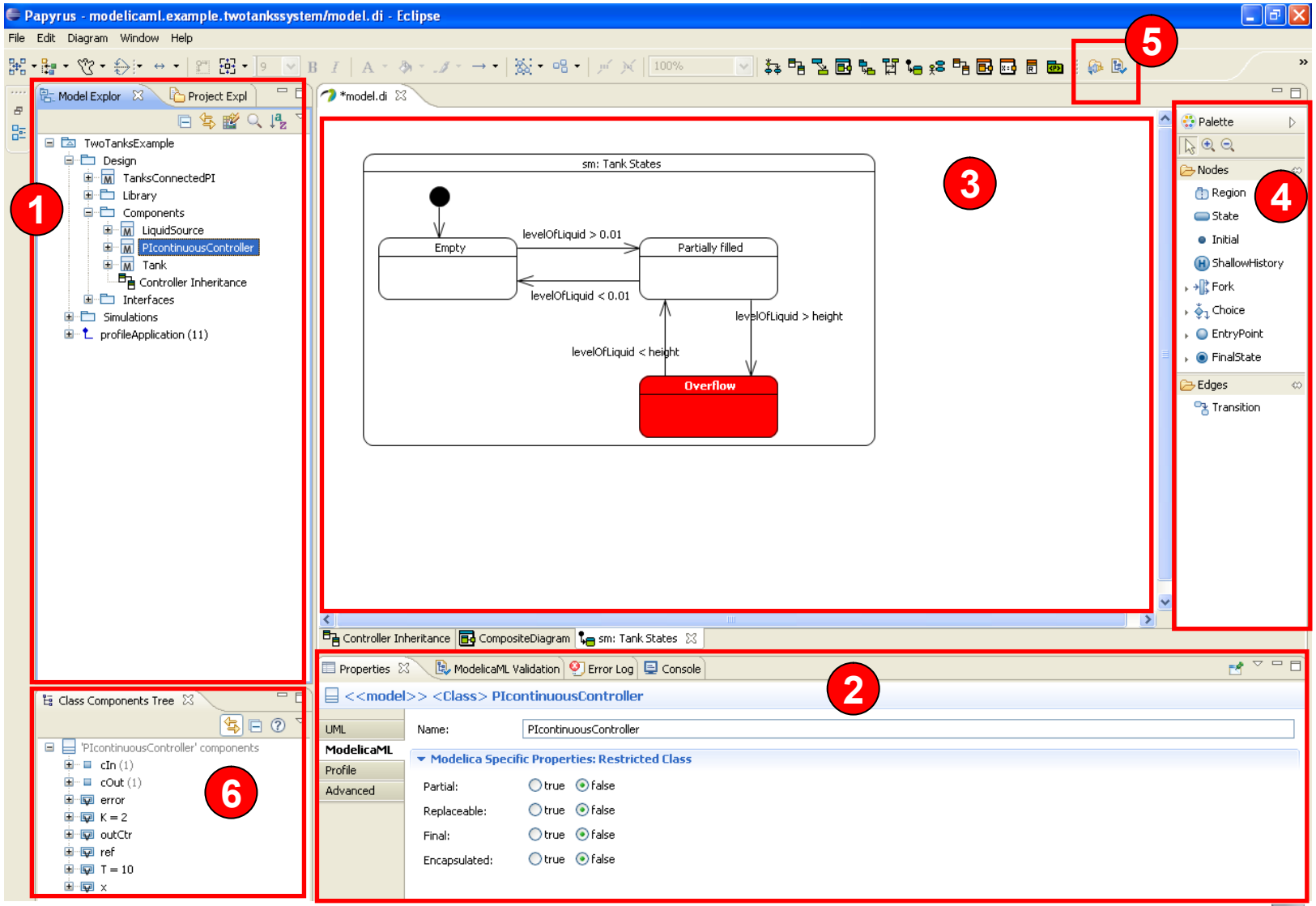
Example: Two Tanks System

- From “source” liquid flows into the “tank1”
- Controller “piContinuous1” controls the level of liquid in “tank1”, based on a predefined reference value, by opening and closing the tank outflow valve.
- Liquid flows from “tank1” into “tank2”
- Controller “piContinuous2” controls the level of liquid in “tank2”



Source: Peter Fritzson. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley-IEEE Press, 2004. page 391.

ModelicaML Papyrus MDT GUI Overview



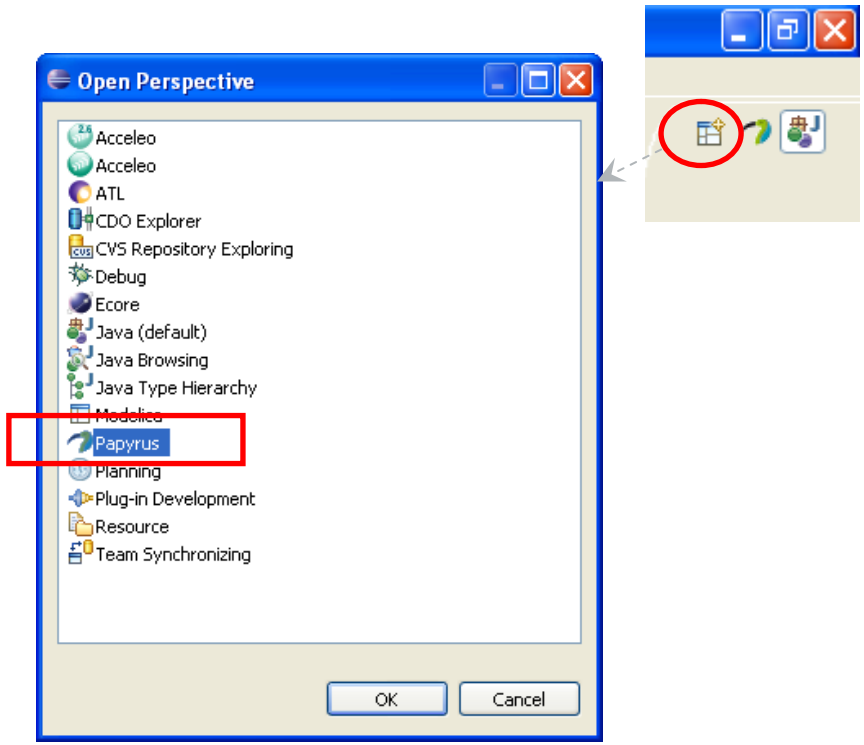
ModelicaML Papyrus MDT GUI Overview

1. Model Browser: Shows model elements
2. Properties View: Shows the properties of selected element
3. Diagram Editors (different UML-based diagrams)
4. Palette (different for each diagram)
5. ModelicaML code generation and validation buttons
6. Component tree: Shows the components hierarchy of the selected class

ModelicaML Project Setup

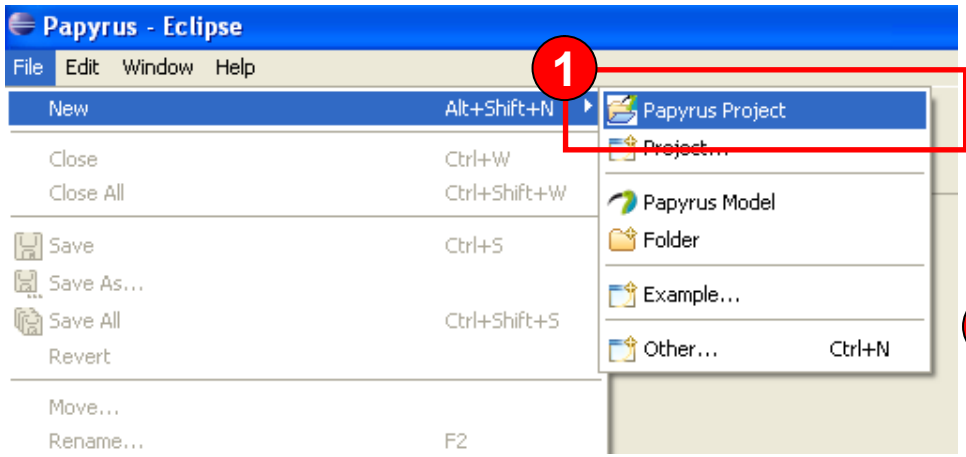
Create ModelicaML Project

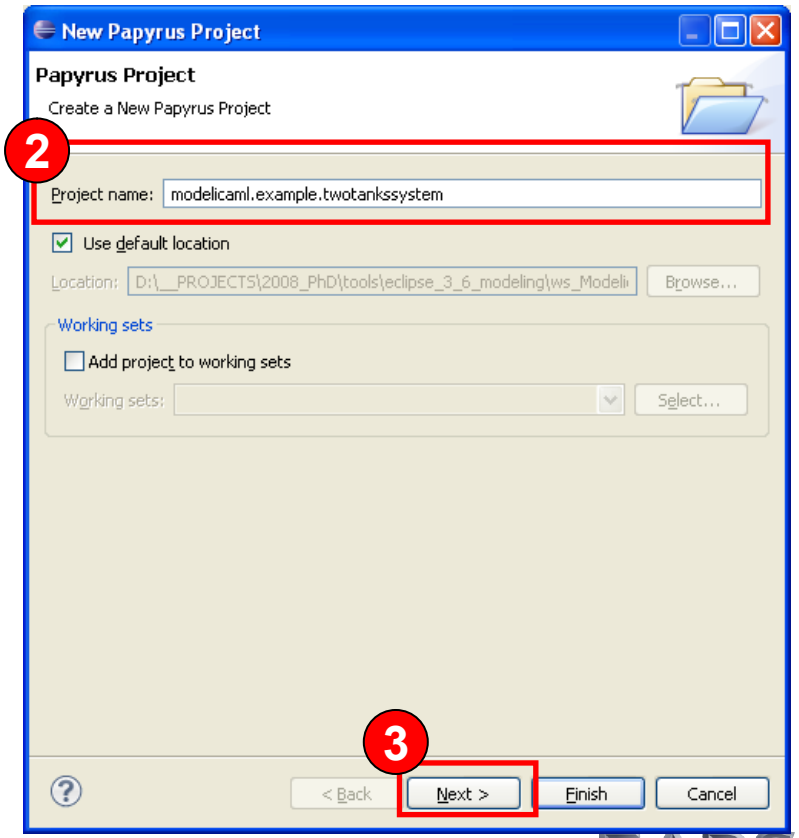
- Open Eclipse
- Change the Perspective to Papyrus Perspective



EADS

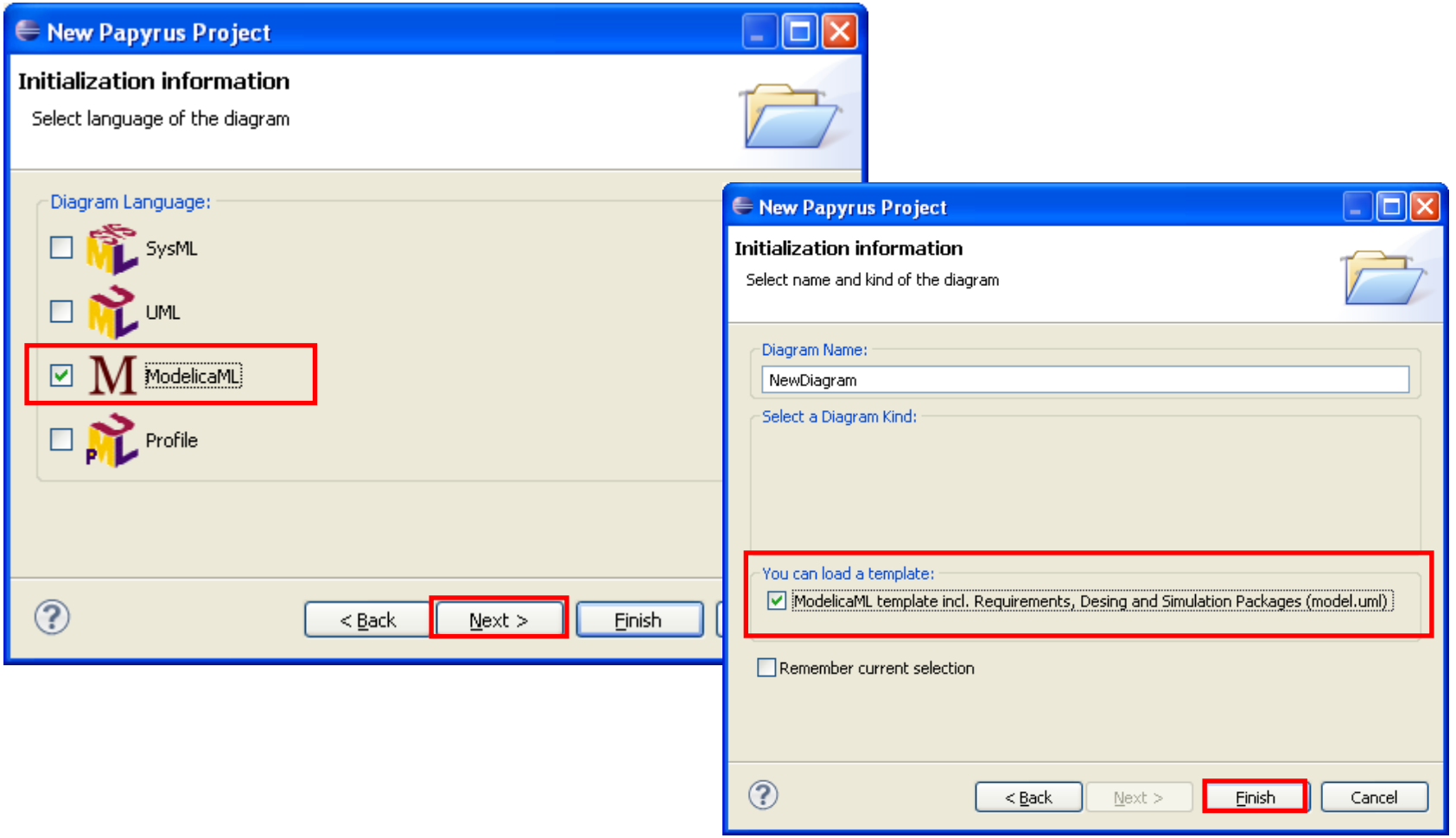
Create ModelicaML Project





Go to File -> Create …

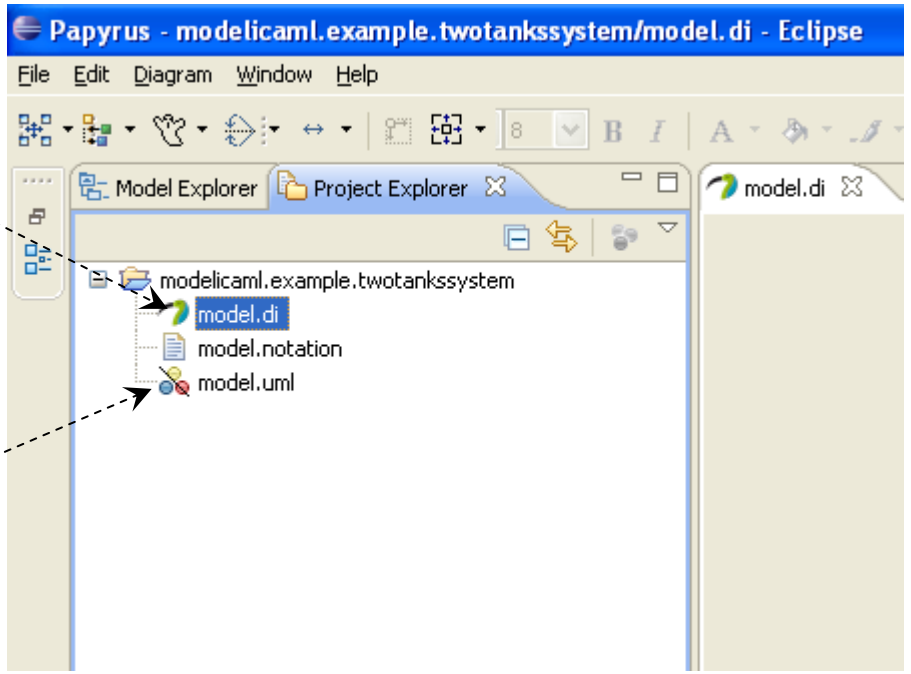
Create ModelicaML Project



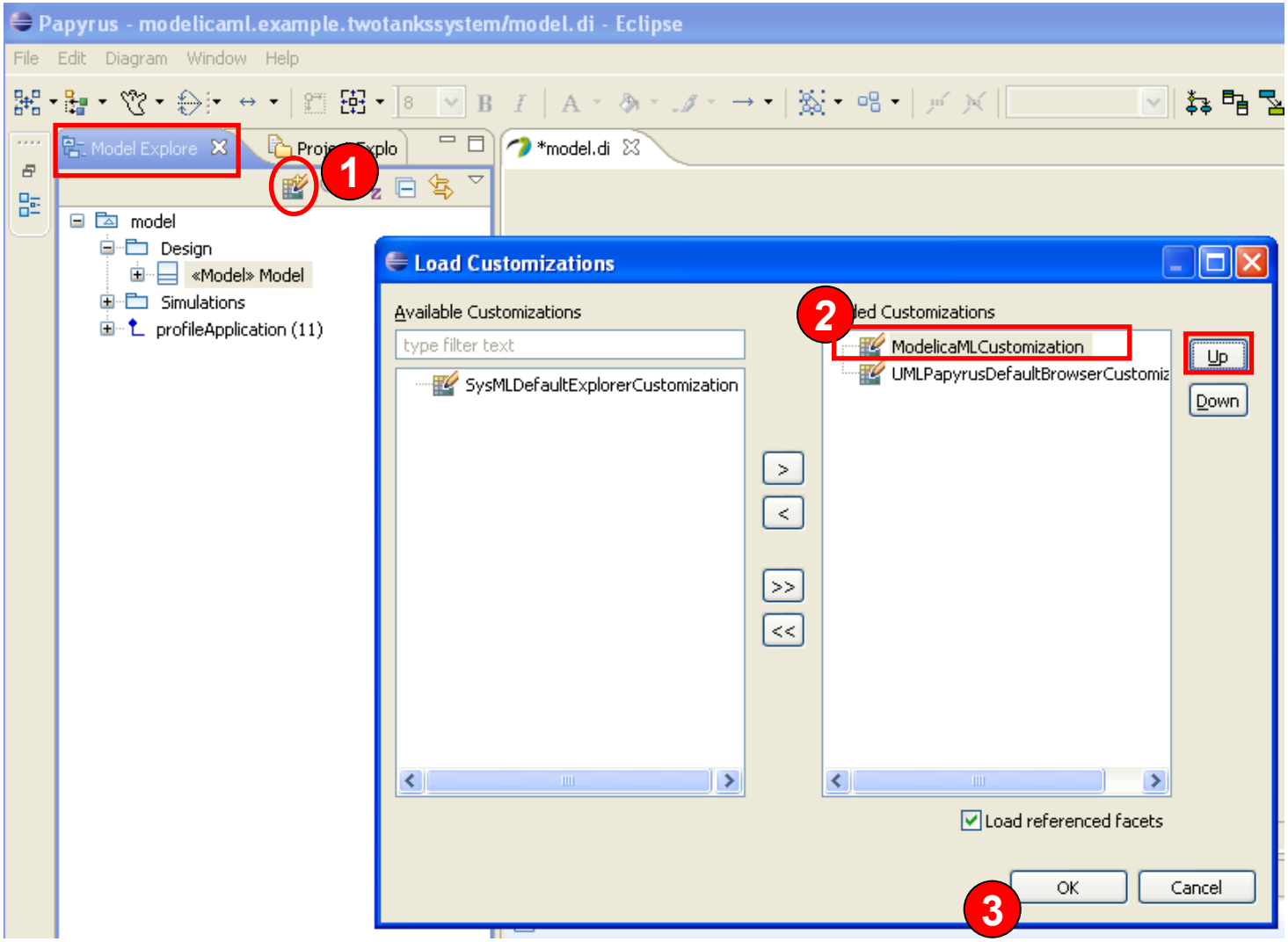
Papyrus Project Files

Diagram file
(can only be edited using Papyrus MDT)

Model file
(can be edited using any UML2 tool)

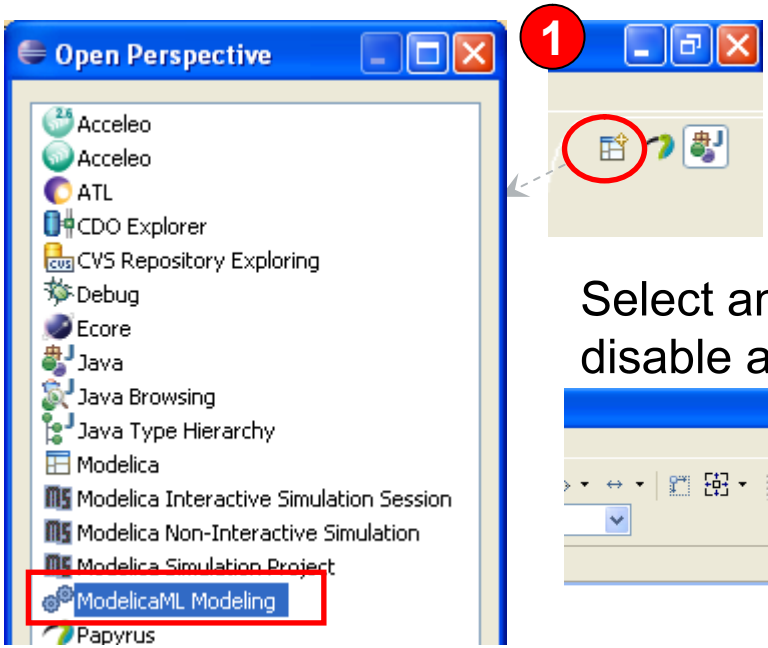


Configure Model Explorer

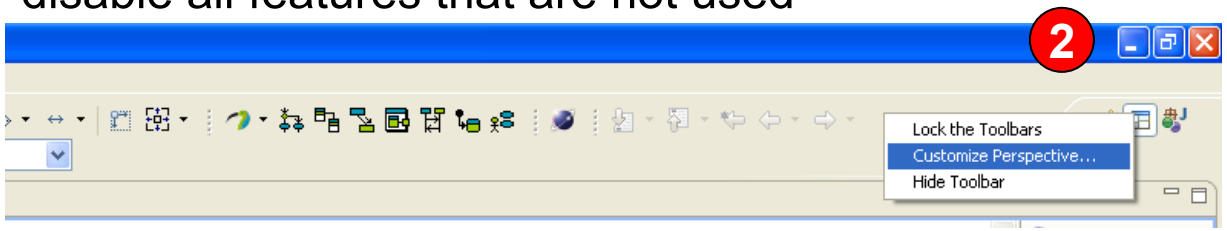


EADS

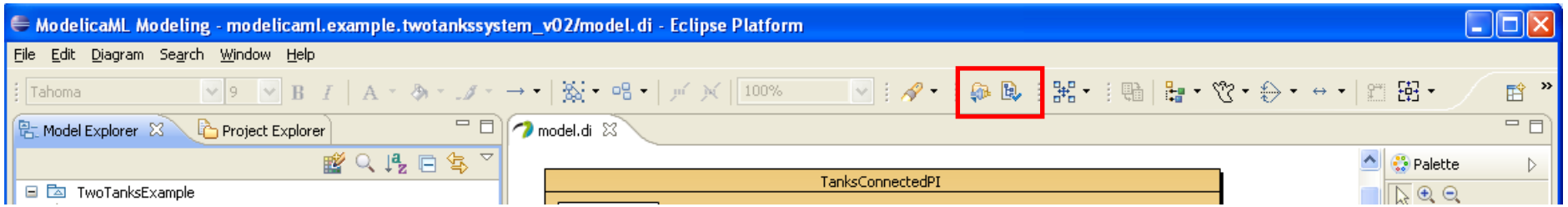
Change and Customize the Perspective



Select and customize the ModelicaML perspective – disable all features that are not used

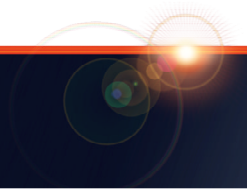


Typical ModelicaML customized perspective:



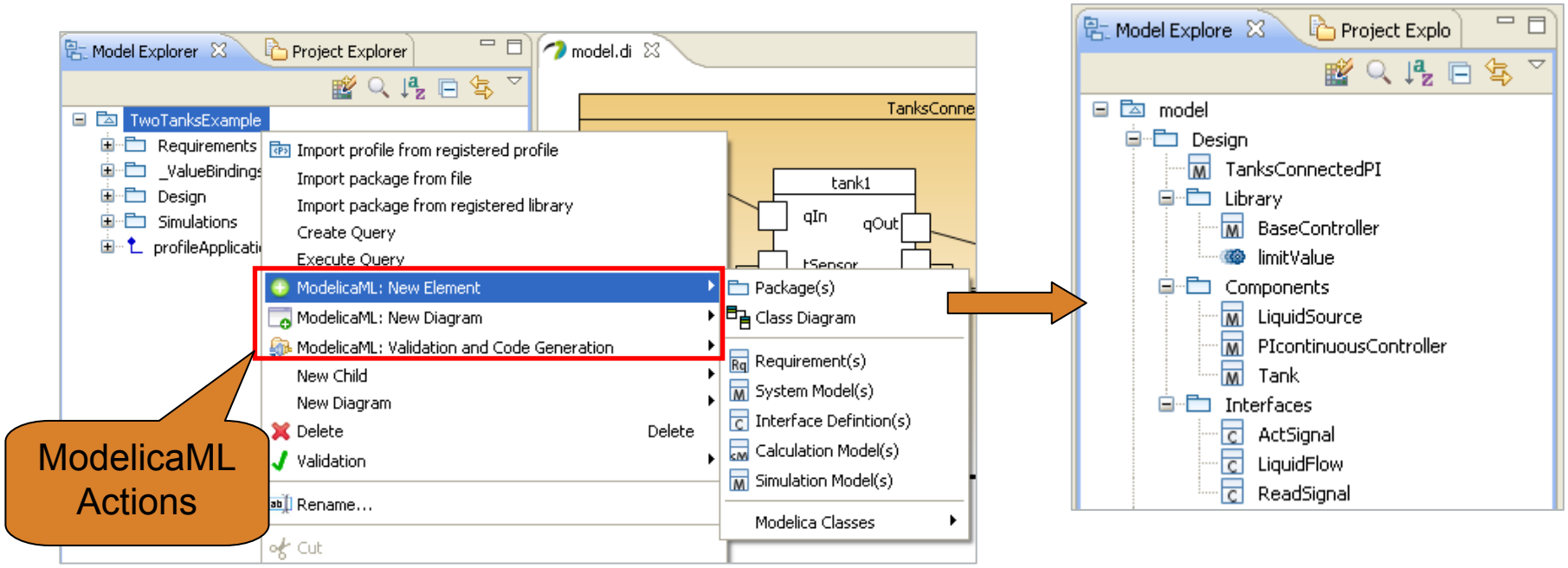
EADS

Model Setup

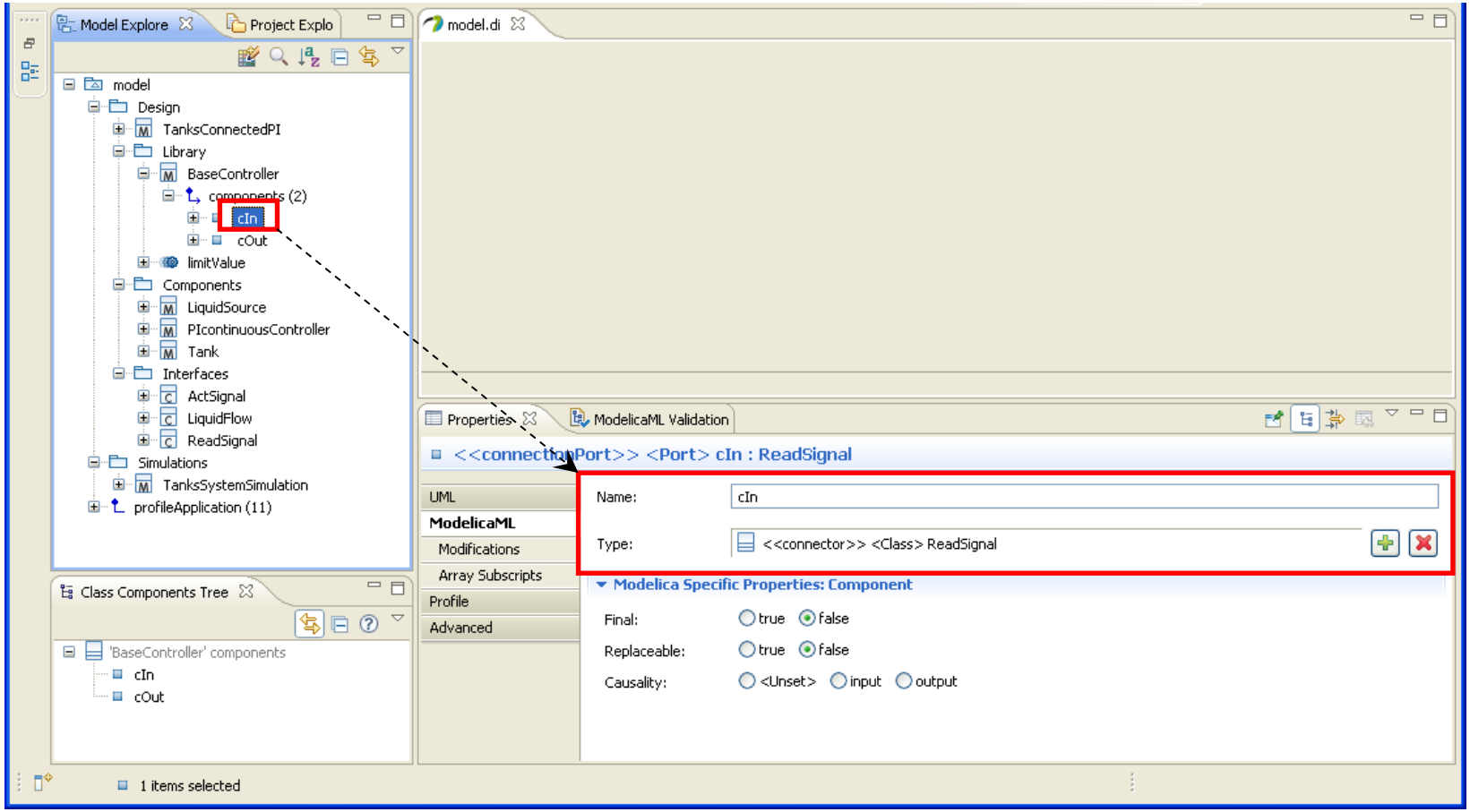


Create Model Structure

Create Packages and Classes using ModelicaML menus



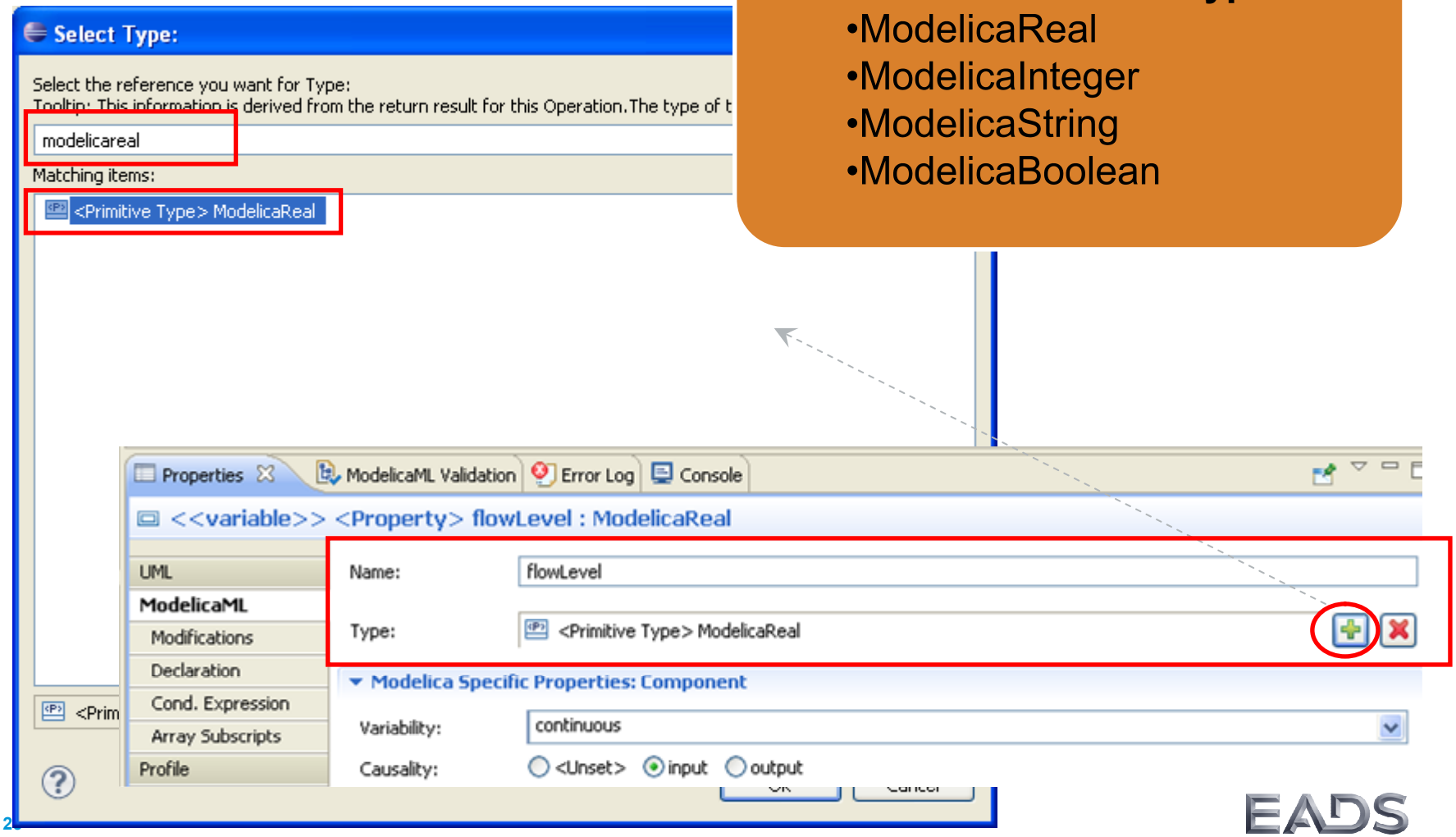
Hint: Setting type of components



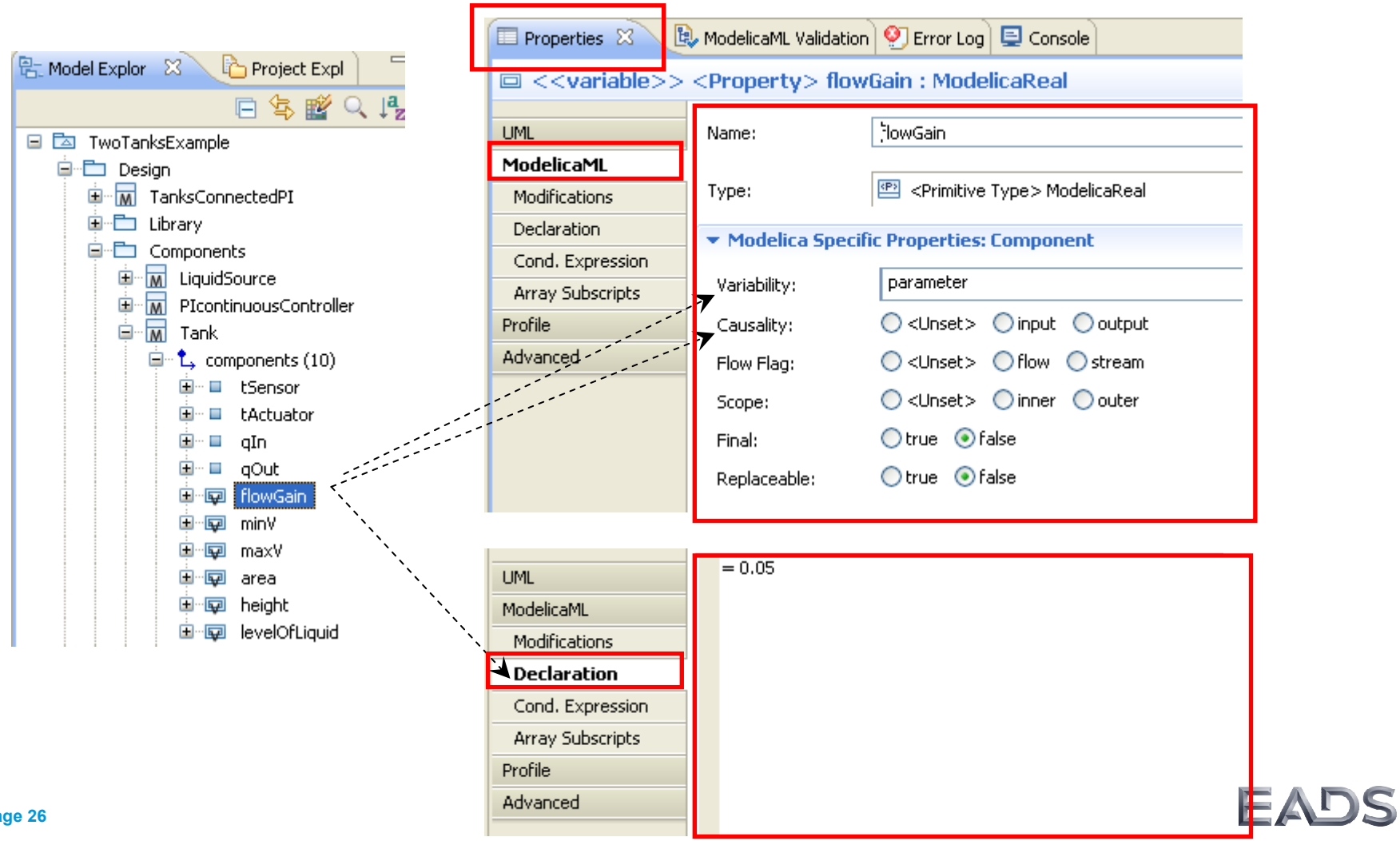
EADS

Hint: Setting type of components

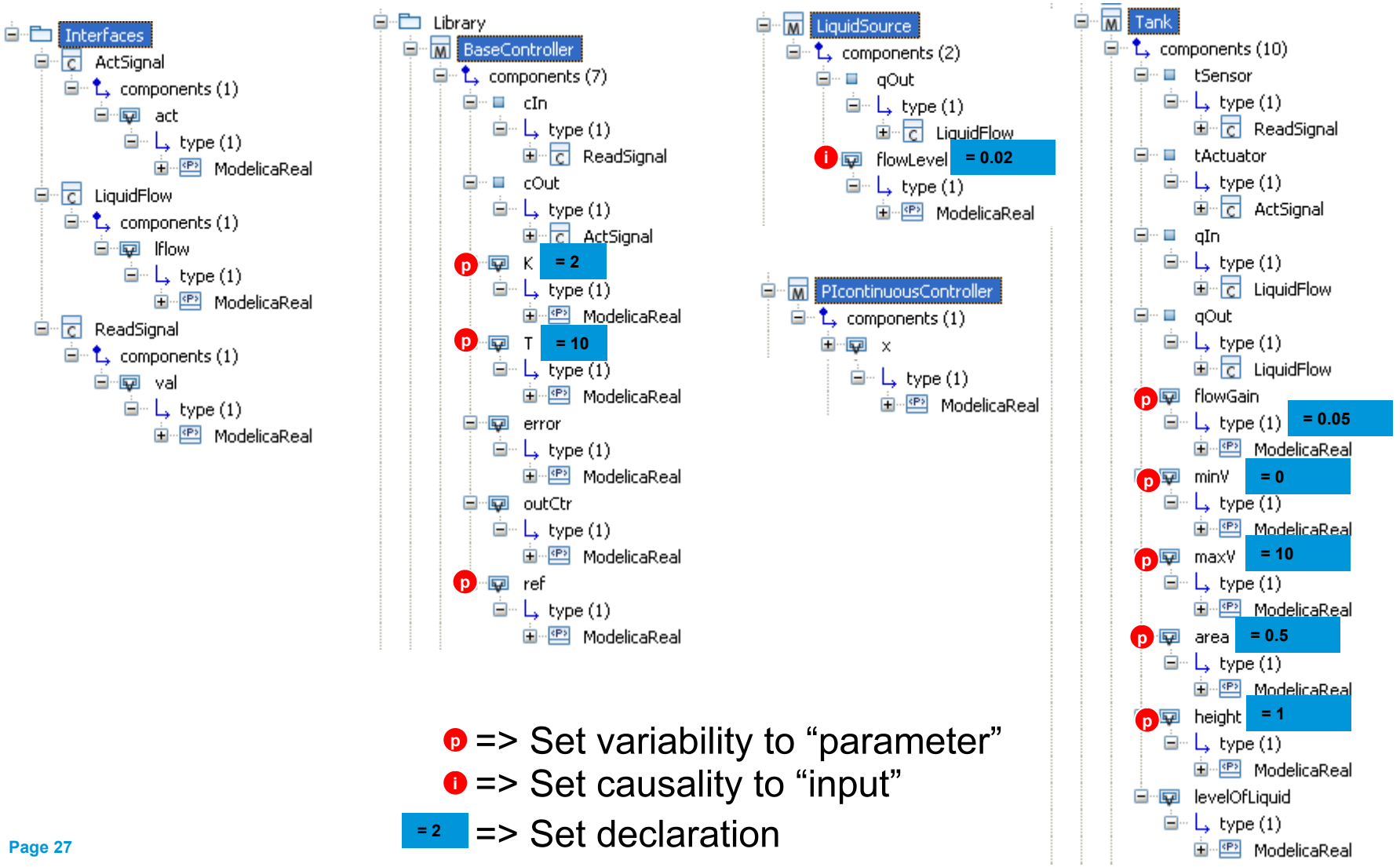
ModelicaML Primitive Types are:
- ModelicaReal
- ModelicaInteger
- ModelicaString
- ModelicaBoolean



Hint: Setting of the component properties (Declaration, Causality, Variability, etc.)



Create Class Components

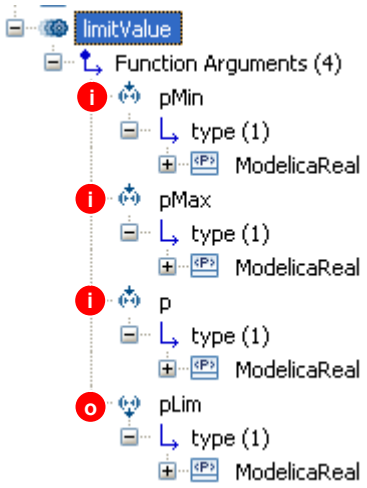


p => Set variability to “parameter”

i => Set causality to “input”

= 2 => Set declaration

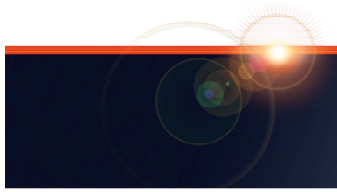
Create Function Arguments



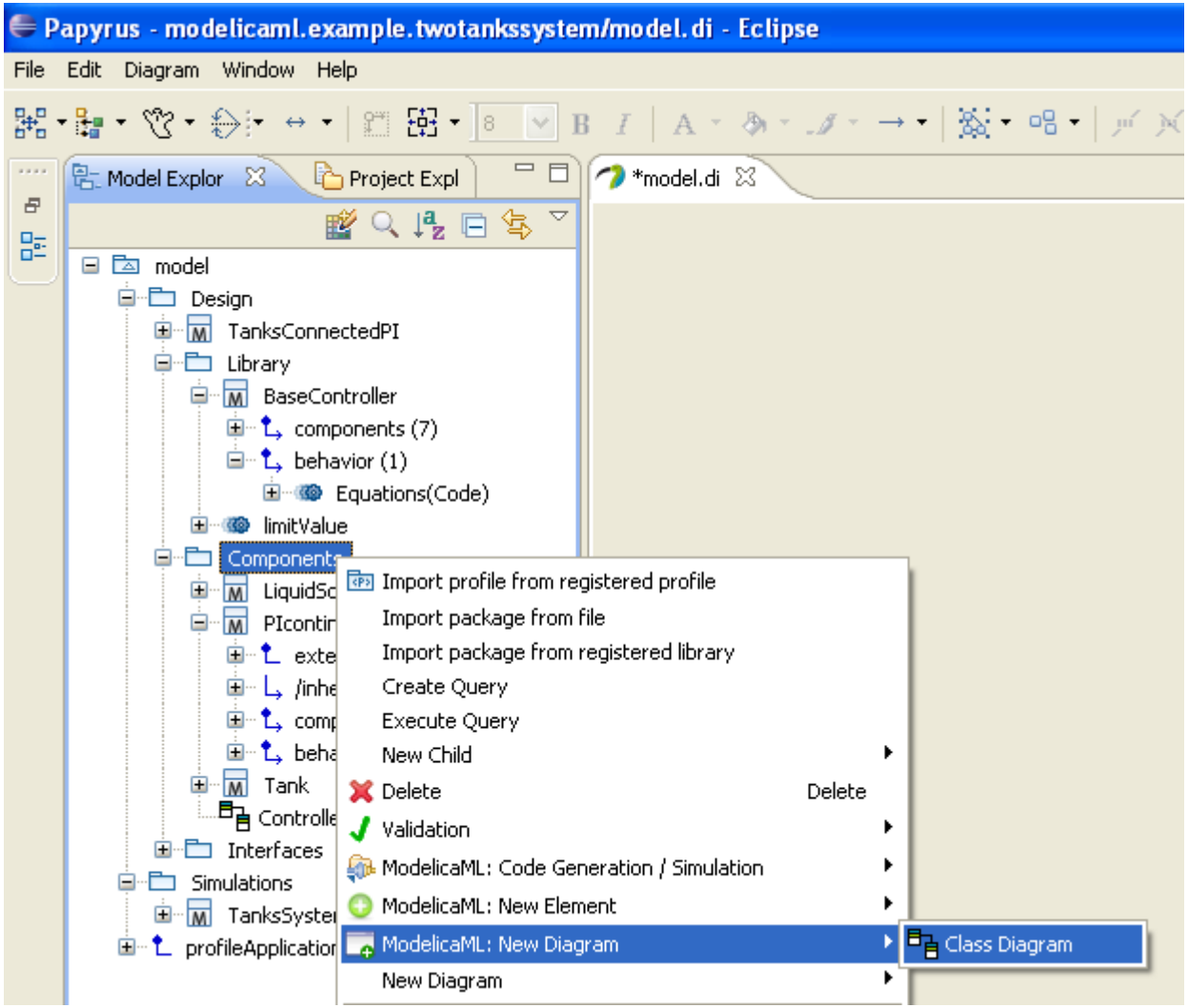
i => Set causality to “input”

o => Set causality to “output”

Inheritance/Extension Modeling



Create Class Diagram

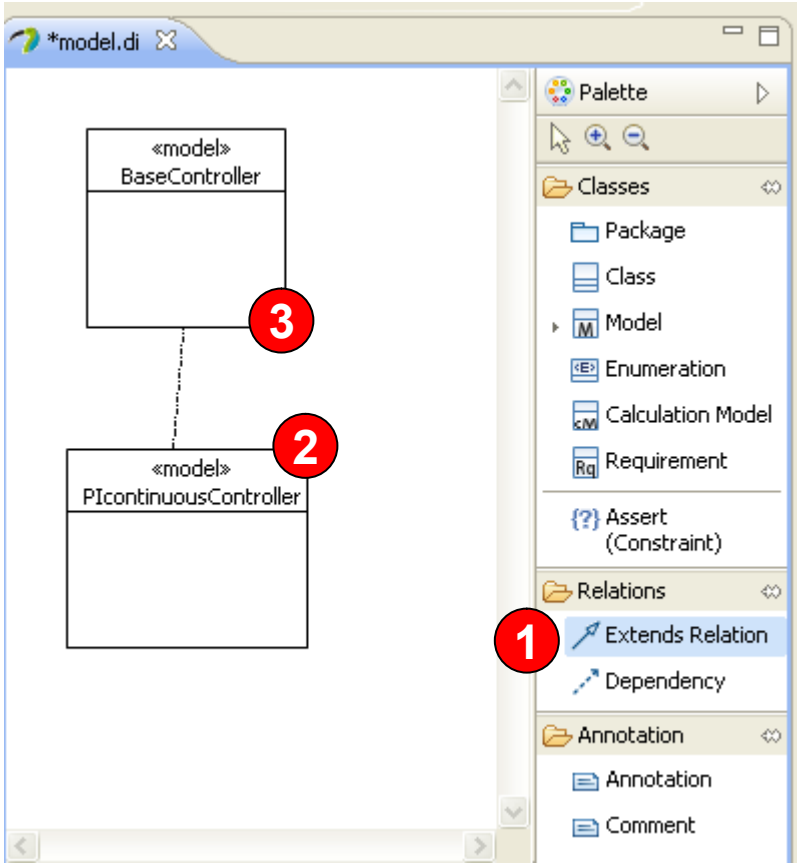


EADS

General: Working with diagrams

Creating edges:

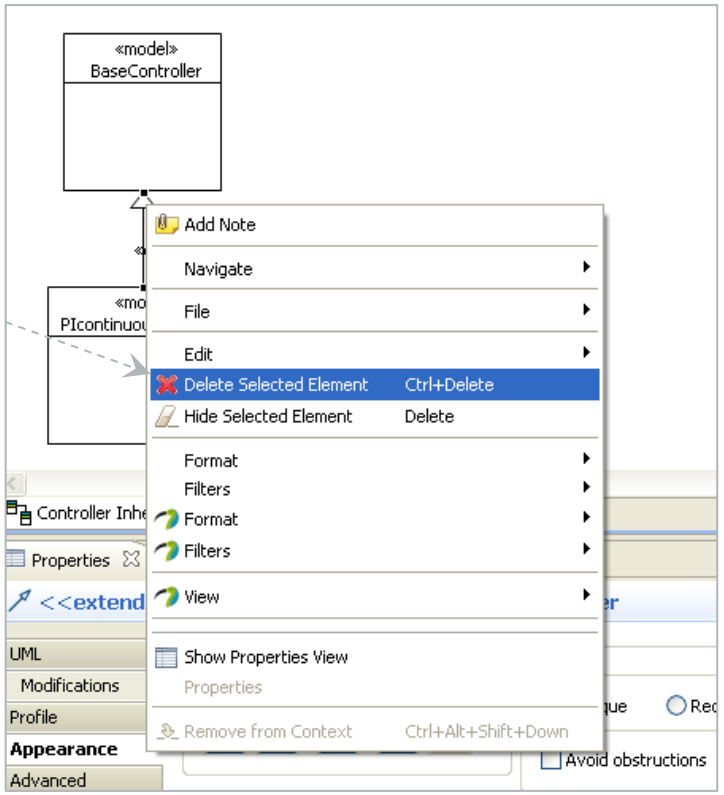
1. Select the palette tool
2. Click on the edge source element and hold the mouse button
3. Move the mouse to the target element and release the mouse button



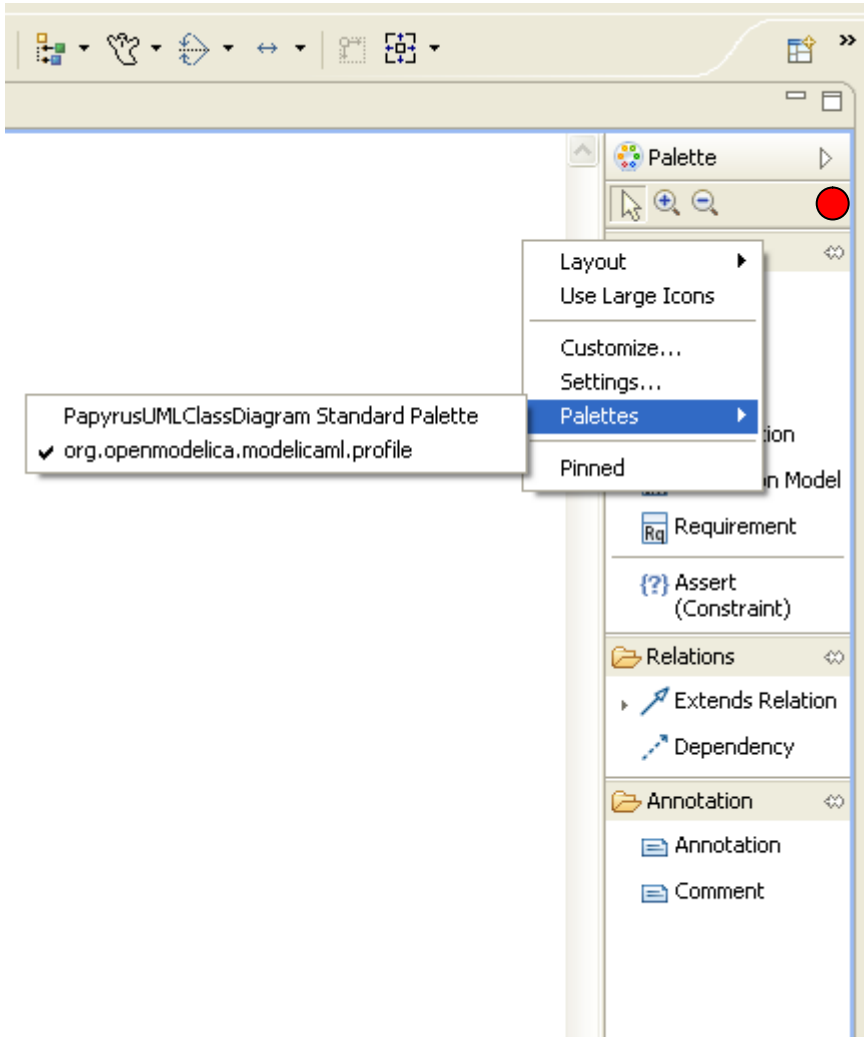
General: Working with diagrams

Deletion of elements:

- Right click on a diagram element
- (preferred) Select the option “Delete Selected Element” in order to delete it from the model. This is recommended in order to keep the model and the diagram consistent
- Select the option “Hide Selected Element” in order to remove the element from the diagram. The element will still exist in the model and can be shown on the diagram by drag&drop.



Configure Diagram Palette



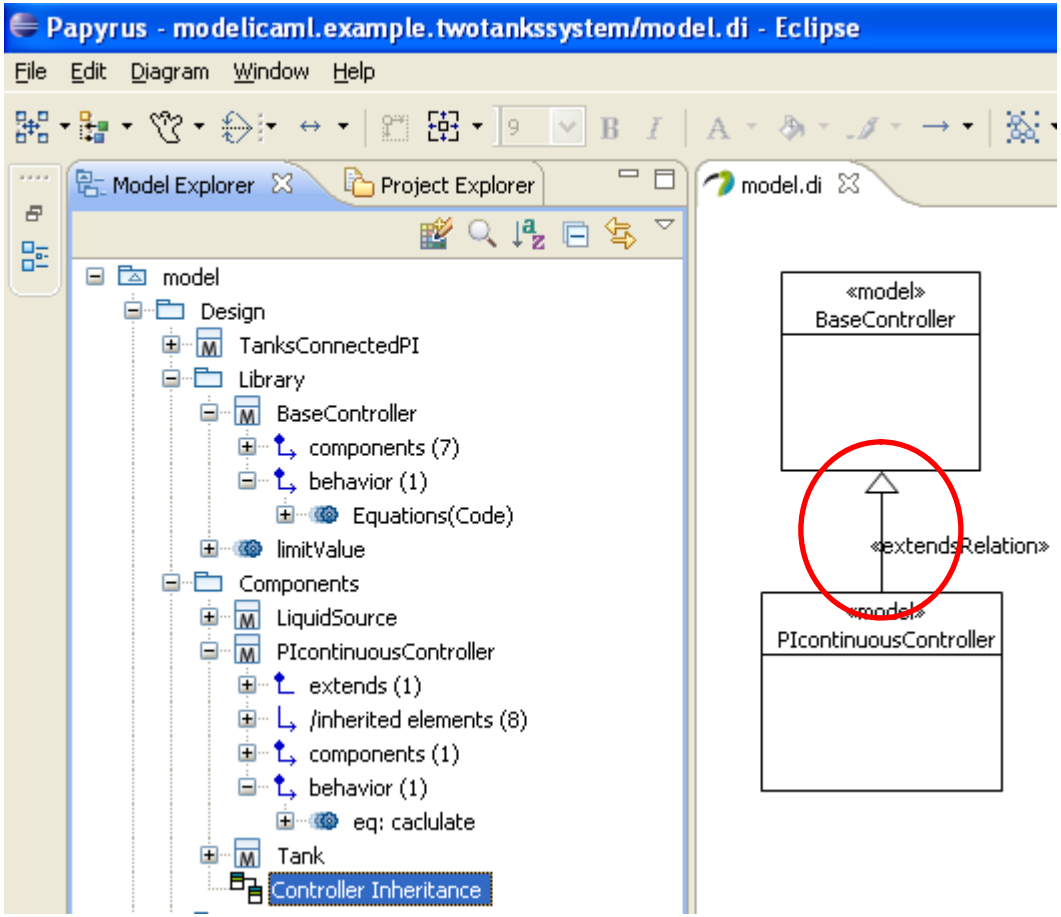
right click here…

EADS

Create Extends Relation

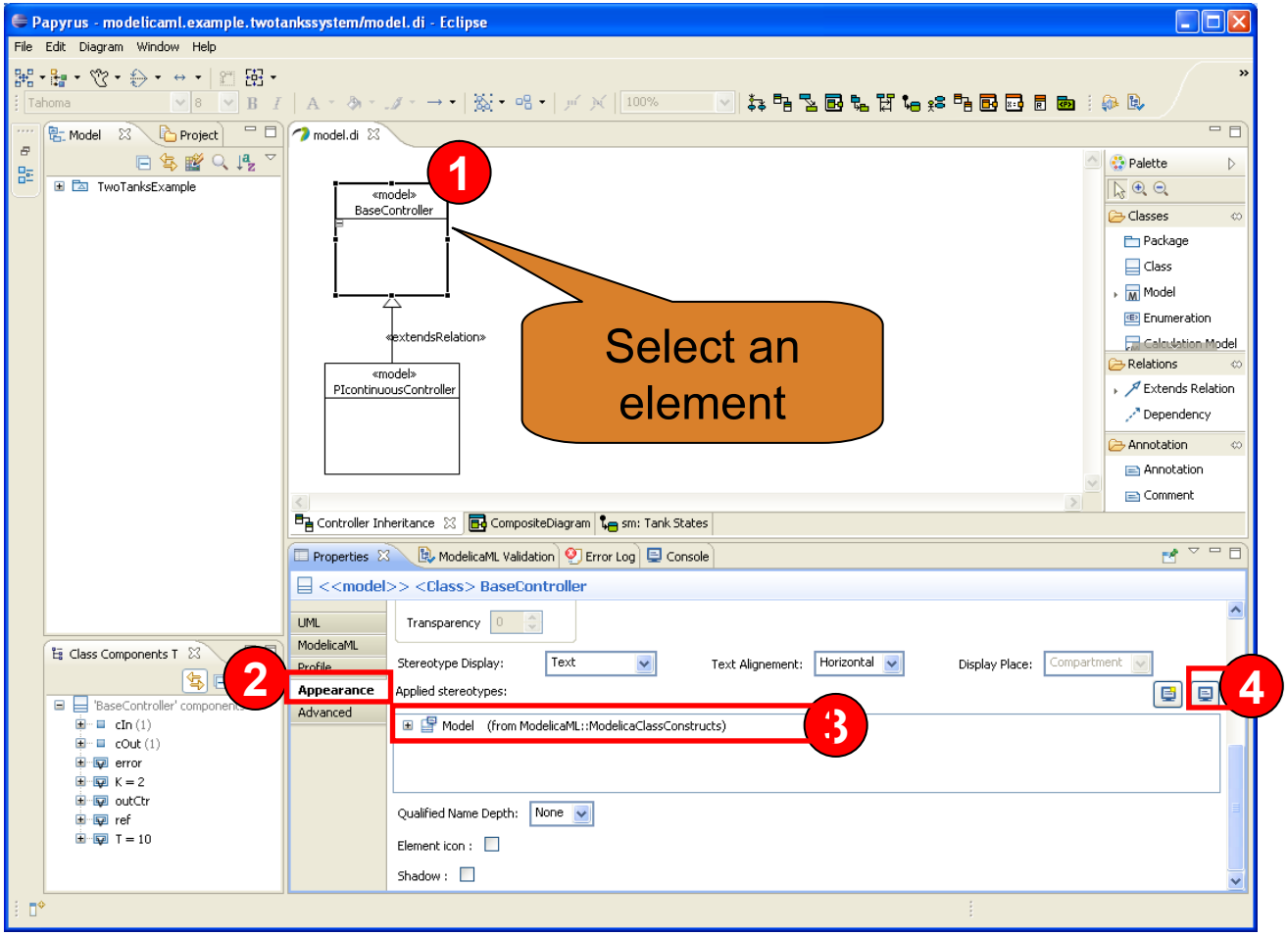
•Drag & drop BaseController and PIcontinuousController onto diagram

•Use the palette tool “Extends Relation”

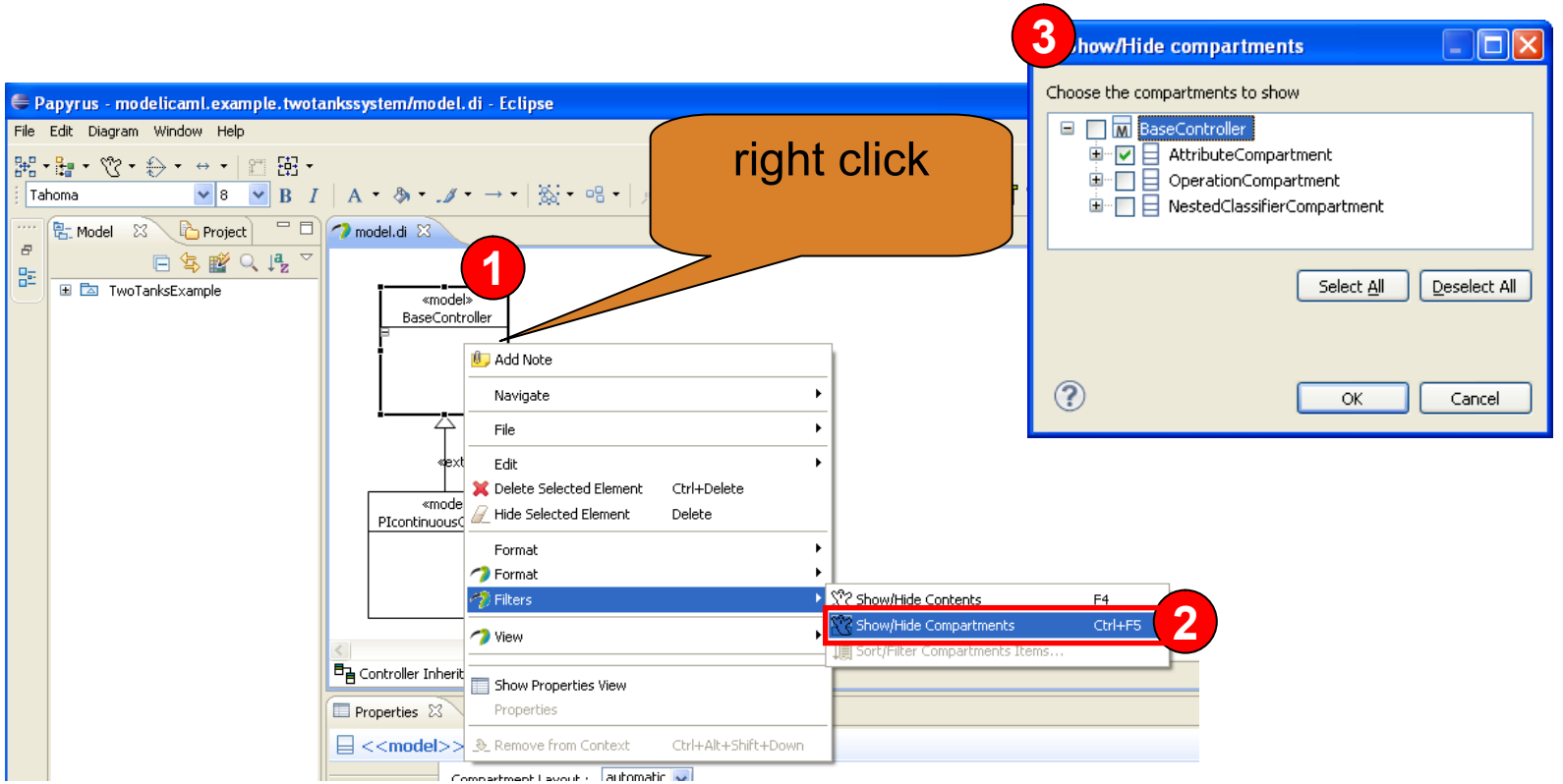


EADS

Hint: Element Appearance: Show stereotype name

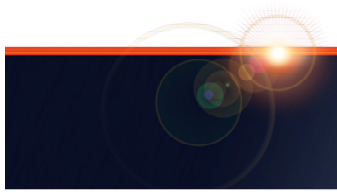


Hint: Element Appearance: Compartments



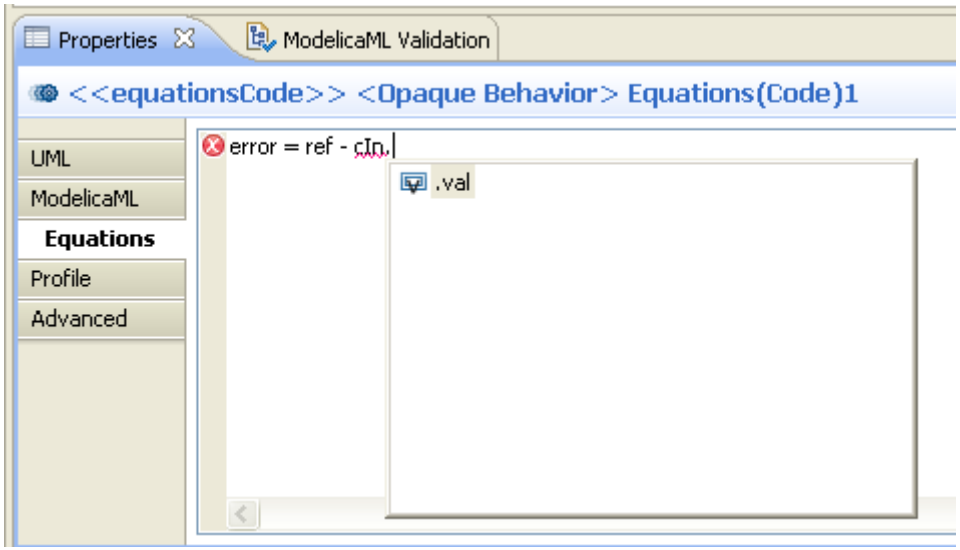
EADS

Behavior Modeling

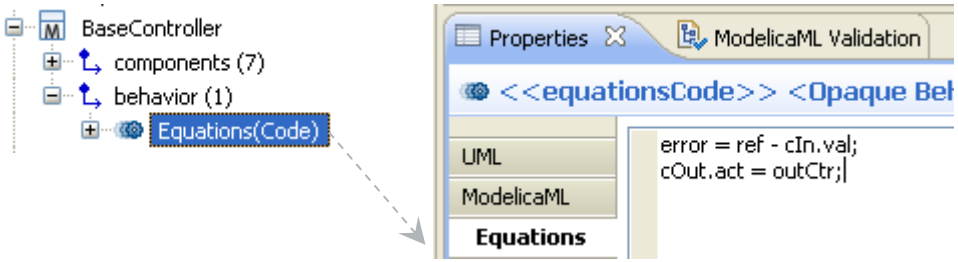


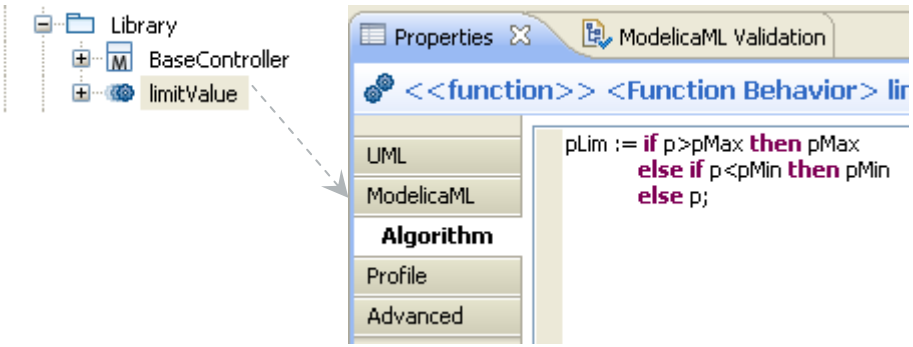
Hint: Editing Modelica Code

- Syntax highlighting and code completion is supported in code editors
- Hit **Ctrl + Space** for code completion when editing Modelica code



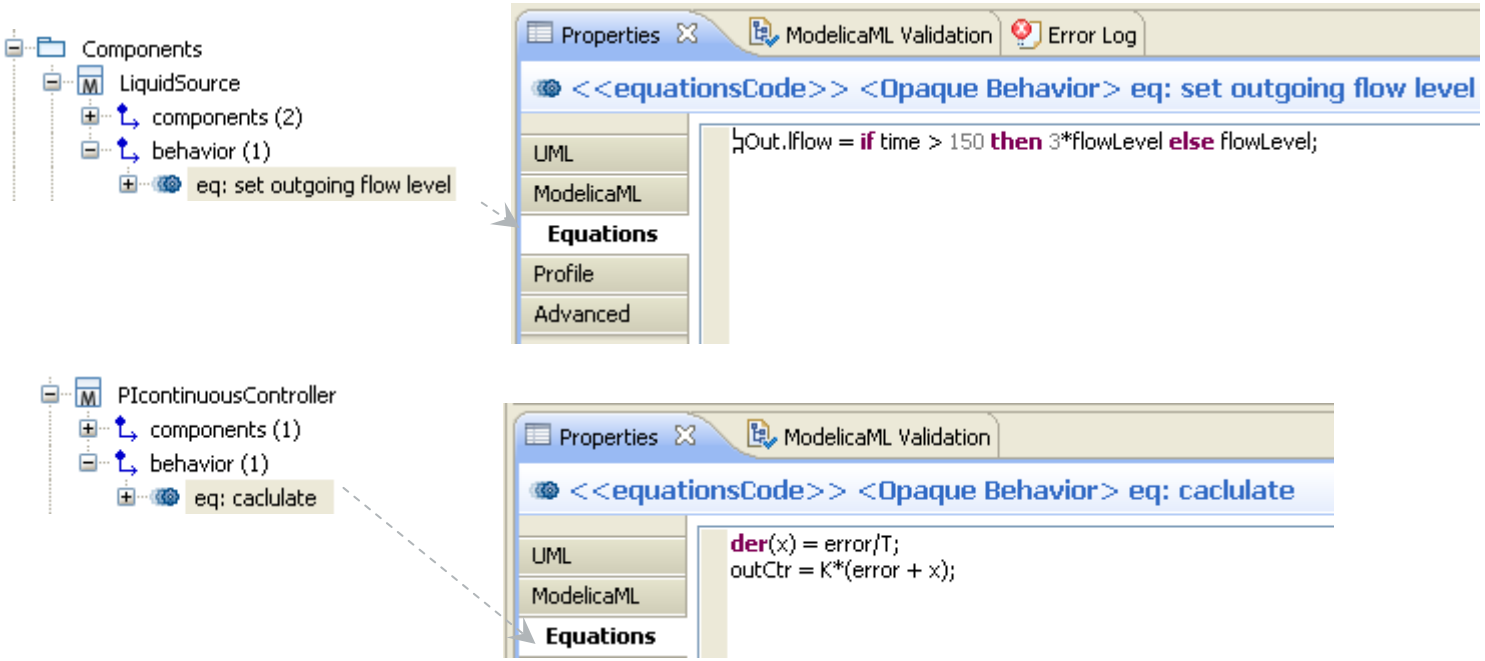
Create Behavior





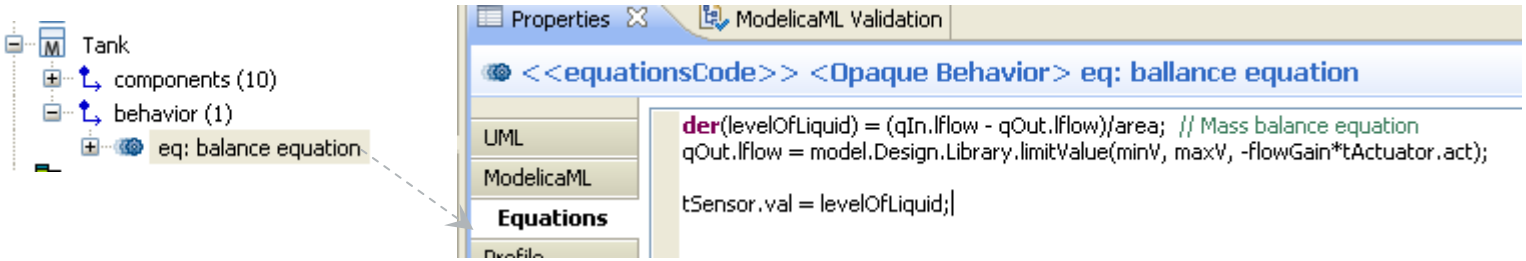
EADS

Create Behavior

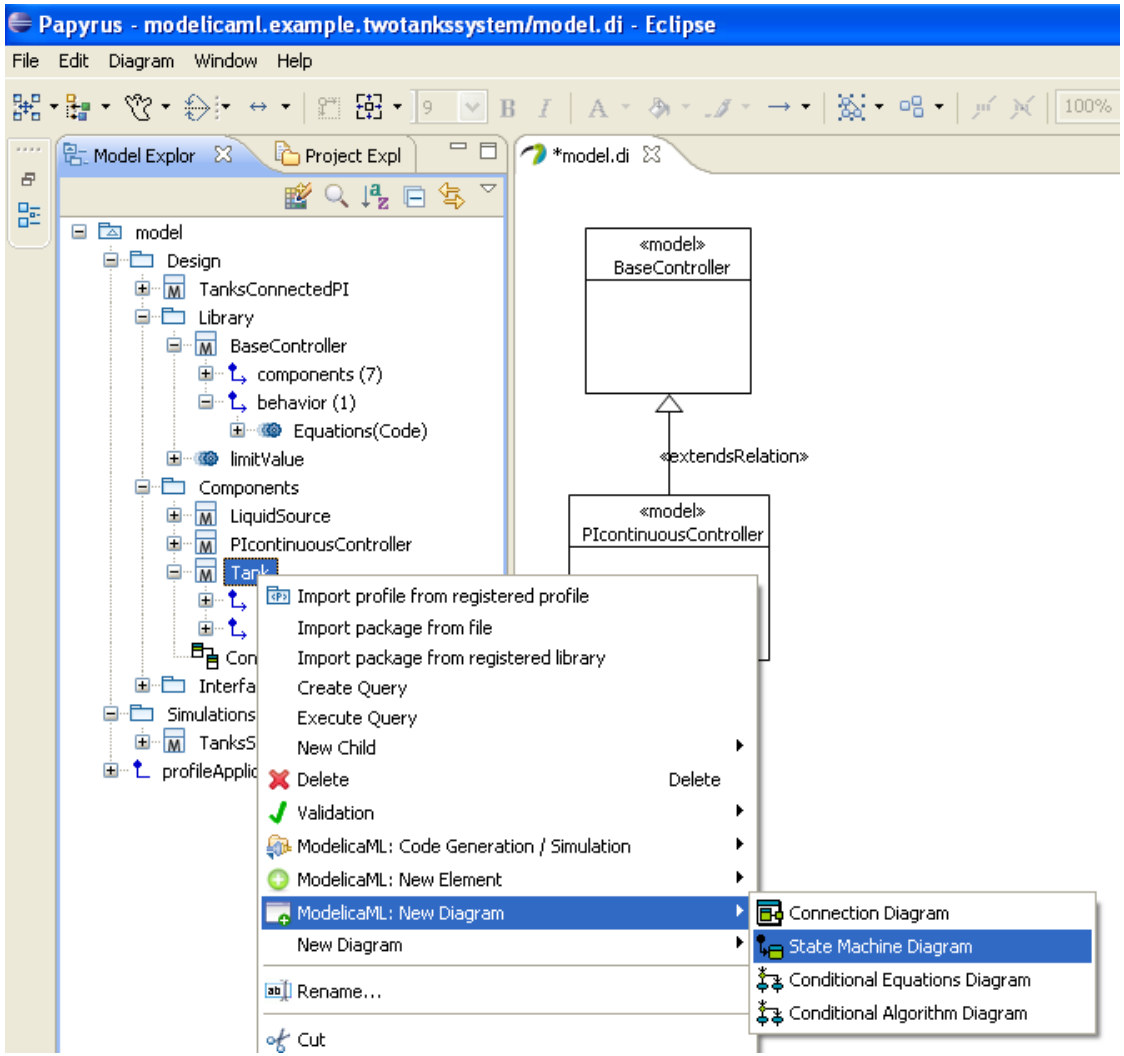


EADS

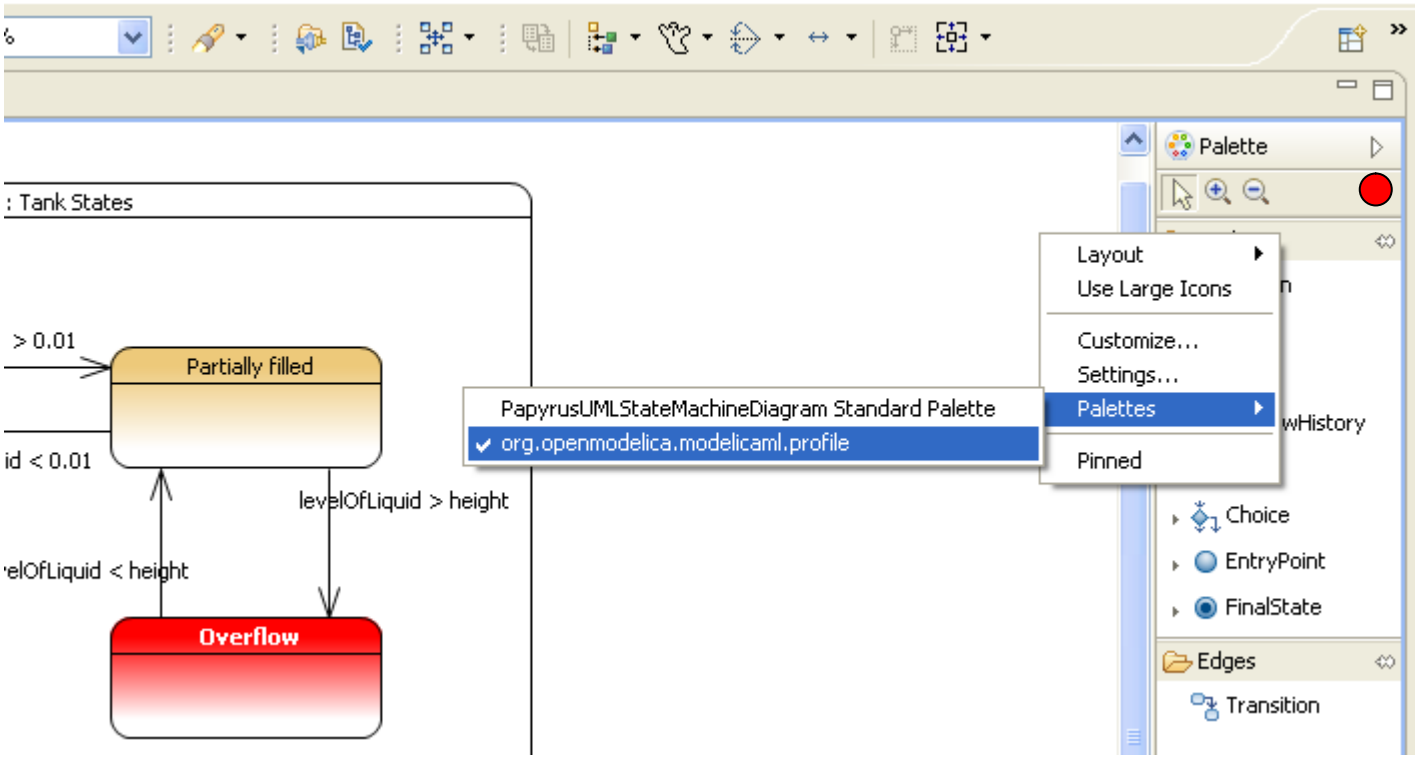
Create Behavior



Create State Machine



Configure Diagram Palette



right click here…

EADS

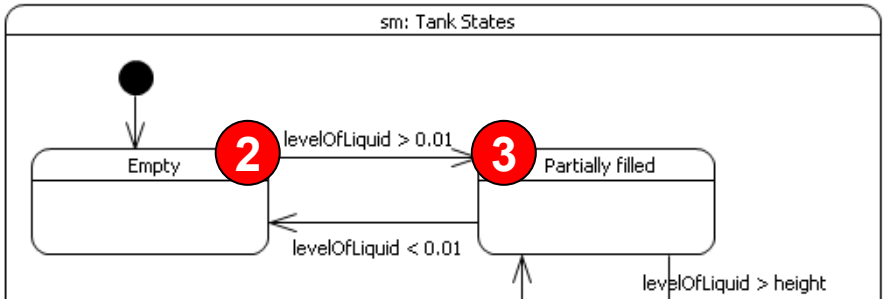
General: Working with diagrams

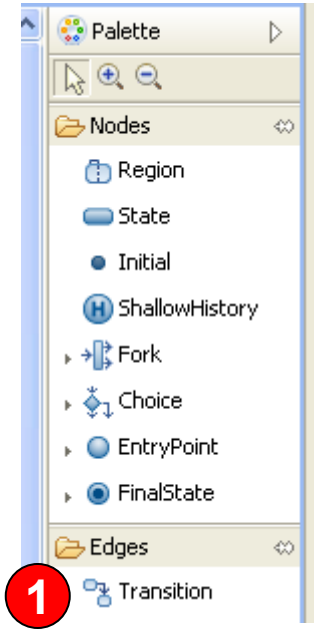
State Transitions:

1. Select the palette tool

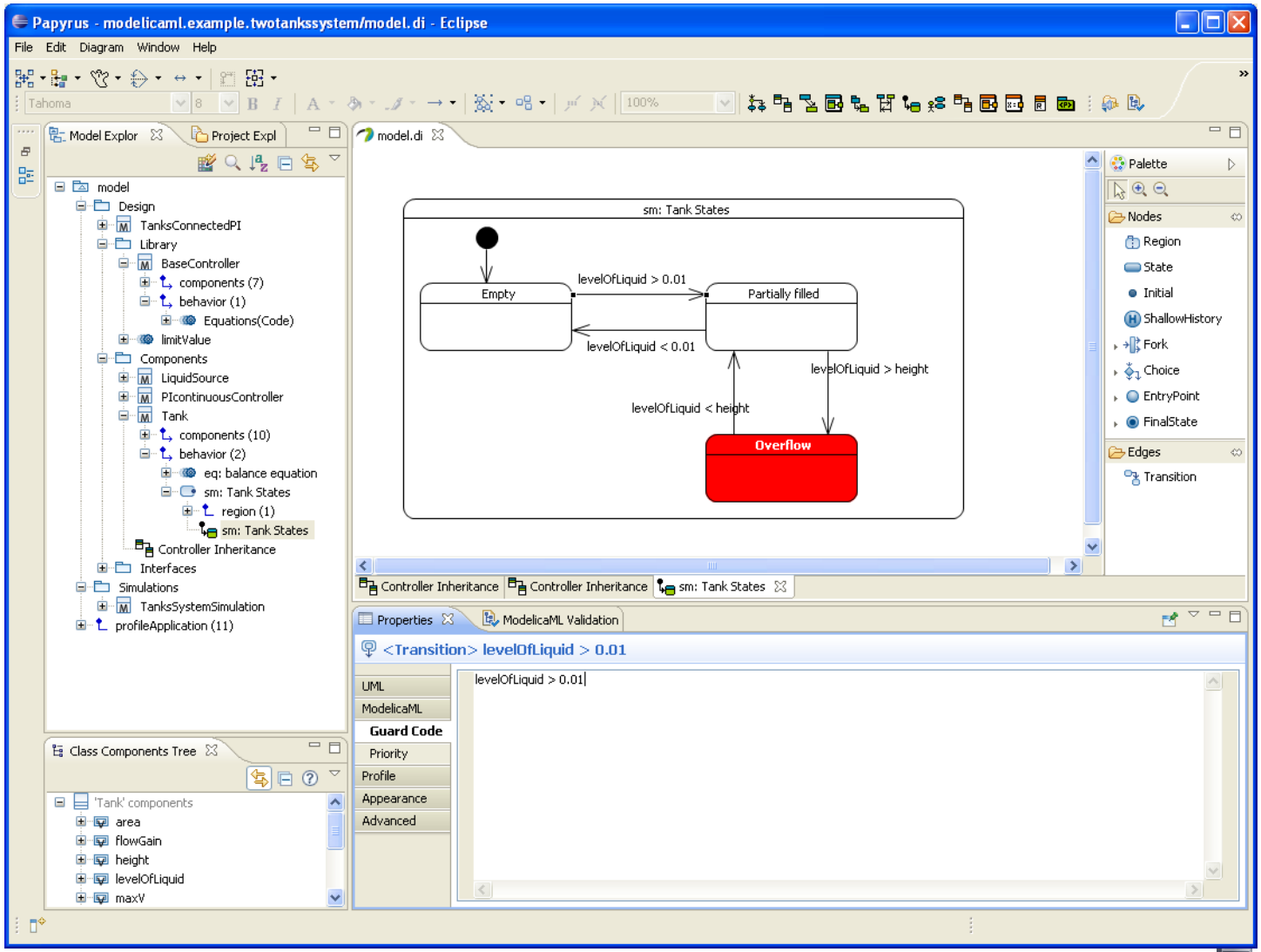
2. Click on the transition source state (click on label) and hold the mouse button

3. Move the mouse to the target state (to its label) and release the mouse button





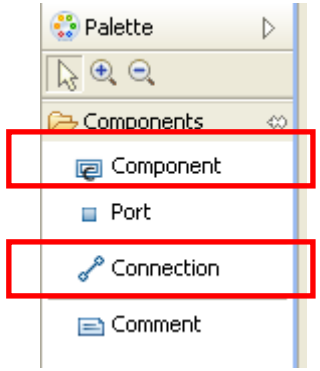
Create State Machine



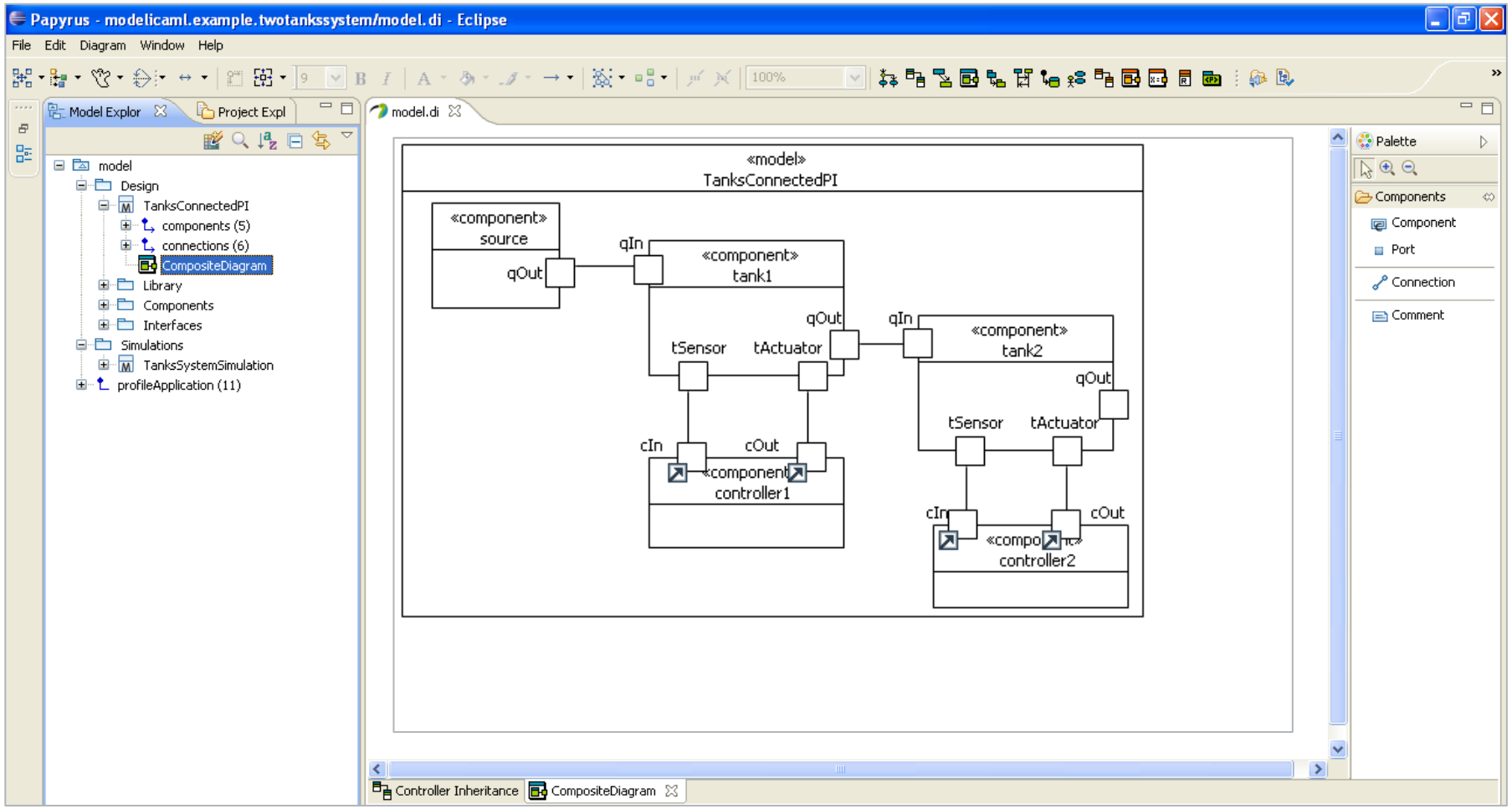
Architecture Modeling

Create Connection Diagram

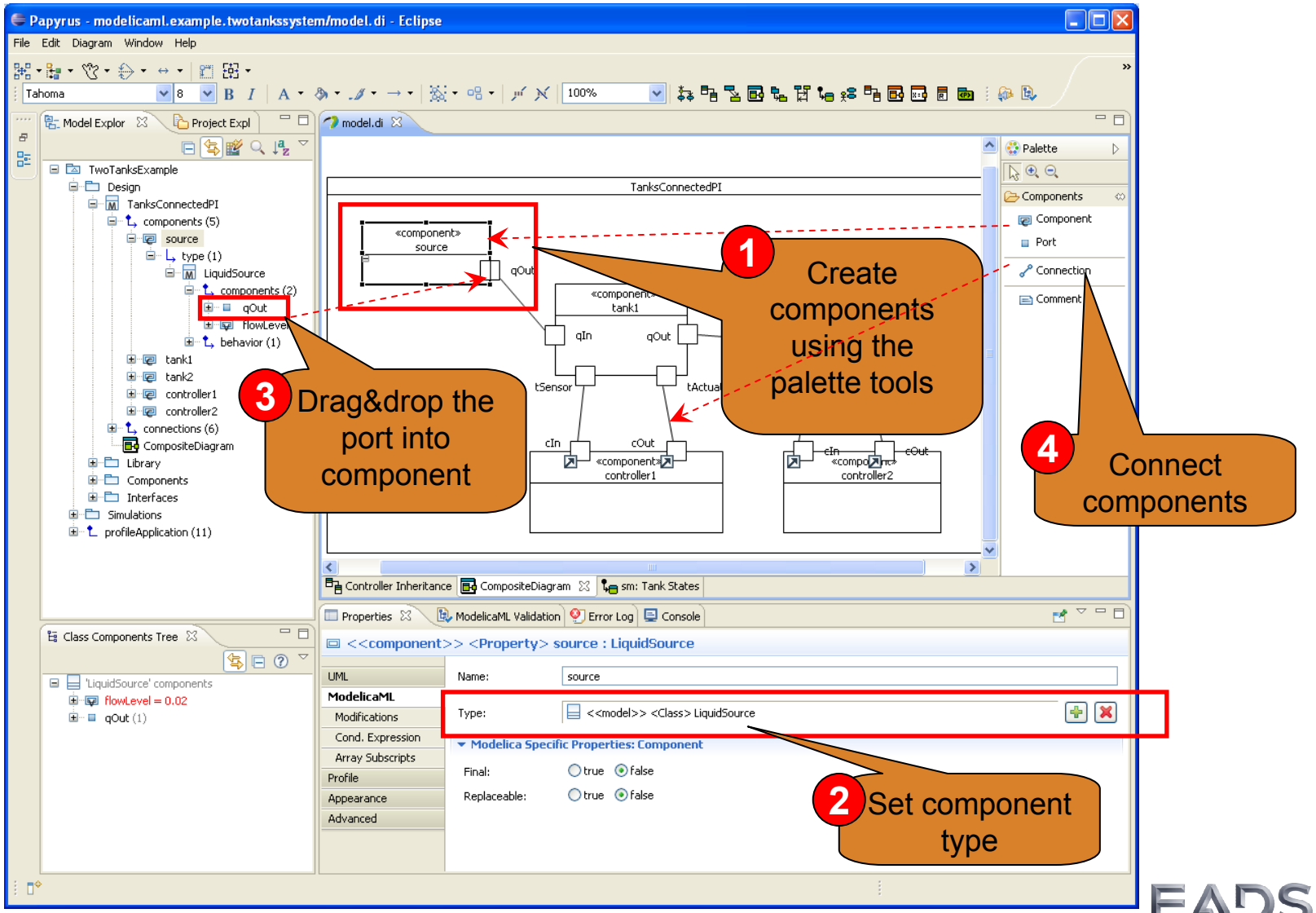
- Create a ModelicaML Connection Diagram under the TanksConnectedPI class
- Use components tool from the palette to create components inside the class on the diagram
- Define the types of components
- Use Model Explorer to find the ports
- Drag&Prop ports into respective components
- Arrange the components
- Use the “Connection” tool from the palette for connecting ports



Model System Architecture

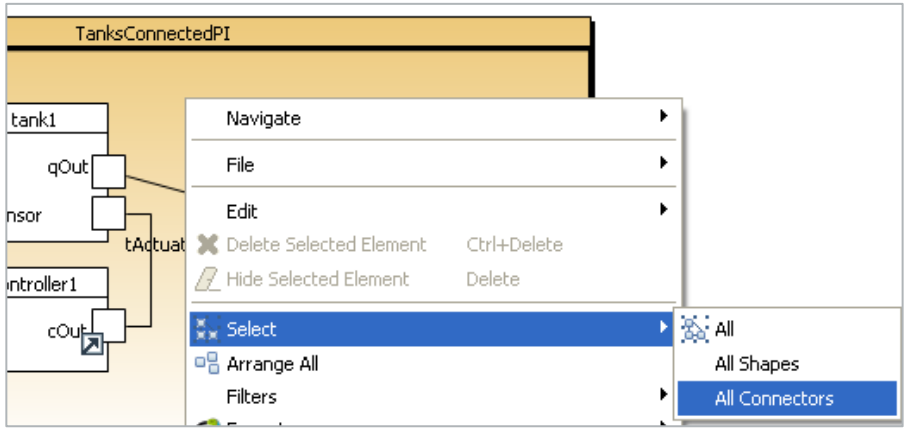


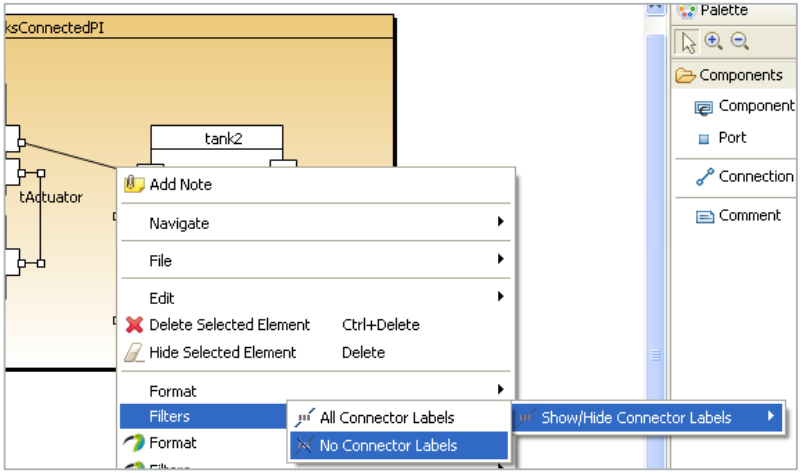
Create Connection Diagram



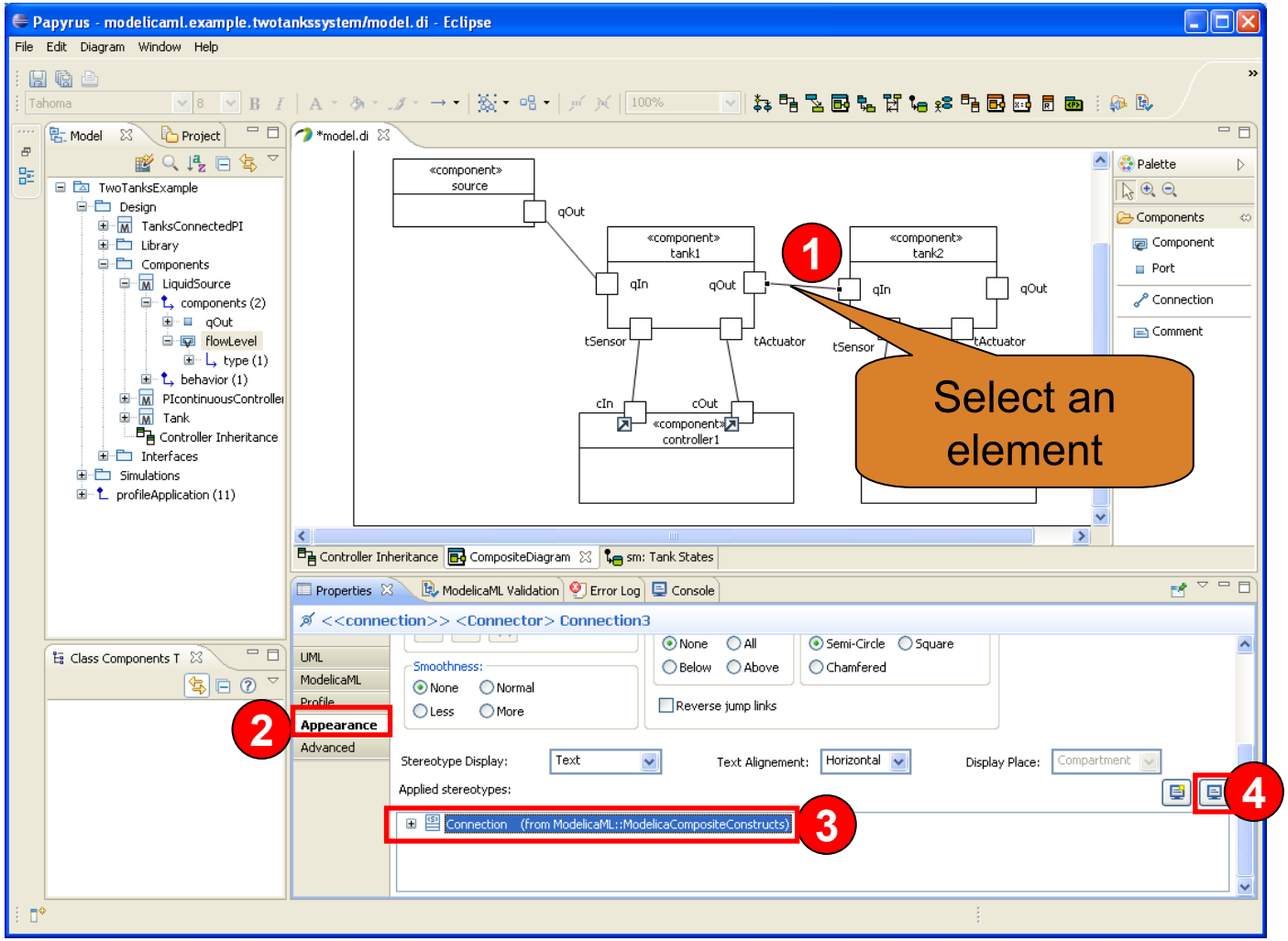
Hint: Hide the name of all connectors

- Click on the compartment of the class
- Right-click -> “Select” -> “All Connectors”
- Right-click on one of the selected connectors -> “Filters” -> “Show/Hide connector Labels” -> “No connector Labels”





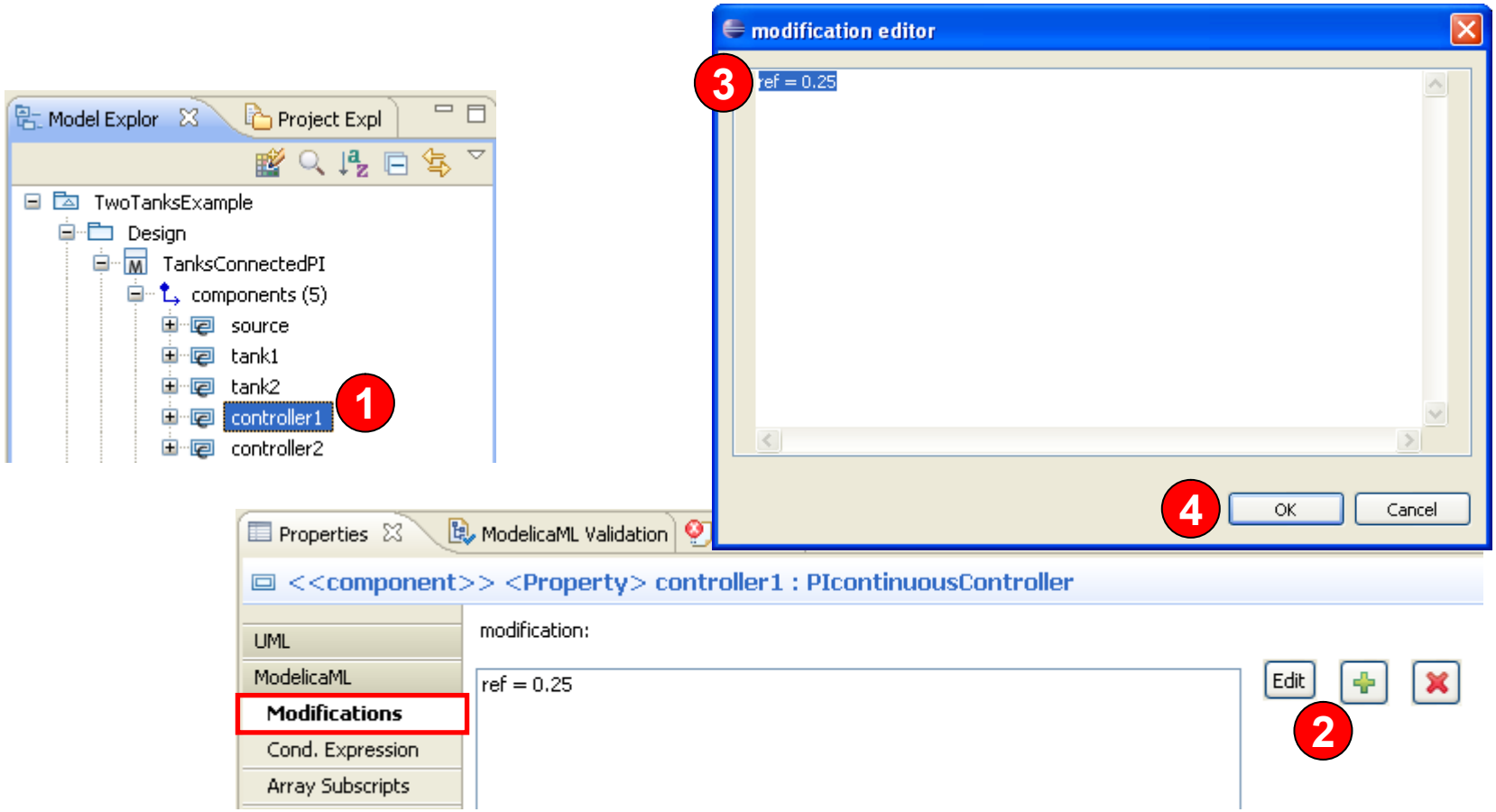
Hint: Element Appearance: Hide the name of the connection stereotype



EADS

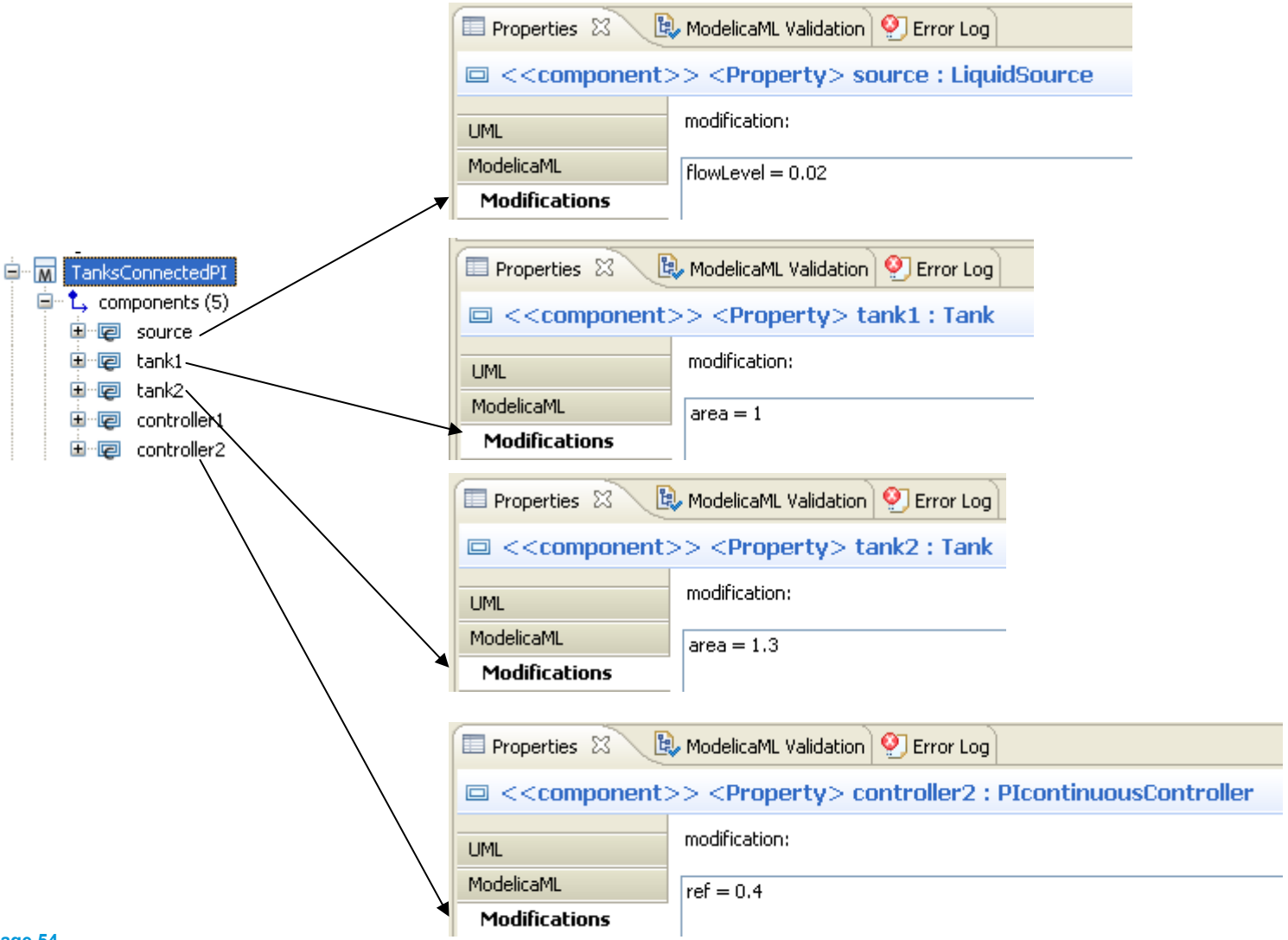
Component Modifications

Define Component Modifications



EADS

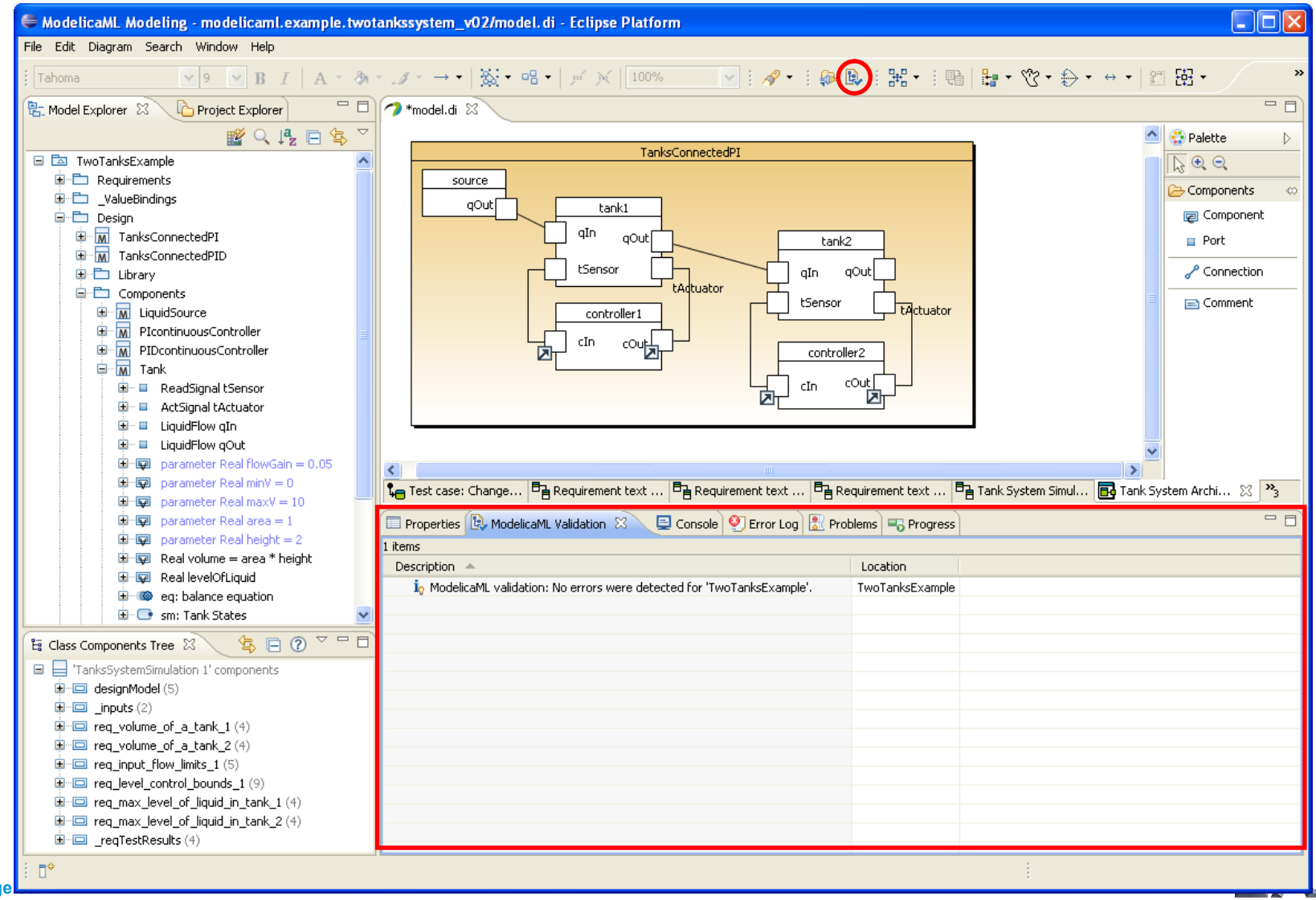
Define Component Modifications



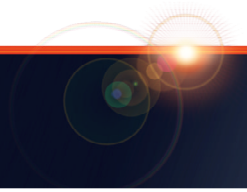
EADS

Model Validation

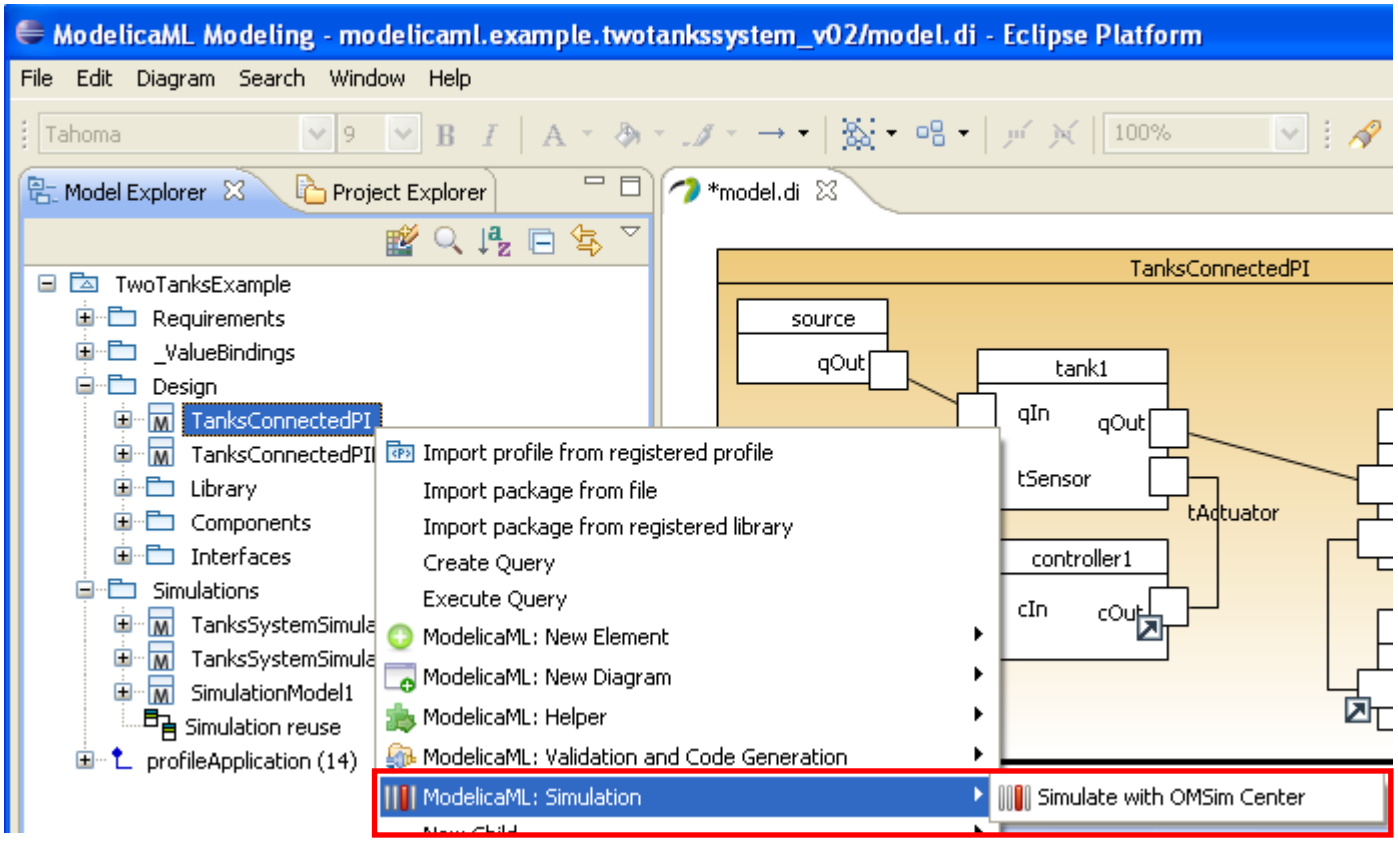
Validate Model



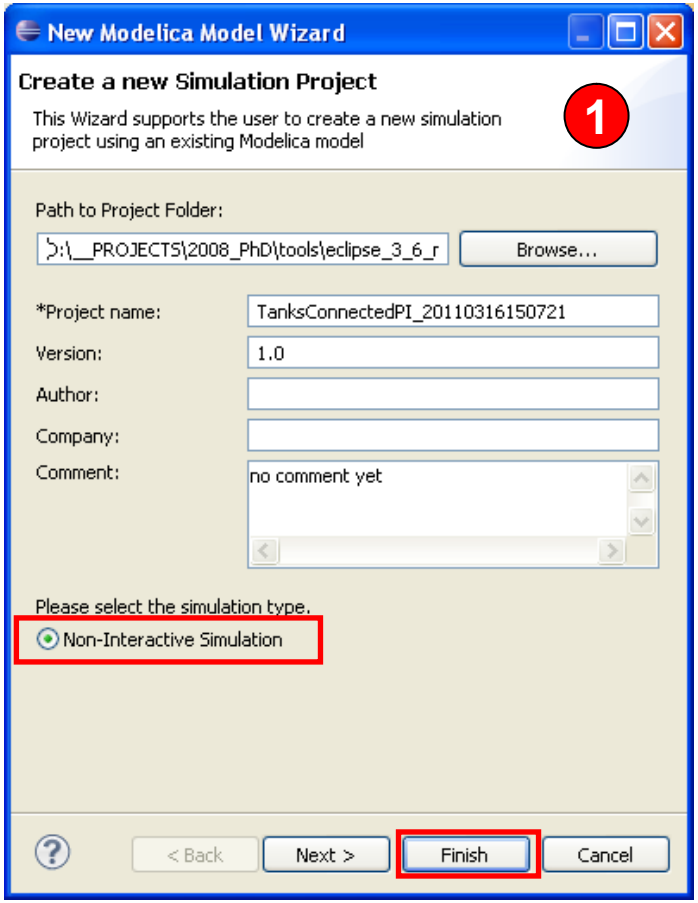
Model Simulation

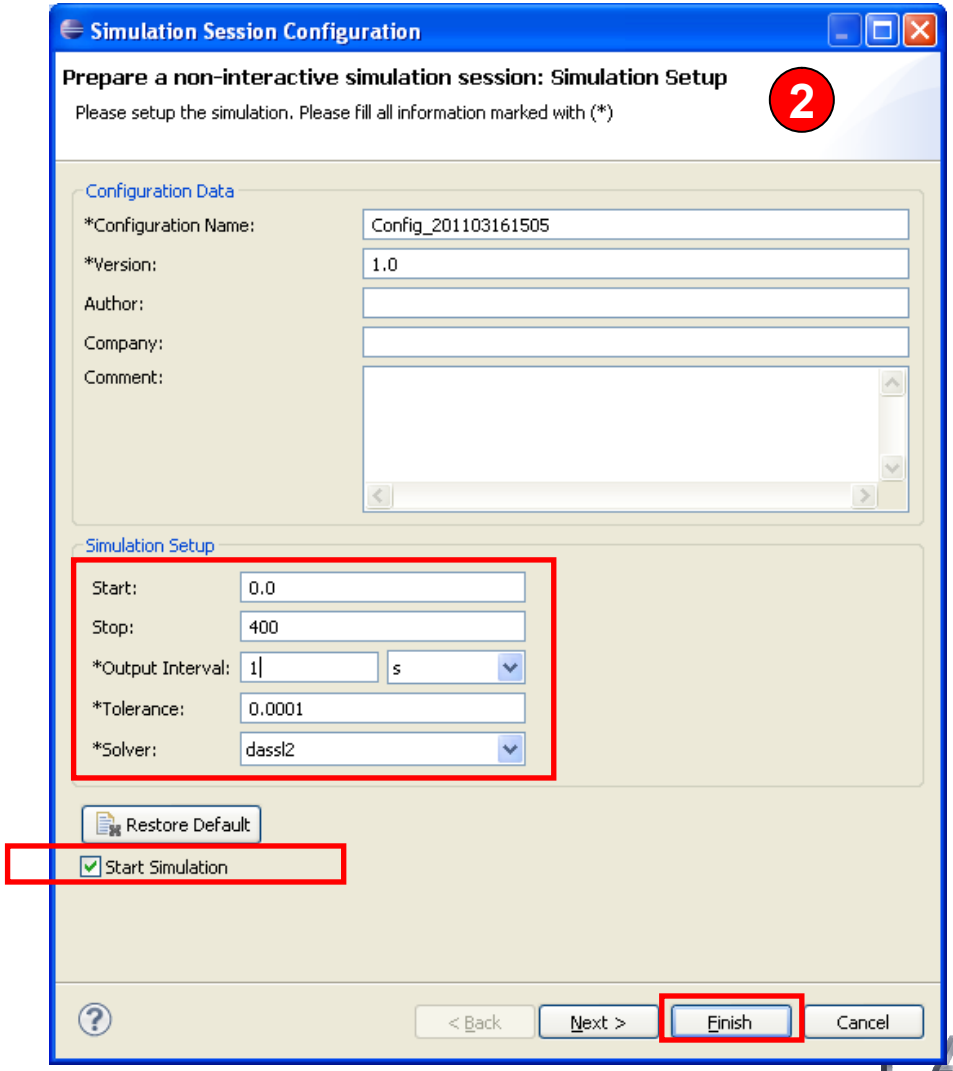


Simulate Model

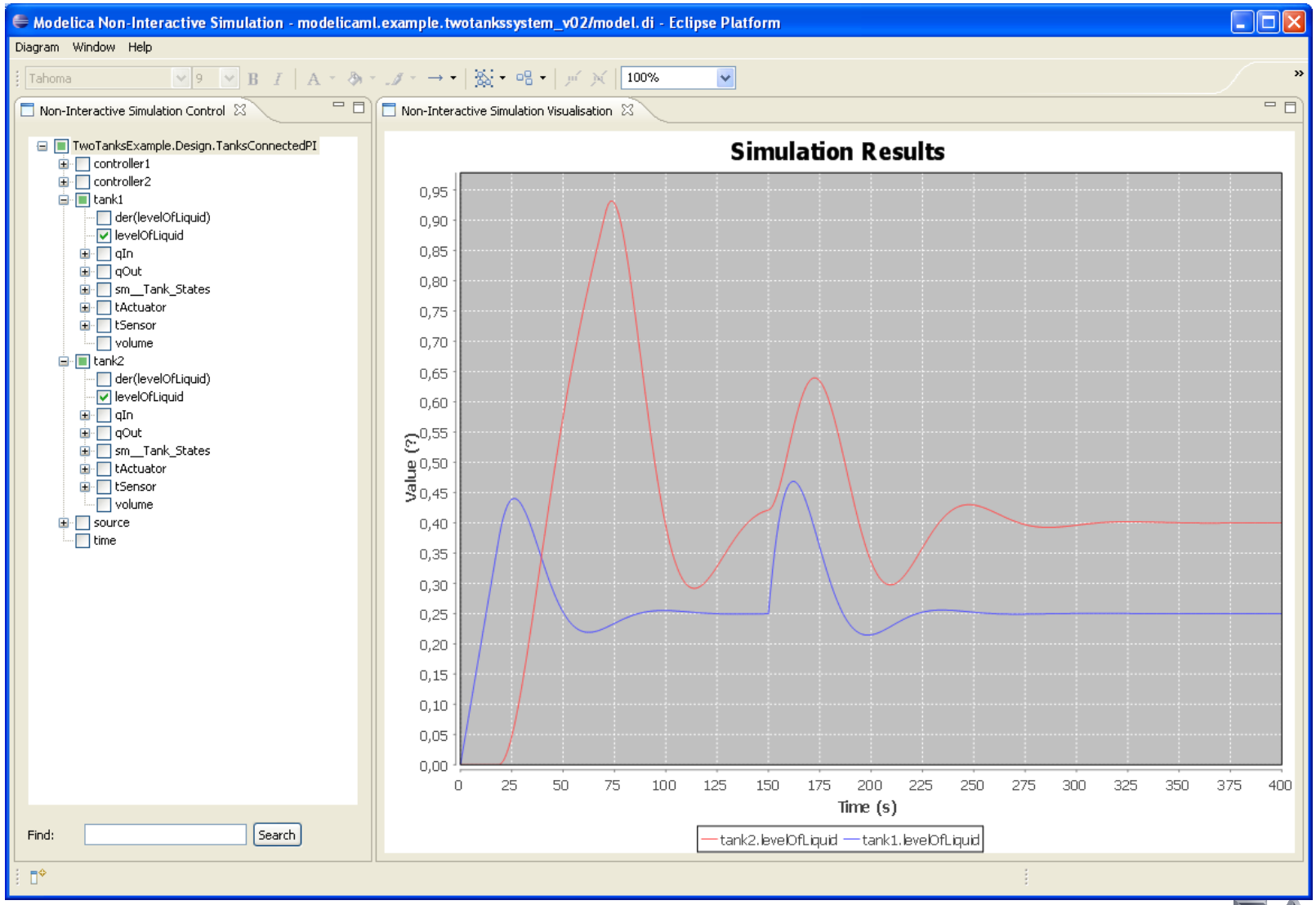


Simulate Model



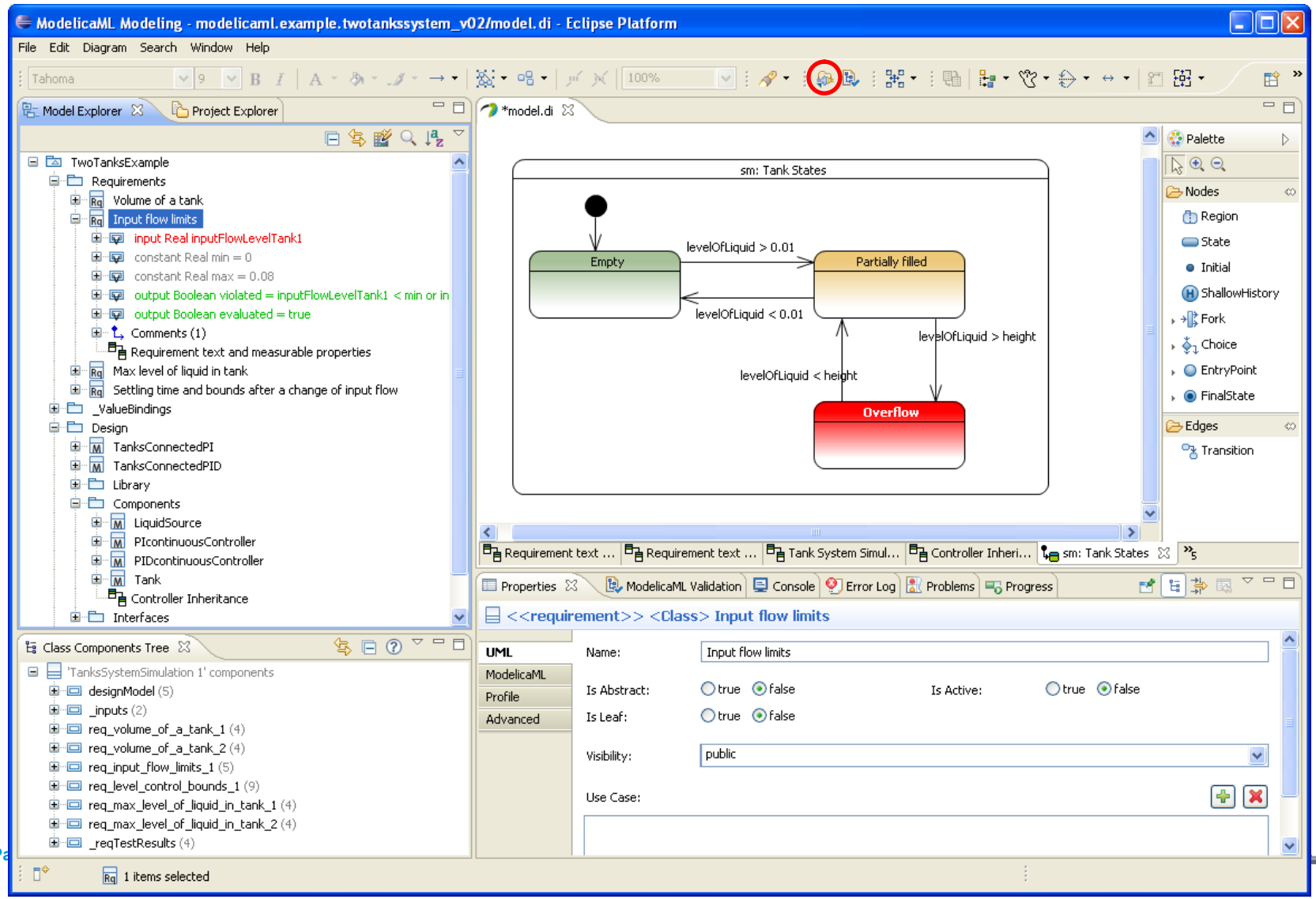


Simulate Model

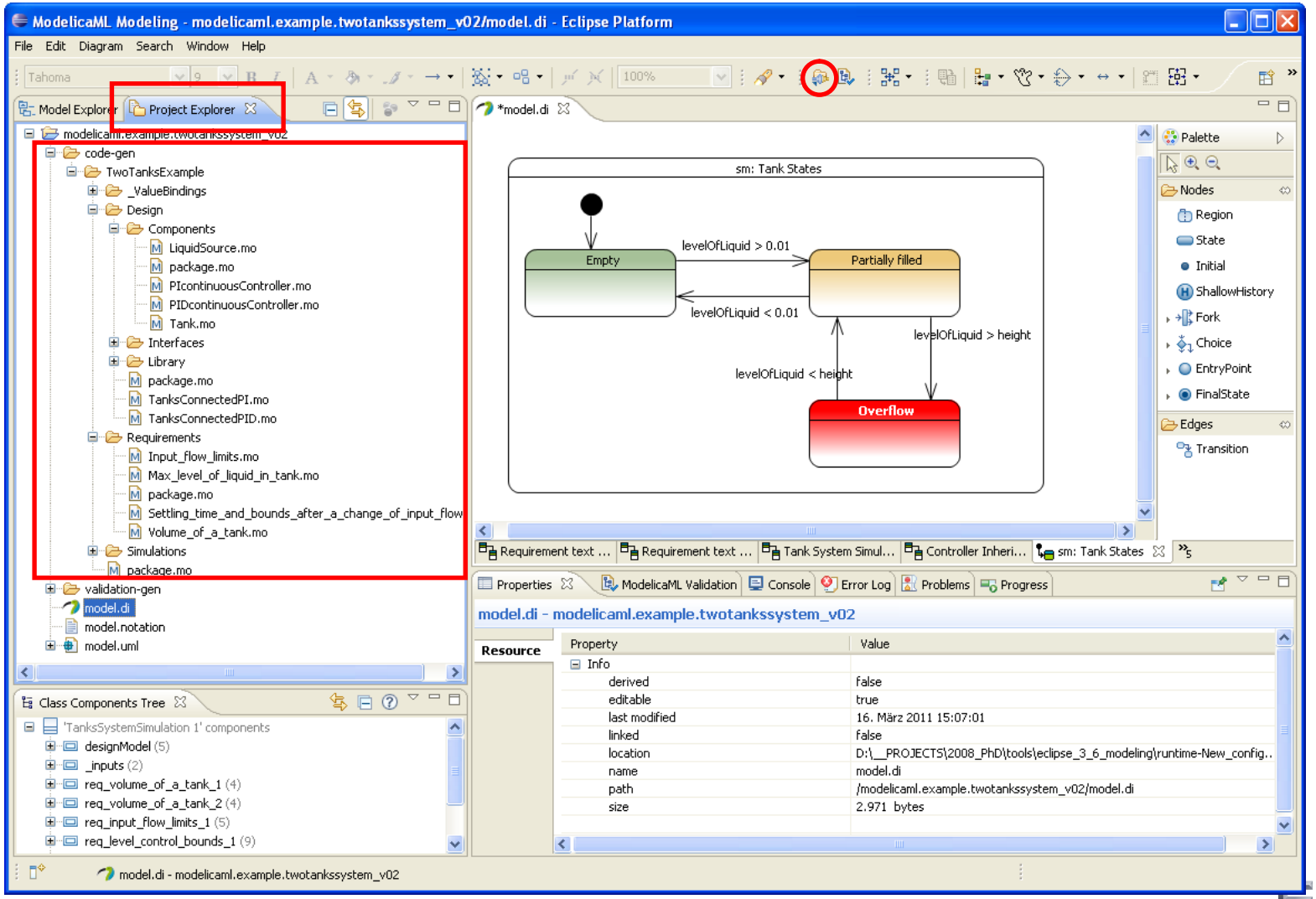


Modelica Code Generation

Launch Modelica Code Generation



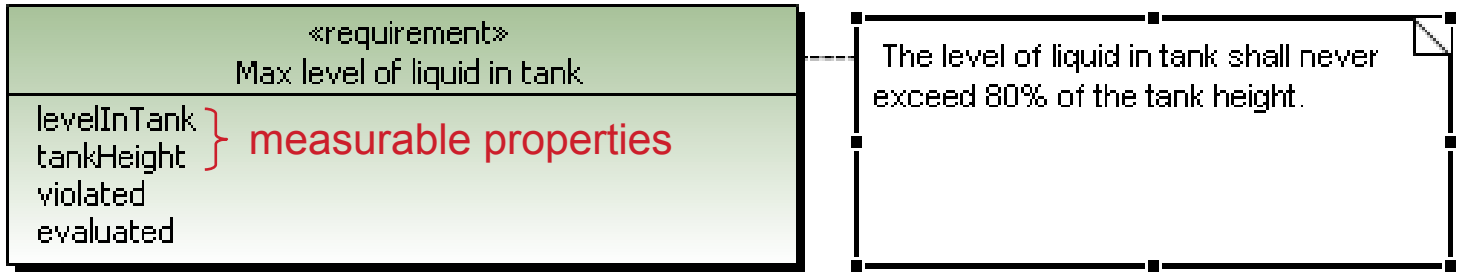
Generated Modelica Code

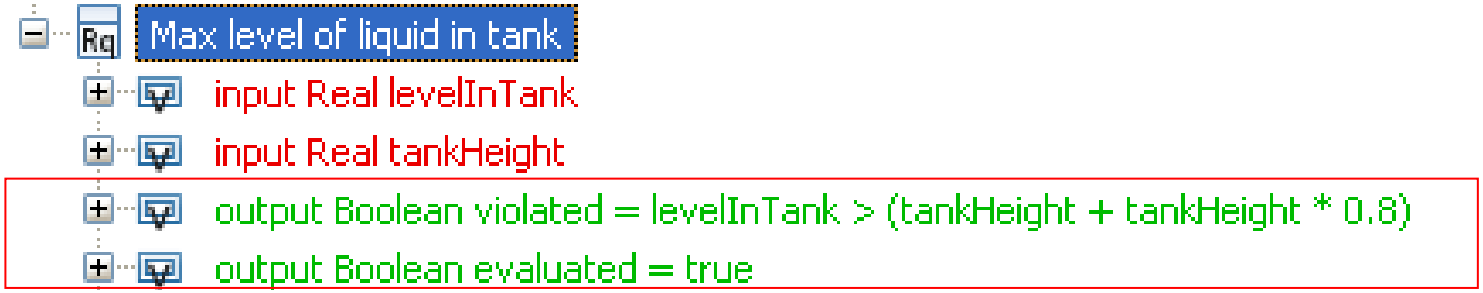


Handling of System Requirements

Formalize a Textual Requirements (Example 1)

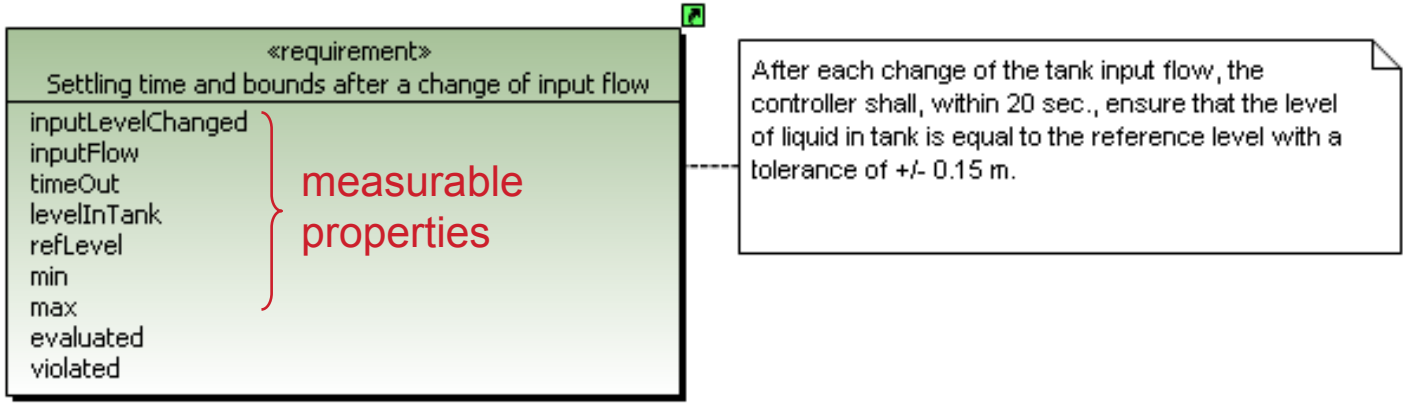
1. Identify and formalize measurable properties addressed in the requirement statement
2. Define when requirement is evaluated (pre-condition) and violated

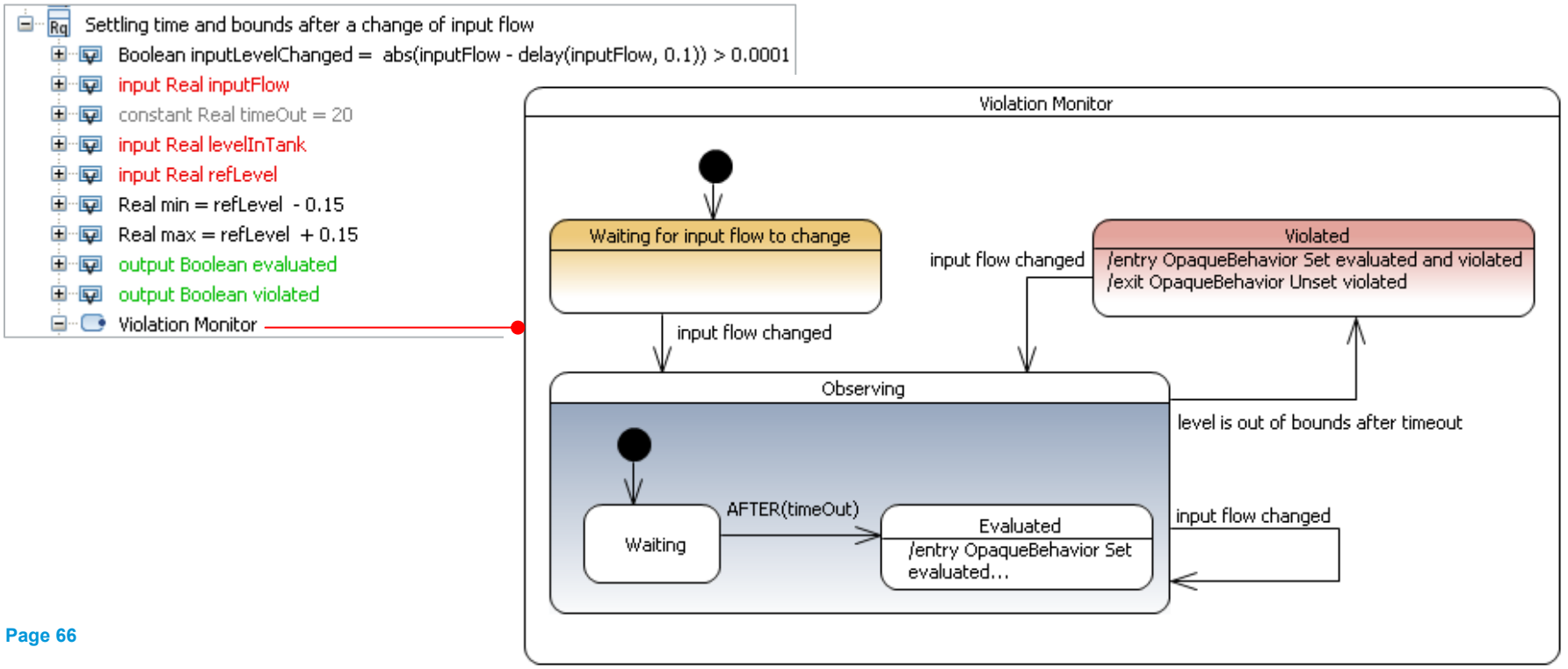




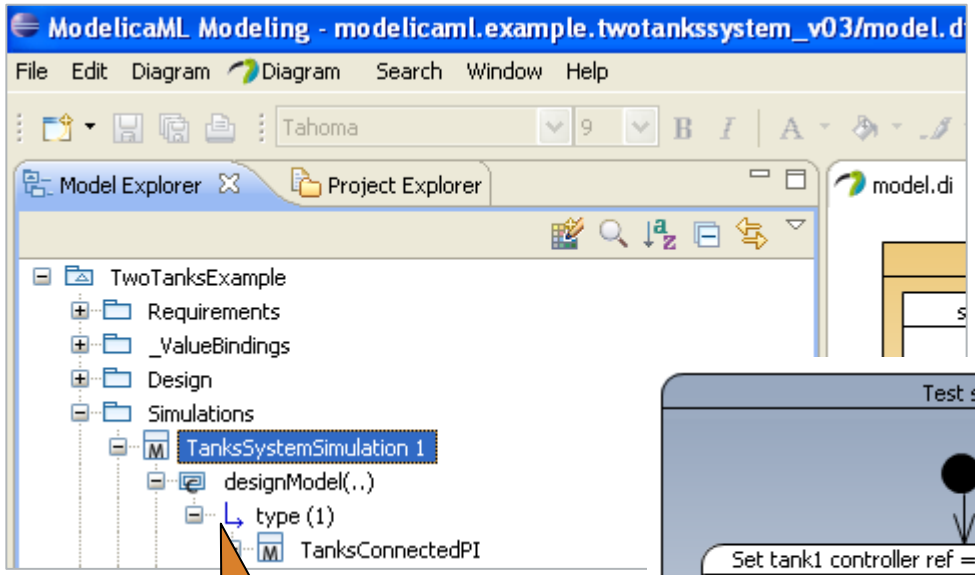
EADS

Formalize a Textual Requirements (Example 2)



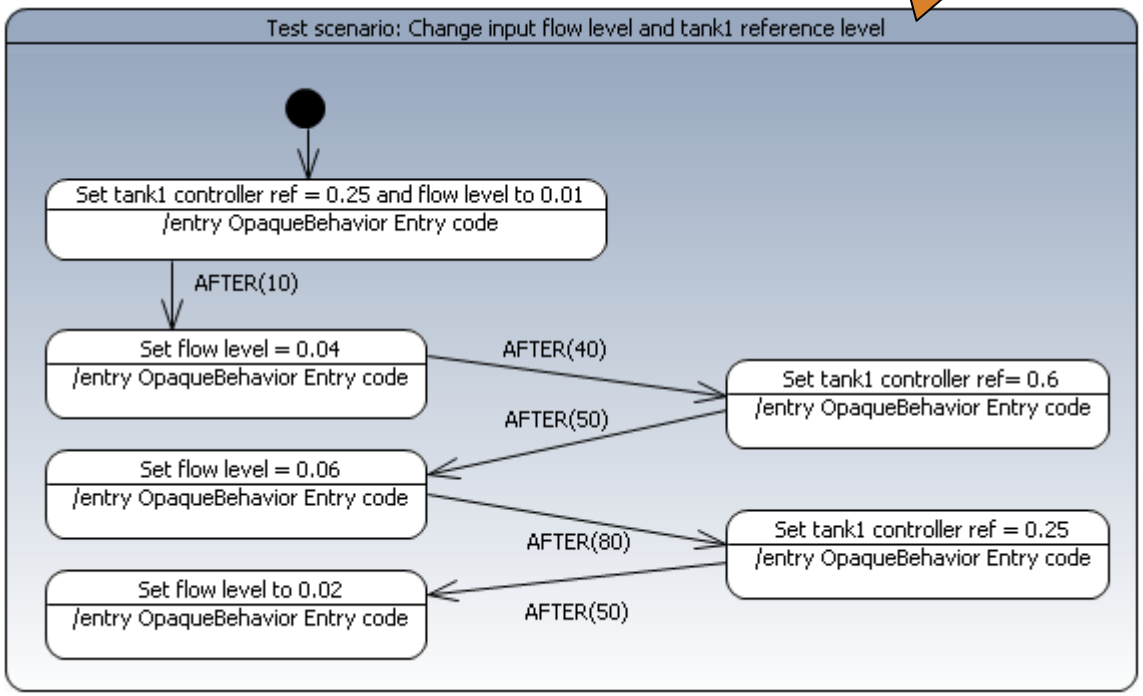


Instantiate Design Model

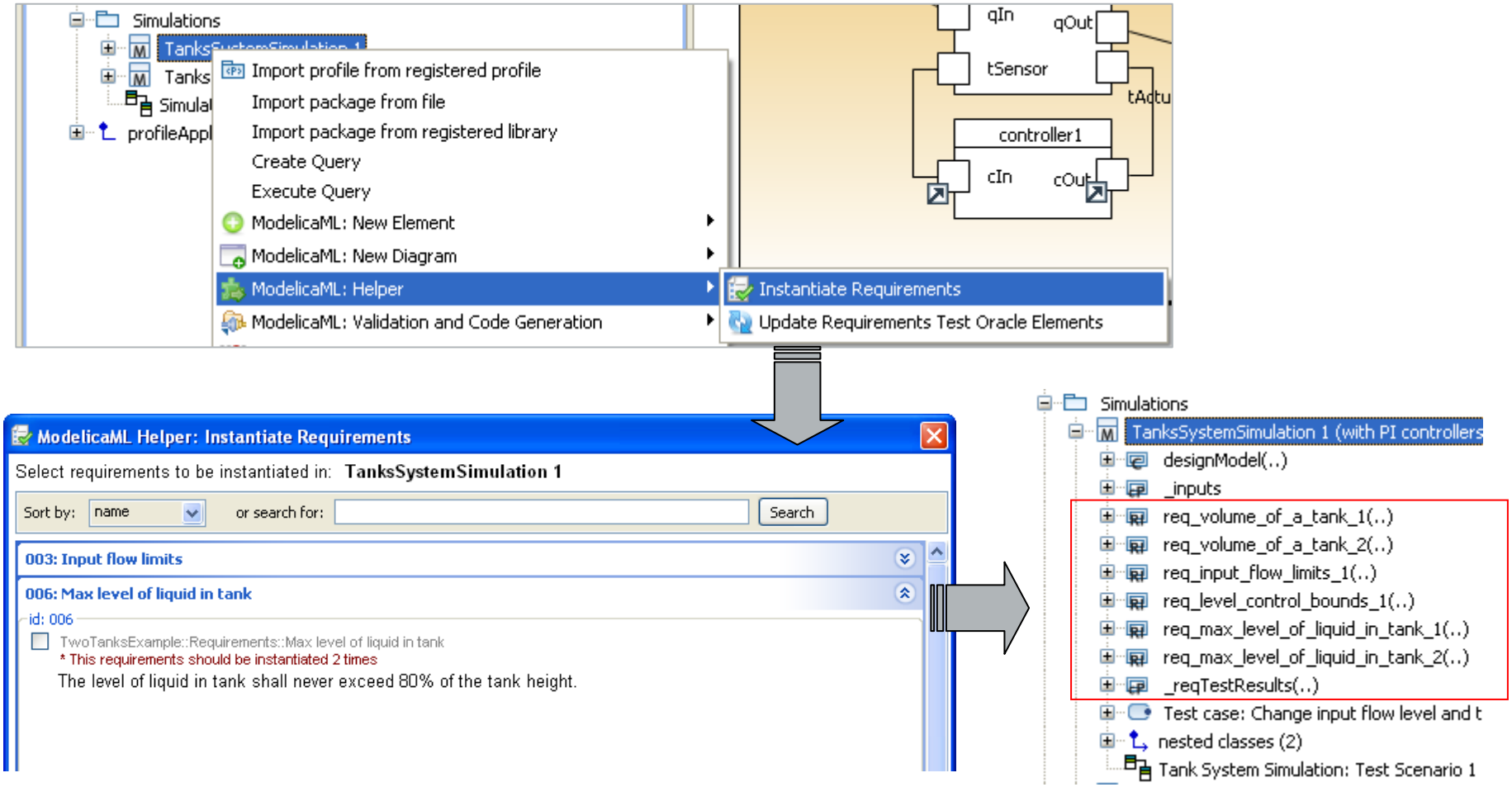


instantiated design model

Test scenario that stimulates the design model



Instantiate Requirements



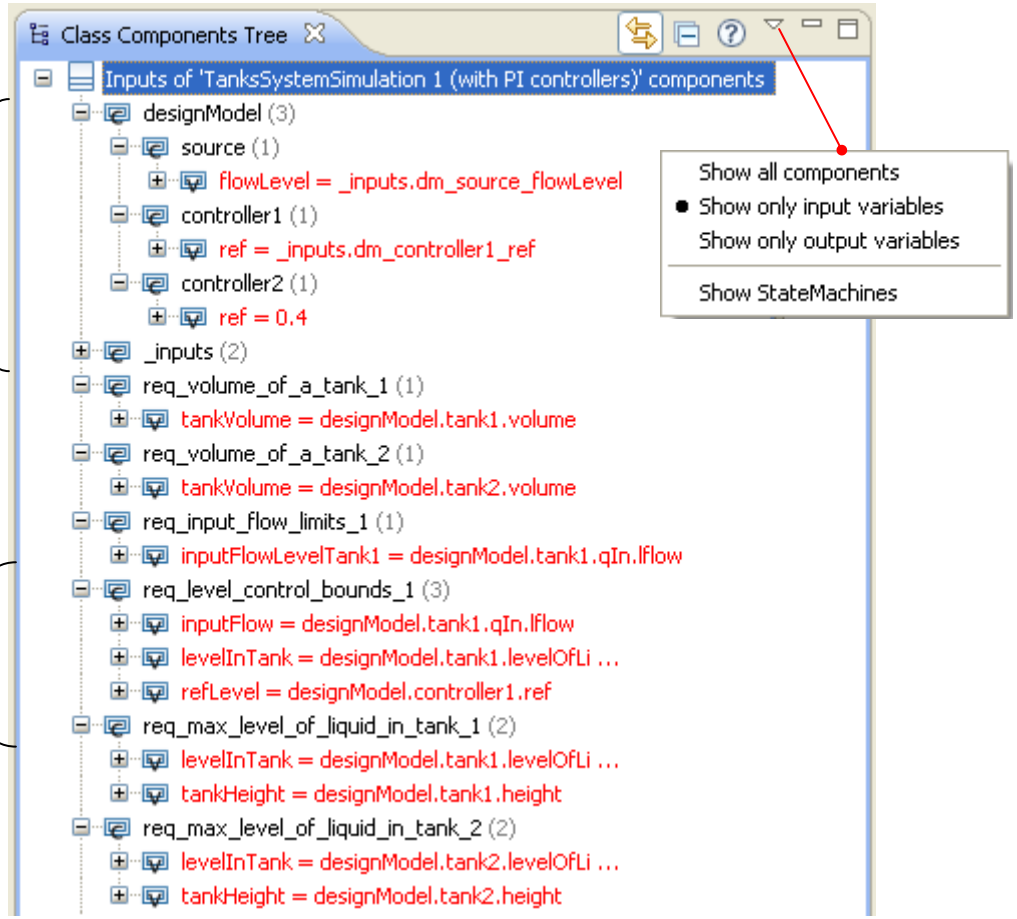
Bind Design Inputs and Requirement Properties

Stimuli and parameters for design model:

```
controller1.ref = _inputs.dm_controller1_ref
source.flowLevel = _inputs.dm_source_flowLevel
```

Example for requirement properties binding:

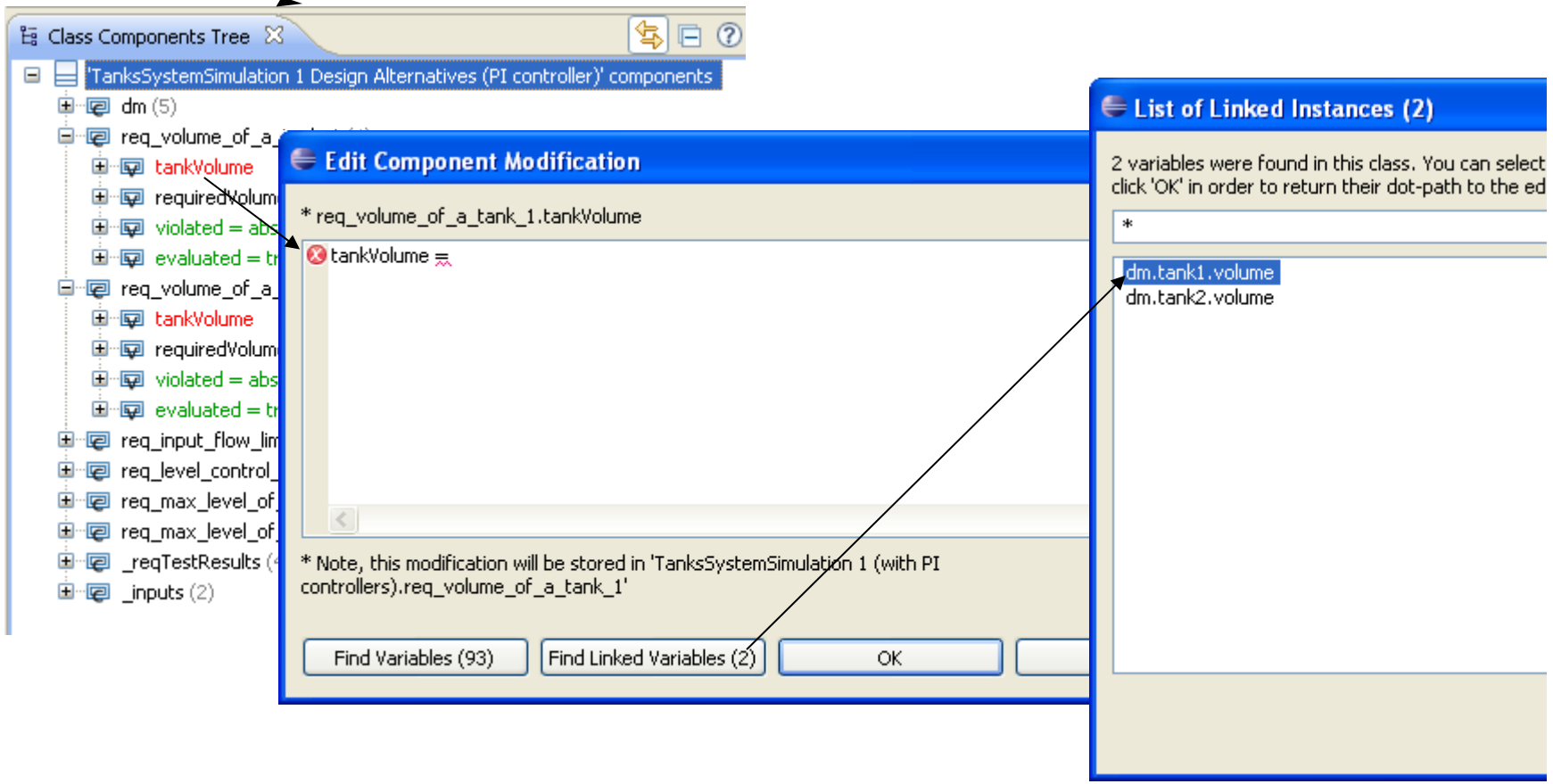
```
inputFlow = designModel.tank1.qIn.lflow
levelInTank =     designModel.tank1.levelOfLiquid
refLevel =    designModel.controller1.ref
```



EADS

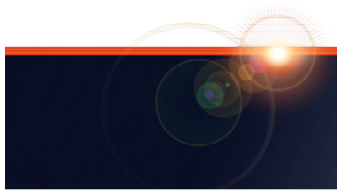
Finding Properties that are Linked Through Mediators

Instantiates a class and enables the finding of different properties that are linked through mediators



EADS

Using Mediators for Value Binding



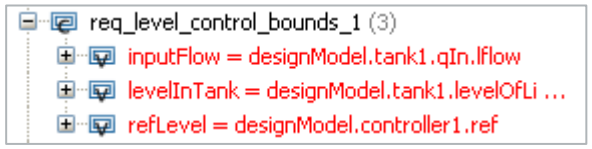
Bind Design Inputs and Requirement Properties

Challenge:

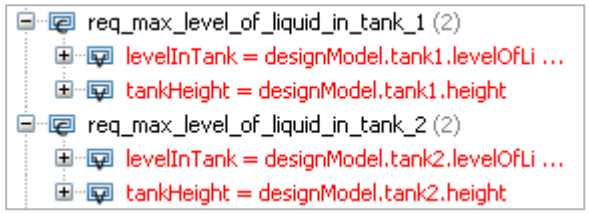
- When requirements and design are instantiated in a test model for simulations their properties need to be bound (i.e. requirement properties must access the right values from the design model)
- Is it possible to support or even automate the binding?

Questions:

- How to find the right design model properties, that requirements properties shall be bound to, without having a deep knowledge of the design model?



- A requirement may be needed to be instantiated multiple times (e.g. for each tank or controller). How to know how many times a requirement should be instantiated and what are the respective bindings to the design model properties?



Bind Design Inputs and Requirement Properties

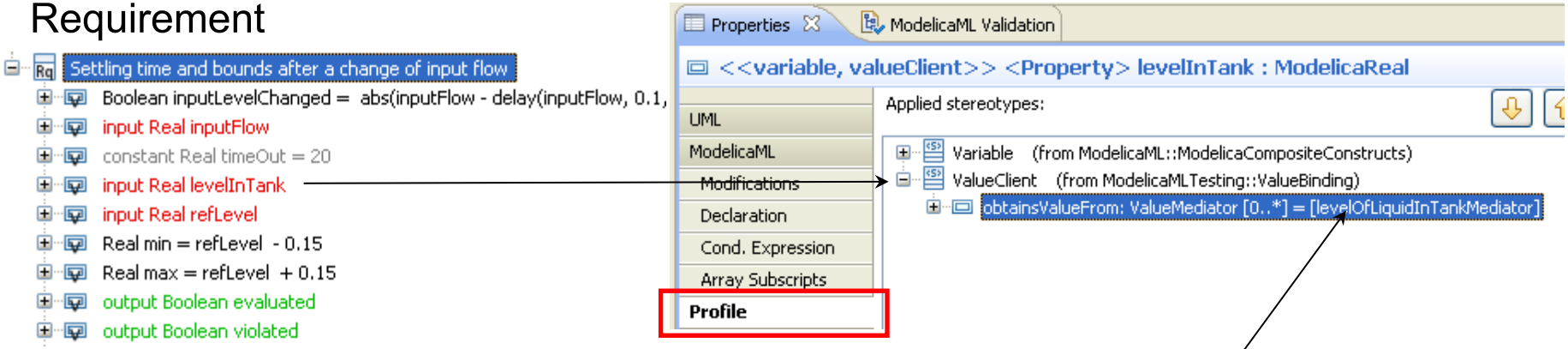
Mediators Concept:

- ValueMediator is a ModelicaML concept that allows to define a n:n relation between properties of different classes.
  - This is done at the class property level and enables the finding of linked properties when they are instantiated in the same class hierarchy (e.g. in the same simulation model)
  - ValueMediator are only used at modeling level and do not affect the code generation
- ValueProvider provides the value
- ValueClient obtains the value

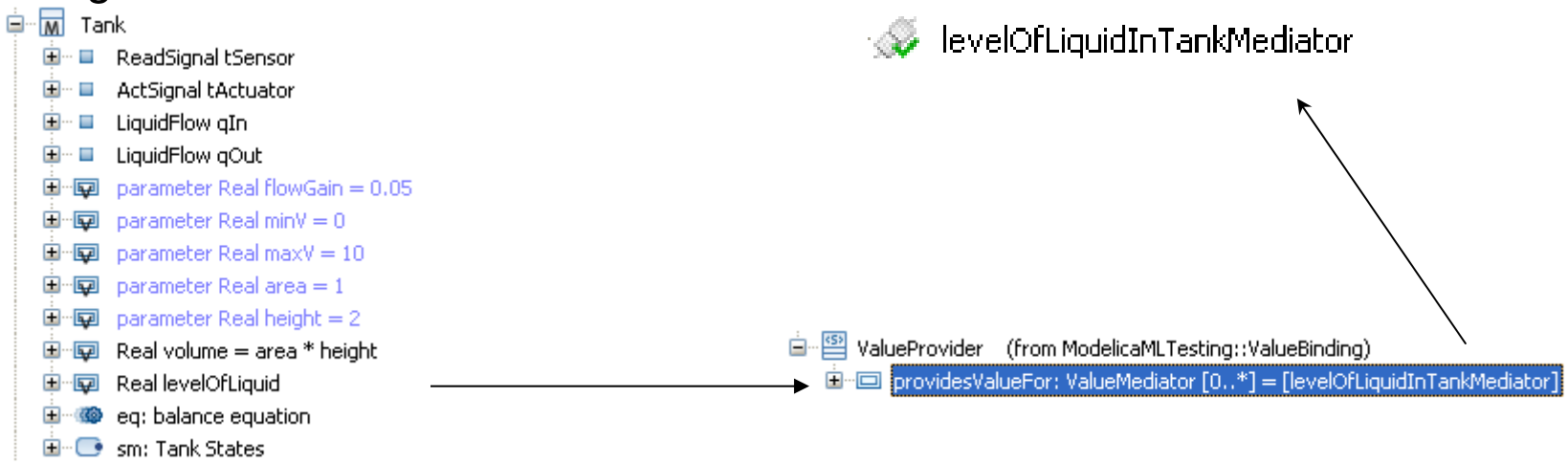
- When formalizing requirements the requirements analyst creates and reuses ValueMediators and defines which requirement property should obtain the value from which ValueMediator
- When creating the design model (based on requirements) the designer defines which design model properties provide value for which ValueMediator

Using Mediators

Requirement



Design Artifact



Why Using Mediators?

Support for the access of design properties values:

- ValueMediator is an interfaces the design model provides in order to enable support for requirements properties to the right design properties.

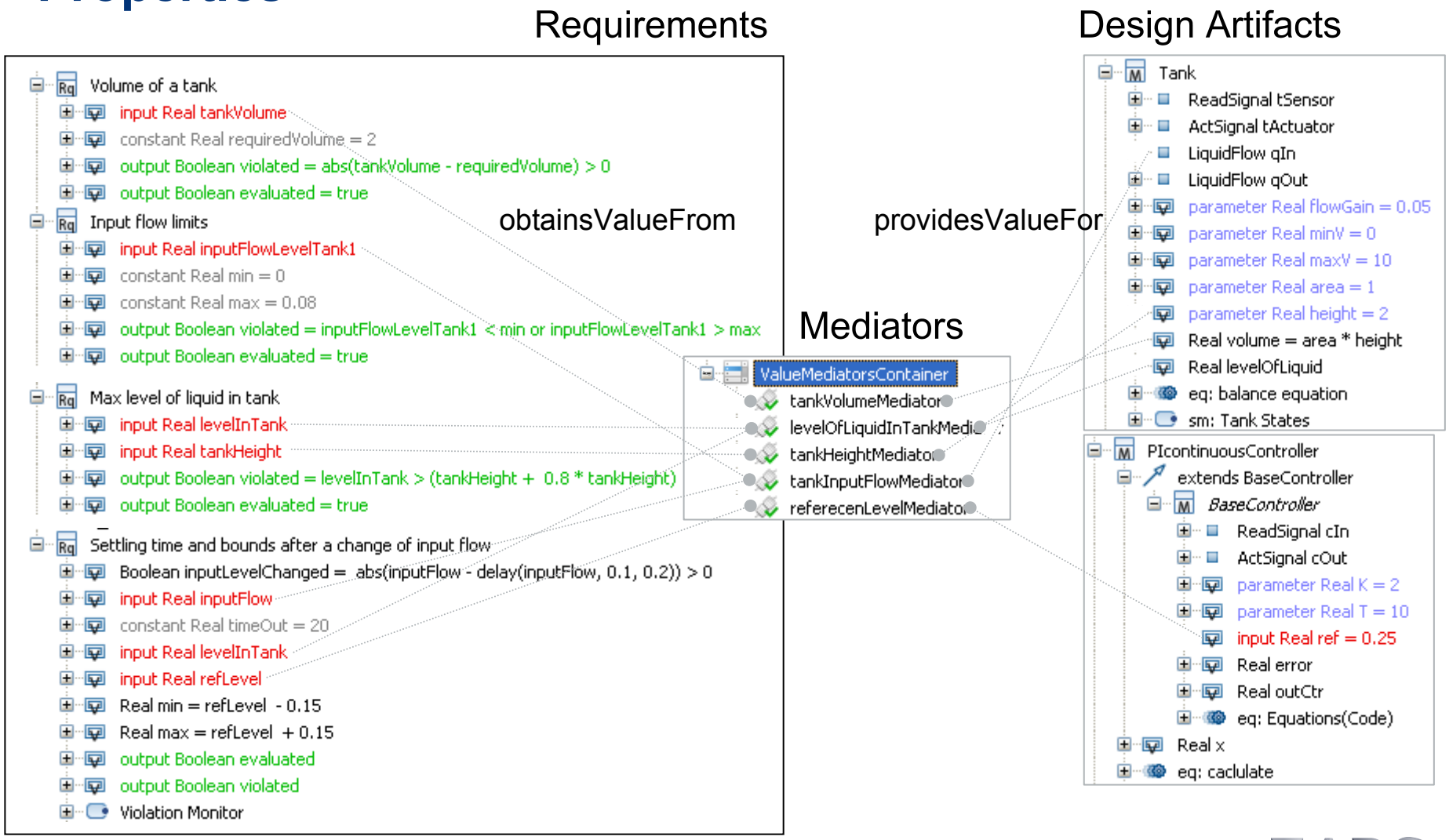
Handling of properties that are mentioned in multiple requirements

- ValueMediator concept is beneficial because it enables a consistent list of same properties that are reused in multiple requirements

Design Model Alternatives Handling

- ValueMediator concept is used to encapsulate the design model alternatives. Multiple design alternatives provide the value for the same ValueMediator.

Traceability Between Requirements and Design Properties



EADS

Finding the properties that are linked through mediators

Instantiates a class and enables the finding of different properties that are linked through mediators