Bindings: a New Approach for co-simulation applied to System Design Verification Against Requirements

Yulu Dong

Daniel Bouskela

Thuy Nguyen

Audrey Jardin

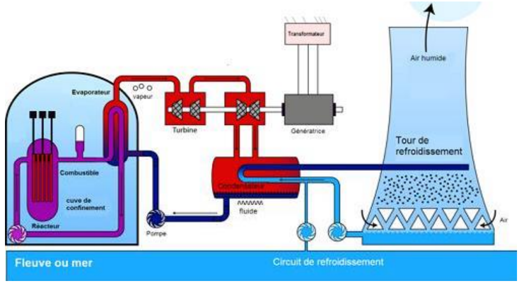
EDF R&D

Problem: how to efficiently verify complex system design against requirements ?

- Complex physical systems exhibit large numbers of
  - dynamic continuous states: physical states (temperature, pressure, mass flow…)
  - dynamic discrete states: operating modes (normal, dysfunctional, I&C…)
  - described by multiple models from different modelling tools

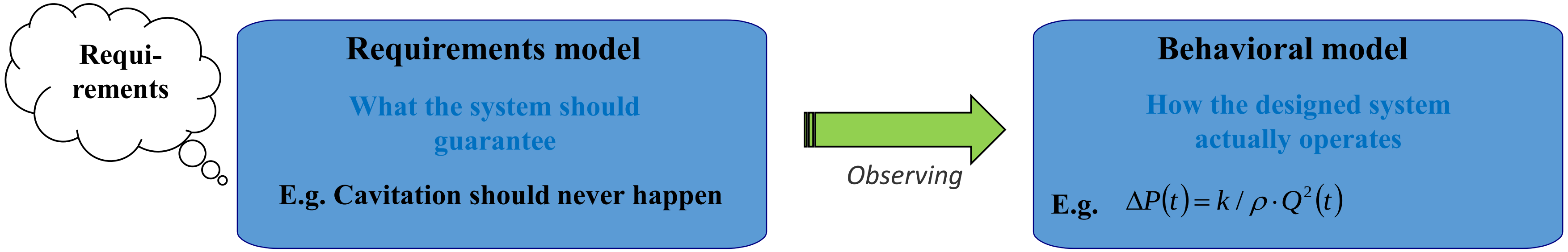
They are subject to large numbers of requirements
  - And need many test scenarios for proper verification

- Difficulty: possible combinatorial explosion of situations to be explored.



Solution: automate as much as possible the testing of complex systems

- How:
  - Build a formal model of the requirements
  - Build a physical model of the behavior of the system
  - Generate automatically test scenarios that comply with the assumptions made and the test coverage criteria
  - Use the requirements model as an observer of the behavioral model to automatically detect possible violations of the requirements



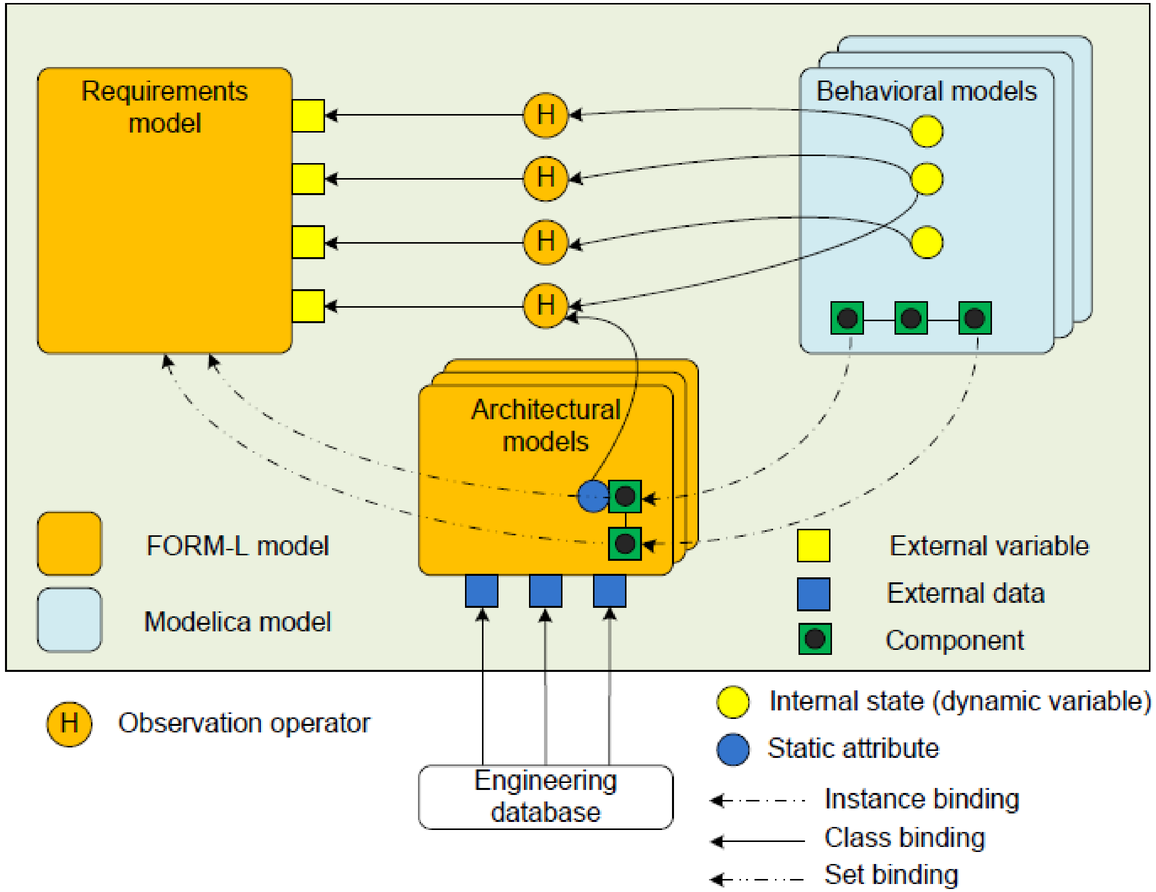
Purpose of this presentation: a method for automatic binding of models and a Python implementation

- The requirement model uses external variables that are computed by other models (so-called external models). The links between the external variables and the variables of the external models are called bindings.
- Problem to be solved:

  The bindings should be generated automatically because large numbers of external variables are involved

- A method for the automatic generation of bindings is presented.
  - The method is independent of any modelling language
  - The current implementation uses Modelica for the requirements and the behavioral models

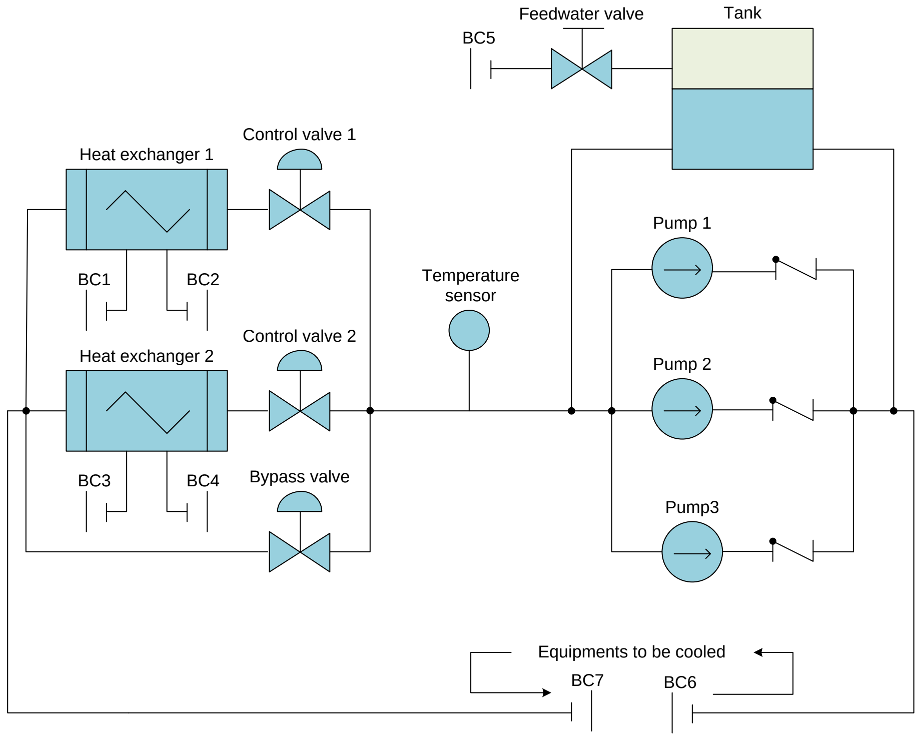
- The implementation is done with a Python script.



Modeling architecture outline

Example: cooling system of auxiliary equipment of a power plant

- Objective of the cooling system: remove heat from large equipment units of a power plant.
- System is critical for plant availability: in case of failure, the plant must be shutdown.

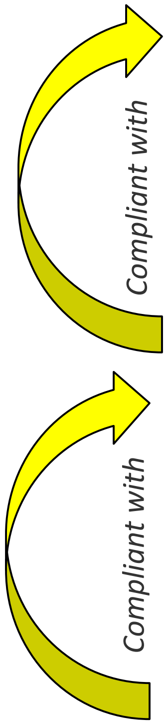


Architectural model of the cooling system (ThermoSysPro, EDF)

Cooling system example: informal requirements

- Req. 1: ‘The maximum heating power to be removed is 25 MW’.
- Req. 2: ‘When they are in operation, pumps shall not cavitate’.
- Req. 3: ‘The water temperature shall not vary by more than 10°C per hour’.
- Req. 4: ‘When the system is in operation, there should be no less than two pumps in operation for more than 2 seconds. Violation shall not occur more than 3 times per year with a confidence rate of [a given percentage]”.
- etc. (other examples in the paper)

Building the verification model: basic principles



1. Formalize requirements using a **property modeling** language ➔ requirements model (with ReqSysPro developed at EDF R&D)

   Expert: operation engineer

2. Describe system design using the **same property modeling** language ➔ architecture model (with ThermoSysPro developed at EDF R&D)

   Expert: mechanical and system control engineer

3. Model dynamic physical behavior of the system using mathematical modeling ➔ behavioral model (with ThermoSysPro developed at EDF R&D)

   Expert: physicist (thermal-hydraulics, neutronics, combustion, etc.)

Binding between the requirements model and the architecture model (Set bindings)

‘When they are in operation, pumps should not cavitate’.

```
propertyModel tPump_R
  String type = “Pump”;

  external Boolean Cavitate;
  external Boolean InOperation;
end tPump_R;
```

Pump definition for requirement model

```
propertyModel CoolingSystem_R
    external tPump_R Pumps = { };
    requirement R2 = check card ( { P in Pumps
        | P.InOperation and P.Cavitate } ) == 0;
end CoolingSystem_R;
```

Requirement model for (Req. 2)

WHERE /
WHEN /
WHAT

```
propertyModel tPump_A
  String type = “Pump”;
  String subtype;
  String id;
  external Real[:] NPSH_req;
end tPump_A;
```

Pump definition for architecture model

```
propertyModel CoolingSystem_A1
    tPump_A Pump1 (subtype=”Centrifugal”,  id=”PO1”);
    tPump_A Pump2 (subtype=”Centrifugal”,  id=”PO2”);
    tPump_A Pump3 (subtype=”Centrifugal”,  id=”PO3”);
  end CoolingSystem_A1;
```

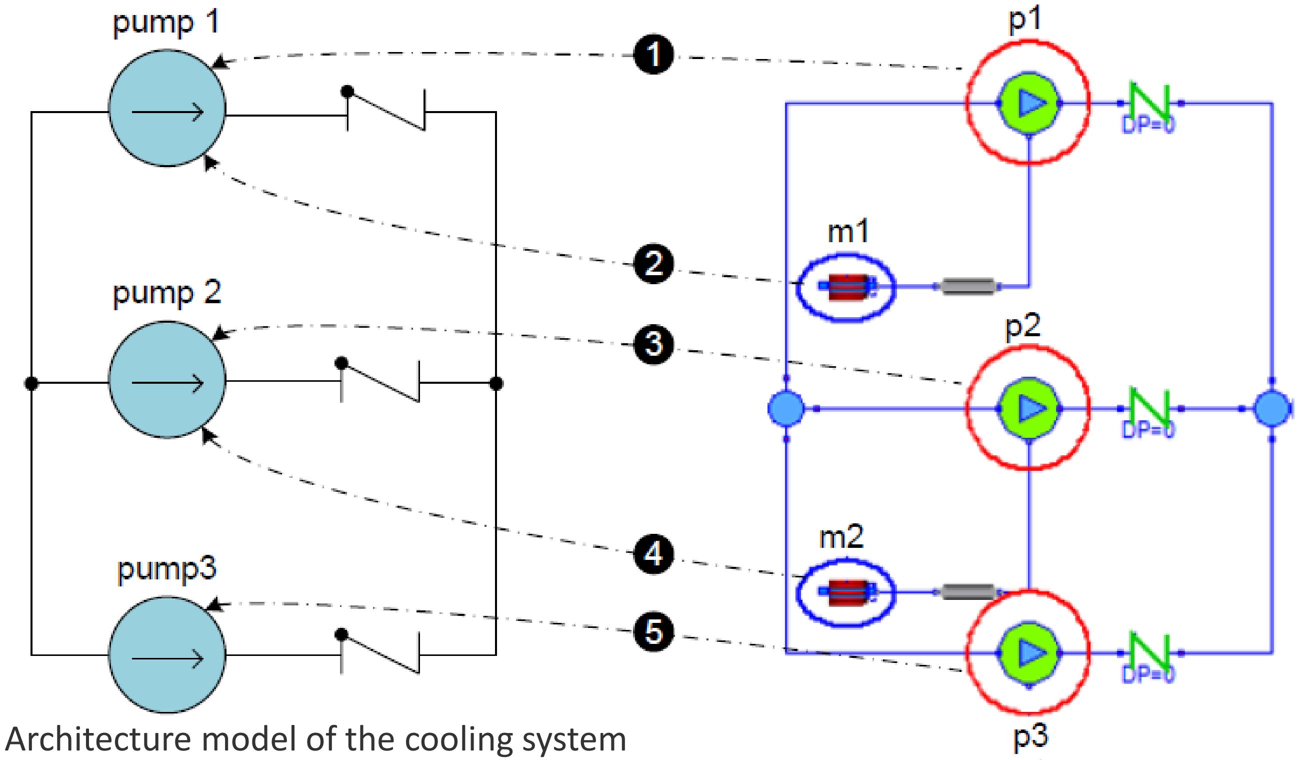
Partial architecture model for (Req. 2)

```
binding CoolingSystem_R1
  bind_set [s1] (CoolingSystem_R.pumps = { CoolingSystem_A1.Pump_A });
  …
end CoolingSystem_R1;
```

```
binding CoolingSystem_R2
  bind_set  (CoolingSystem_R.pumps = { CoolingSystem_A1.pump1, CoolingSystem_A1.pump2 });
  …
end CoolingSystem_R2;
```

Binding between the architecture model and the behavioral model (Instance bindings)

'When they are in operation, pumps should not cavitate'.



Architecture model of the cooling system

Behavioral model of the cooling system

```
binding CoolingSystem_A1
  bind_instance i1 (CoolingSystem_A1.pump1={ CoolingSystem_B1.p1, CoolingSystem_B1.m1 });
  bind_instance i2 (CoolingSystem_A1.pump2={ CoolingSystem_B1.p2, CoolingSystem_B1.m2 });
  bind_instance i3 (CoolingSystem_A1.pump3={ CoolingSystem_B1.p3 });

  i1.role (CoolingSystem_B1.m1) = "inOperation";
  i2.role (CoolingSystem_B1.m2) = "inOperation";
  i3.role (CoolingSystem_B1.p3) = "inOperation";
end CoolingSystem_A1;
```

Binding between the requirements model and the observation operators (Variable bindings)

```
binding Pump_R1
  bind_variable v1 (Pump_R.inOperation = [ Obs_PumpStarted_1, Obs_PumpStarted_2,
                                            Obs_PumpStarted_3 ]);
  bind_variable [v2] (Pump_R.cavitate = [ Obs_PumpCavitating ]);
  v1.role = "inOperation";
  ...
end Pump_R1;
```

'When they are in operation, pumps should not cavitate'.

```
propertyModel tPump_R
  String type = "Pump";

  external Boolean Cavitate;
  external Boolean InOperation;
end tPump_R;
```

Pump definition for requirement model

```
propertyModel CoolingSystem_R
    external tPump_R Pumps = { };
    requirement R2 = check card ( { P in Pumps
          | P.InOperation and P.Cavitate } ) == 0;
end CoolingSystem_R;
```

Requirement model for (Req. 2)

WHERE / WHEN / WHAT

```
function Obs_PumpCavitating
  input Real P "Pressure at the inlet of the pump";
  input Real q "Volumetric flow through the pump";
  input NPSH_req "Required NPSH";
  output Boolean pumpCavitating "Boolean telling whether the pump is cavitating or not";
algorithm
  pumpCavitating := (P < NPSH_req[q]);
end Obs_PumpCavitating;
```

Observation operator for pump cavitation

```
function Obs_PumpStarted_1
  input Real V "Supply voltage";
  output Boolean pumpStarted "Boolean telling whether the pump is started or not";
algorithm
  pumpStarted := (V > 0);    // Threshold is zero
end Obs_PumpStarted_1;
```

Observation operator for pump in operation

Binding between the observation operators and the behavioral model (Input bindings)

'When they are in operation, pumps should not cavitate'.

```
binding PowerPlant_Lib
  bind_input [i1] (Obs_PumpStarted_1 = (V = ElectricMotor.V));
  bind_input [i2] (Obs_PumpStarted_2 = (Cm = CentrifugalPump.Cm));
  bind_input [i3] (Obs_PumpStarted_3 = (q = CentrifugalPump.q));
  bind_input [i4] (Obs_PumpCavitating = (P = CentrifugalPump.Pin, q = CentrifugalPump.q,
                                          NPSH_req = Pump_A.NPSH_req);
  …
end PowerPlant_Lib;
```

```
class CentrifugalPump
  Real Cm "Motor torque";
  Real omega "Angular velocity of the rotor";
  Real Pin "Pressure at the inlet";
  Real q "Volumetric flow rate";
 equation
...
end CentrifugalPump;
```

```
class Pump_A
  String id;
  external Real NPSH_req[:, 2];
end Pump_A;
```

Behavioral model of the cooling system

```
function Obs_PumpCavitating
  input Real P "Pressure at the inlet of the pump";
  input Real q "Volumetric flow through the pump";
  input NPSH_req "Required NPSH";
  output Boolean pumpCavitating "Boolean telling whether the pump is cavitating or not";
algorithm
  pumpCavitating := (P < NPSH_req[q]);
end Obs_PumpCavitating;
```

Observation operator for pump cavitation

```
function Obs_PumpStarted_1
  input Real V "Supply voltage";
  output Boolean pumpStarted "Boolean telling whether the pump is started or not";
algorithm
  pumpStarted := (V > 0);     // Threshold is zero
end Obs_PumpStarted_1;
```

Observation operator for pump in operation

Binding algorithm

'When they are in operation, pumps should not cavitate'.

Input data of algorithm:

a. Set binding O.s 🡐 { p1, …, pn }
b. Variable binding R.y 🡐 [H1, …, Hr]
c. Input binding Hi (u1, …, un) 🡐 (u1=E1.x, …, un=En.x) for i = 1 to i = r
d. Instance binding = p 🡐{ e1, …, en } for all p in target(O.s 🡐{ p1, …, pn })

<u>Let us consider:</u>

a. an external variable y declared in a class R: R.y,

b. an external set s of objects of type R declared in an object O: O.s

<u>Algorithm:</u>

For each object p in the set {p1, p2, …, pn}:

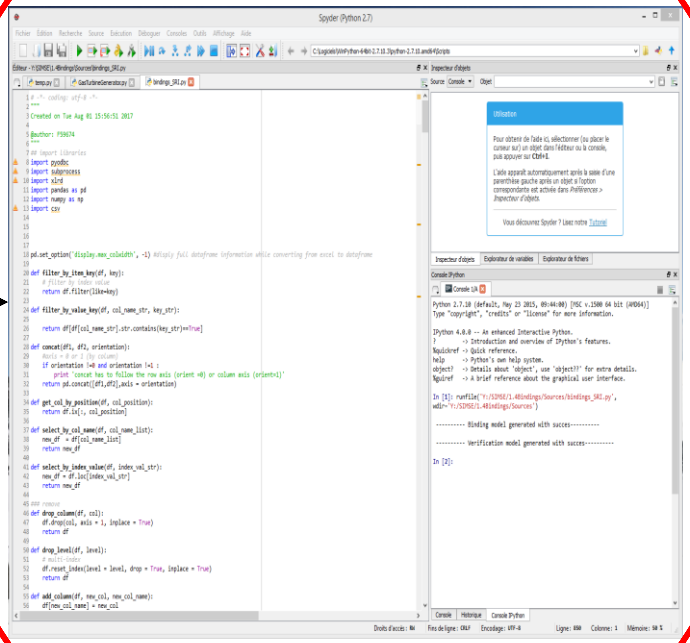
1. Find H of lowest rank in R.y 🡐 [H1, …, Hr] such that sig(H) ⊂ sig(p 🡐 { e1, …, en }). The result is Hy.
2. From the binding Hy (u1, …, un) 🡐 (u1=E1.x, …, un=En.x), form symbolically the expression p.y = Hy (u1=E1.x, …, un=En.x).
3. For each Ei in { E1, …, En }, assume that there's a unique instance ei in the target of p 🡐 { e1, …, en } such that class(ei) = Ei. Keep ei.
4. Replace in p.y = Hy (u1=E1.x, …, un=En.x) the classes Ei by the found instances ei. The result is p.y = Hy (e1.x, …, en.x).

Binding programming

Coding language: Python. Why: interpreted language, open source libraries, easy access to data base such as Excel, SQL, …

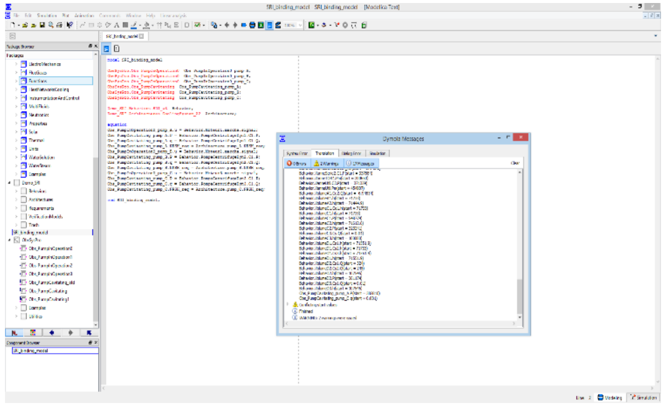
Data base

- Set binding
- Instance binding
- Variable binding
- Input binding

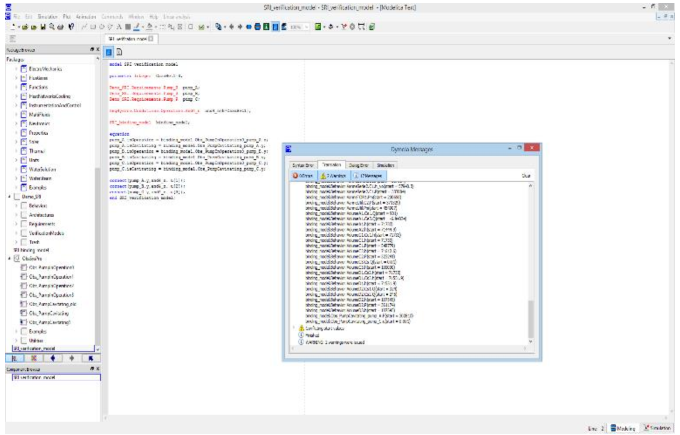


Python code: binding.py

Automatic genaration



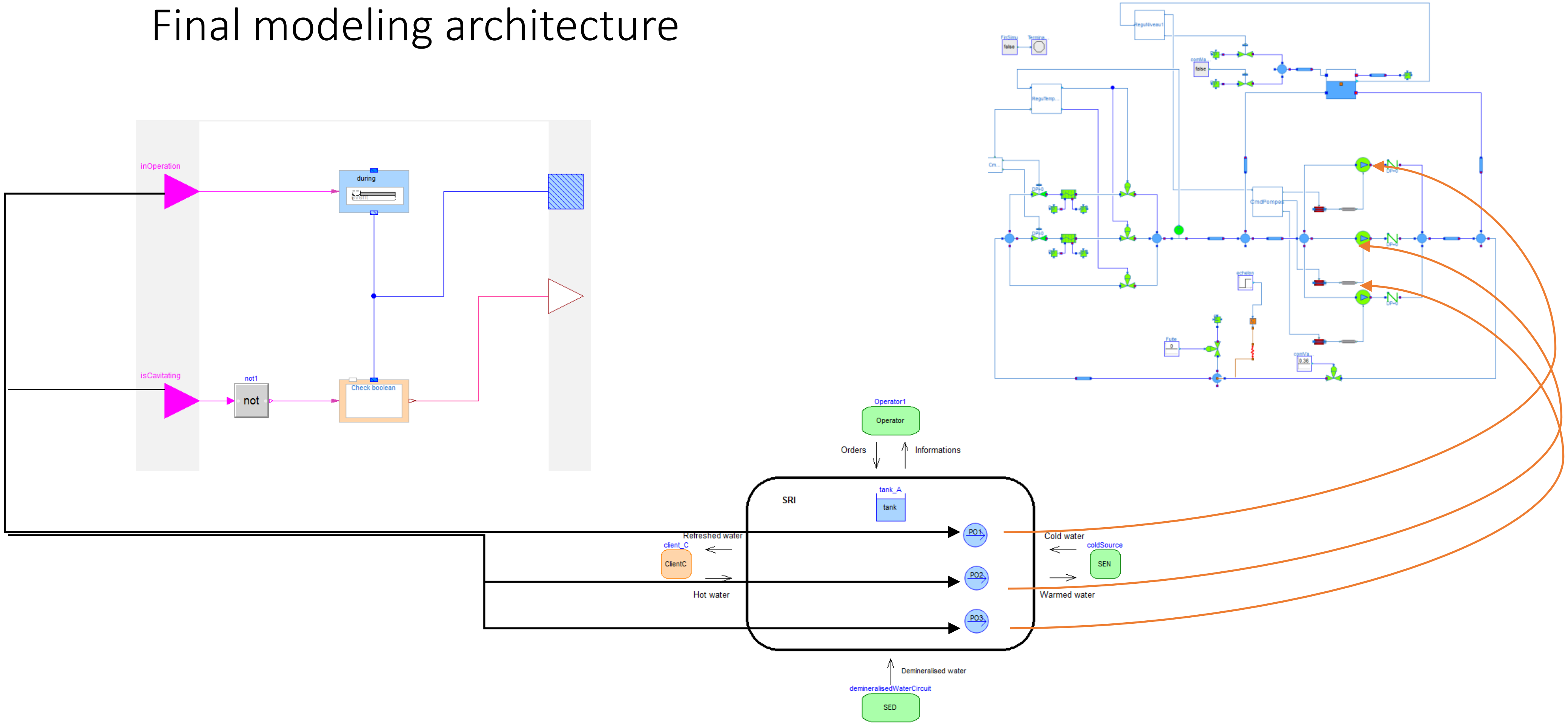
binding_model.mo



verification_model.mo

Ready to be simulated

Final modeling architecture



- Translation from FORM-L to Modelica needs new translator
- Modelica models may be compiled and run with existing open source and commercial tools

Conclusion and further perspectives

Co-simulation is largely used in complex cyber-physical systems: power plants, aircraft, automobiles, etc. Many factors are taken into account in the modelling & simulation phase: safety & security analysis, rare events and risk analysis, operation & maintenance, functional requirements, socio-economic analysis, … All these factors can be modelled by appropriate tools and co-simulated.