EOOLT 2013 - 5th International Workshop on
Equation-Based Object-Oriented
Modeling Languages and Tools
April 19th, 2013 - University of Nottingham, Nottingham, UK

Modeling System Requirements in Modelica: Definition and Comparison of Candidate Approaches

Andrea Tundis, Alfredo Garro

Systems Engineering and Integration (SEI) Research Group

Department of Computer Engineering, Modeling, Electronics, and System Sciences (DIMES)

University of Calabria – ITALY

Peter Fritzson, Lena Buffoni-Rogovchenko

Programming Environment Laboratory (PELAB)

Department of Computer and Information Science (IDA)

Linköping University - SWEDEN

Outline

- Introduction and Reference Context
- Motivations and Needs: Why?
- Aim of the Proposal
  - A Meta-Model for representing System Requirements
  - Approaches for modeling System Requirements in Modelica
  - A case study
- Conclusions and Future perspectives

Introduction and Context

- **Functional Safety**: the functional correctness of a component is the guarantee that the component behaves the way it should and fulfills all the functional requirements of the system in order to make it more reliable.

- **RAMS** (*Reliability*, *Availability*, *Maintainability* and *Safety*): the engineering discipline which aims at providing an integrated and methodological approach to deal with system dependability.

- **Verification&Validation**: to provide support for the *verification* and the *validation* of models of a systems engineering process in order to check the correctness of the system by verifying the simulated behavior vs. expected/intended behavior against requirements.

Motivations and Needs

Motivations:

- lack of (i) *models* to make requirements machine-readable and executable; (ii) *methods* that provide support during the *Design Phase* of a system engineering process for the formalization and evaluation of requirements to guarantee their fulfillment.
- a high risk of having to revise basic ***design choices*** with a consequent increasing in both ***completion time*** and ***development cost.***

Need of:

- *Models* for representing system requirements in a more formal way;
- *Methods* and *techniques* centered on *model-based approaches* able to support the *modeling*, *evaluation*, and *validation* of *requirements*;
- *Tools* and *Simulation Environments,* able to make easier System Safety analysis.

Modrio project (WP2) – ITEA 2

WP2 involved partners

WP2's Objectives:

- Formalization of system requirements;
- Definition of methods for Safety Analysis of physical systems.

Aim of the Proposal

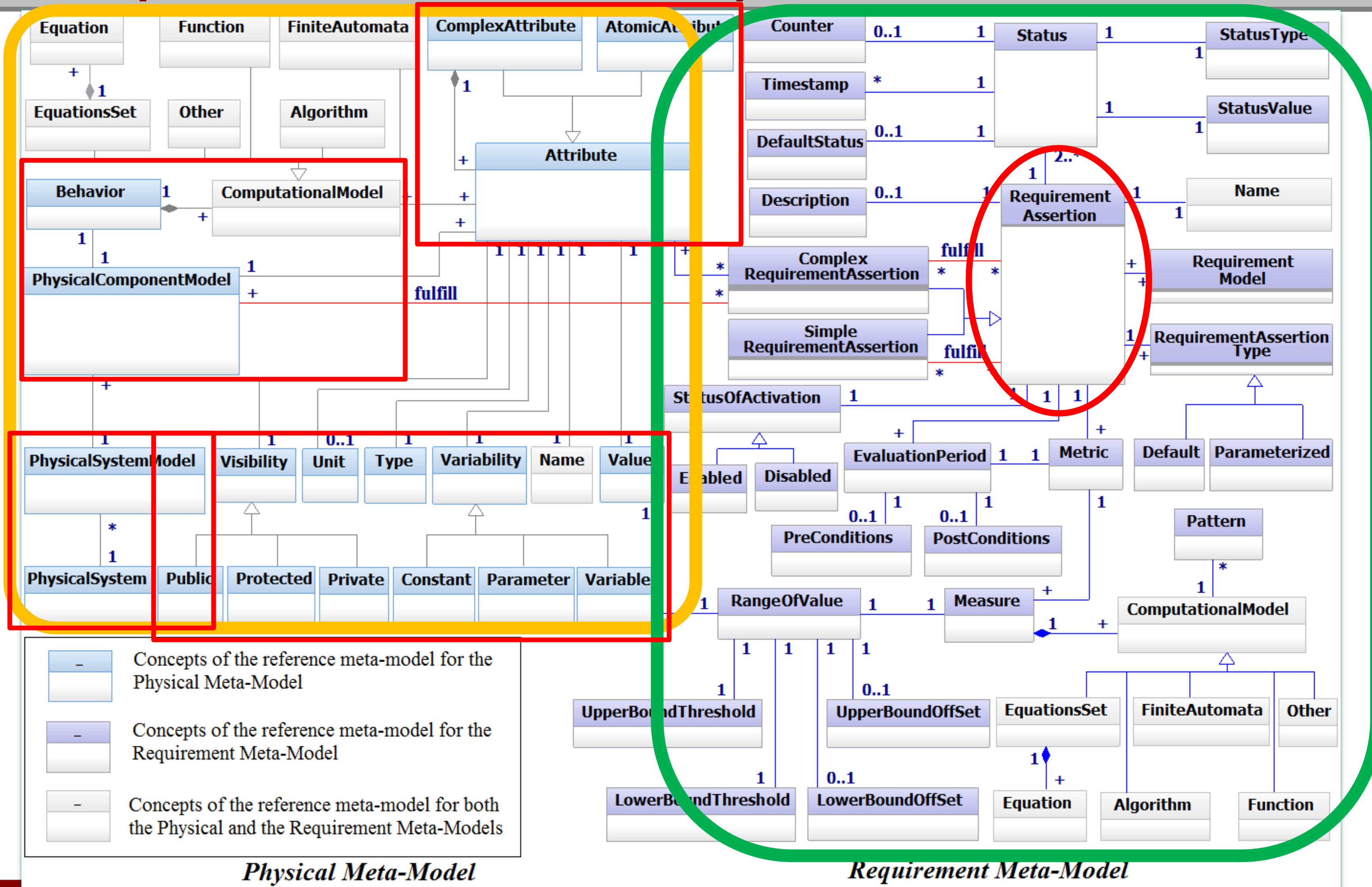
General Goal

- (i) to develop a comprehensive approach for the *definition and modeling of requirements* of a physical system in a more formal way;
- (ii) to define a mechanism to enable their traceability in order to *support the verification process through simulation*.

Proposal

- A *meta-model* to represent system requirements;
- *Approaches* to model them in an equation-based context;
- Some *extensions* of the Modelica language are introduced;
- Implementation in *OpenModelica* (Open Source).

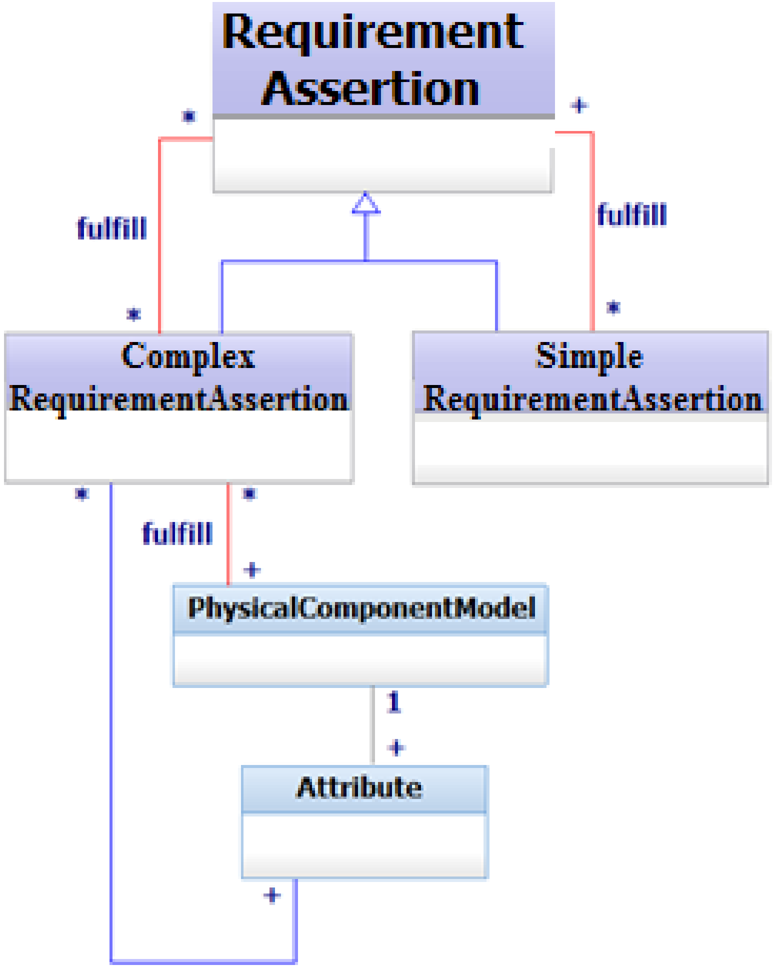
A meta-model for modeling System Requirements as RequirementAssertions



A meta-model for modeling System Requirements as RequirementAssertions

Main Concepts:

- ***requirement***: which is represented by a *RequirementAssertion* able to validate the behavior of a specific *PhysicalComponentModel* which is related to, or to validate interactions among different *PhysicalComponentModels*
- ***fulfill***: which expresses the entailment relationship between *PhysicalComponentModels* and a *RequirementAssertion*, as well as among *RequirementAssertions*. *Fulfill* provides the propagation process of an assessment among *RequirementAssertions*.



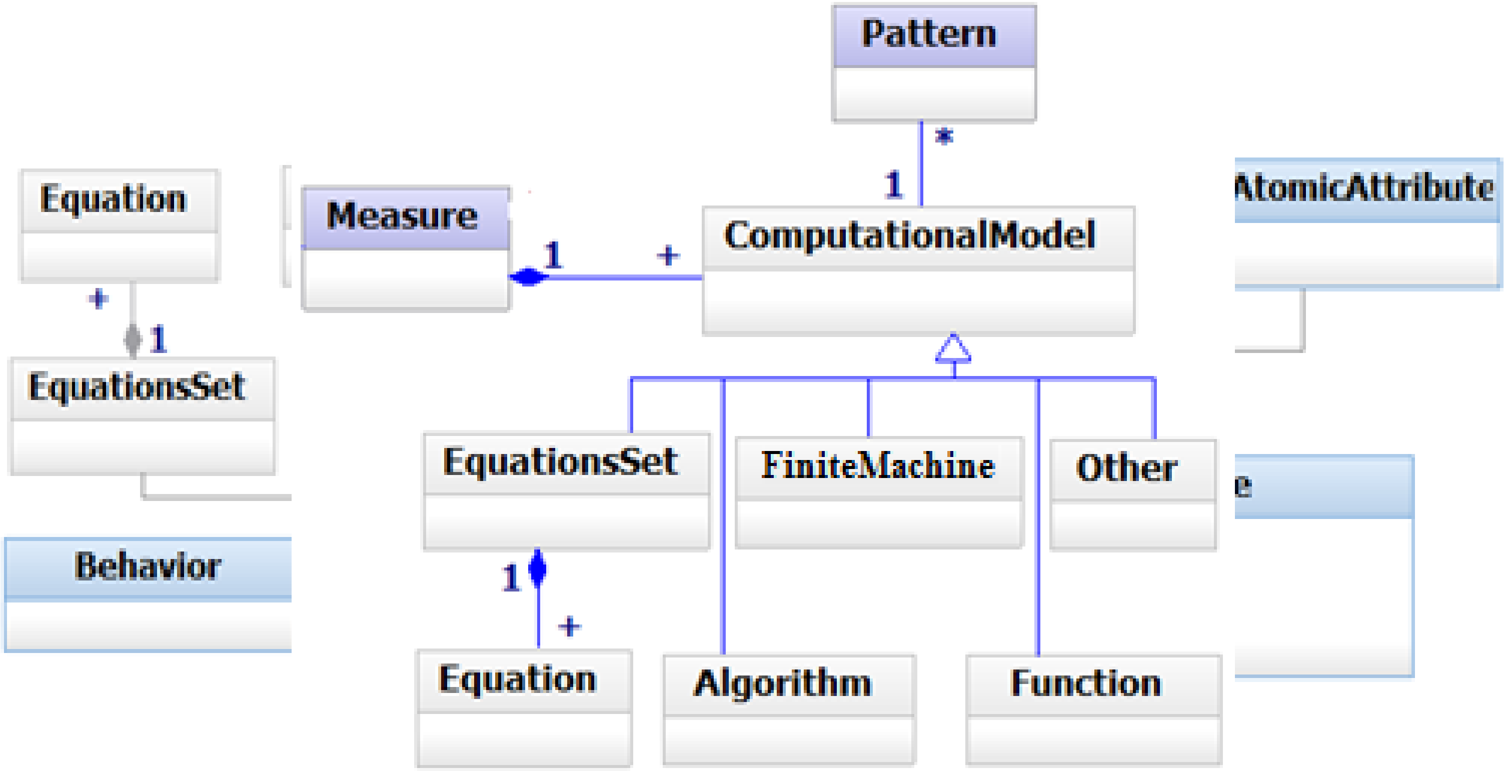
Ex. PhysicalComponentModel ***c1, c2, c3;***
RequirementAssertion **ra1, ra2, ra3;**

c1 **fulfill** ra1; → ra1 Complex
ra1 **fulfill** ra2; → ra2 Simple
c2, c3, ra2, **fulfill** ra3. → ra3 Complex

A meta-model for modeling System Requirements as RequirementAssertions

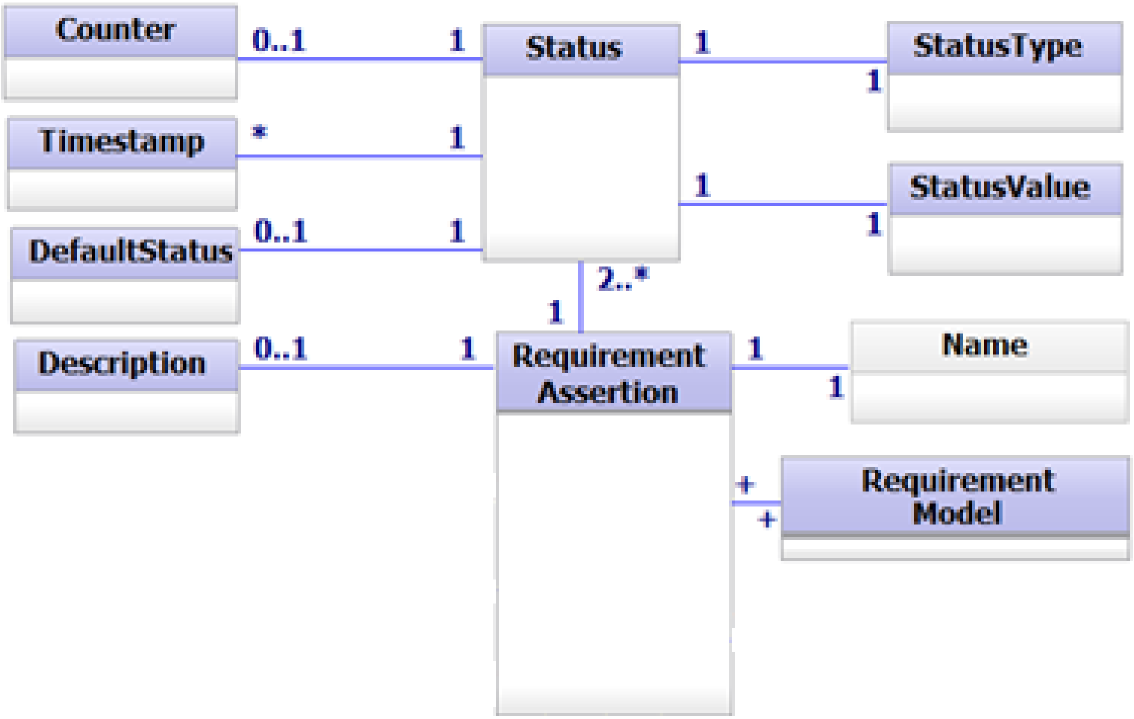
ComputationalModel:

- it defines the ***Behavior*** of a *PhysicalComponentModel*;
- It is used to express a ***Measure*** of a *RequirementAssertion*.



A meta-model for modeling System Requirements as RequirementAssertions

Status: in order to represent the status of fulfillment of the requirement, which in turn is defined in terms of a *StatusType* and a *StatusValue*.



Each ***Status*** could have:

- a *Counter* counting how many times the *RequirementAssertion* has gone in a specific state
- and a *Timestamp* in order to register each occurrence of the event.

Each *RequirementAssertion* is characterized by a *Name* and a possible *Description* in a text format by using the natural language;

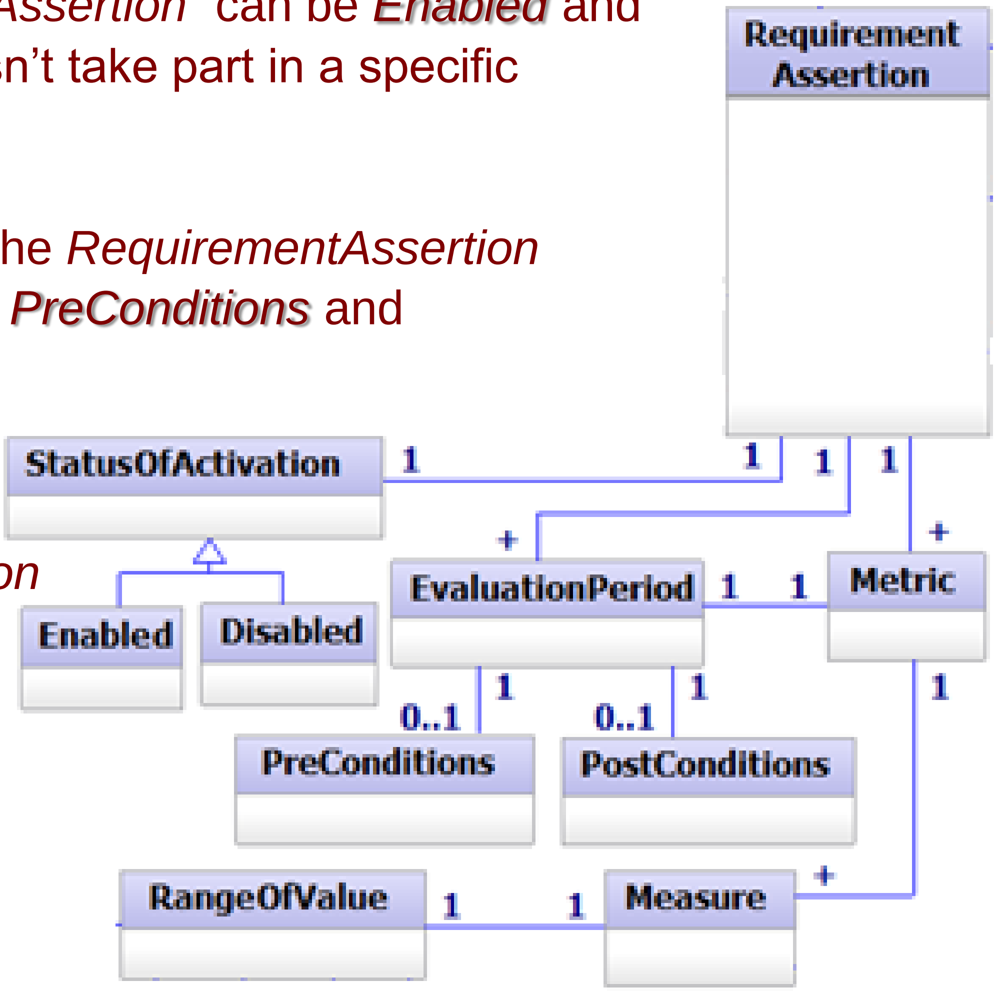
A meta-model for modeling System Requirements as RequirementAssertions

StatusOfActivation: a *RequirementAssertion* can be ***Enabled*** and ***Disabled*** in order to decide if it takes/doesn't take part in a specific scenario or simulation run;

EvaluationPeriod: to indicate when the *RequirementAssertion* has to be evaluated according to possible ***PreConditions*** and ***PostConditions***

Metric: to describe the objective to be verified for which the *RequirementAssertion* has been defined (e.g. MTTF);

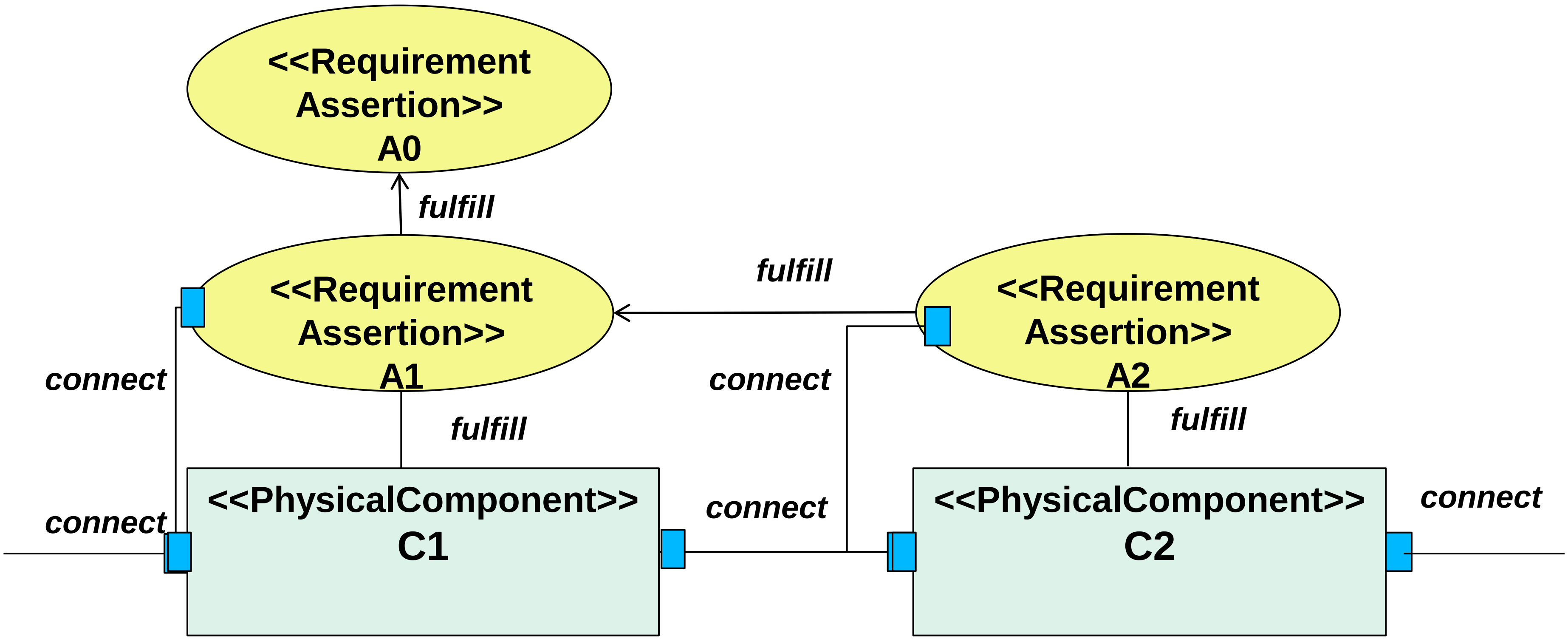
It has to define a way which *objectively* allows its evaluation in terms of ***Measure*** (e.g. the MTTF can be meseasured as number of failures in a period of time).



Approaches for Modeling System Requirements in Modelica

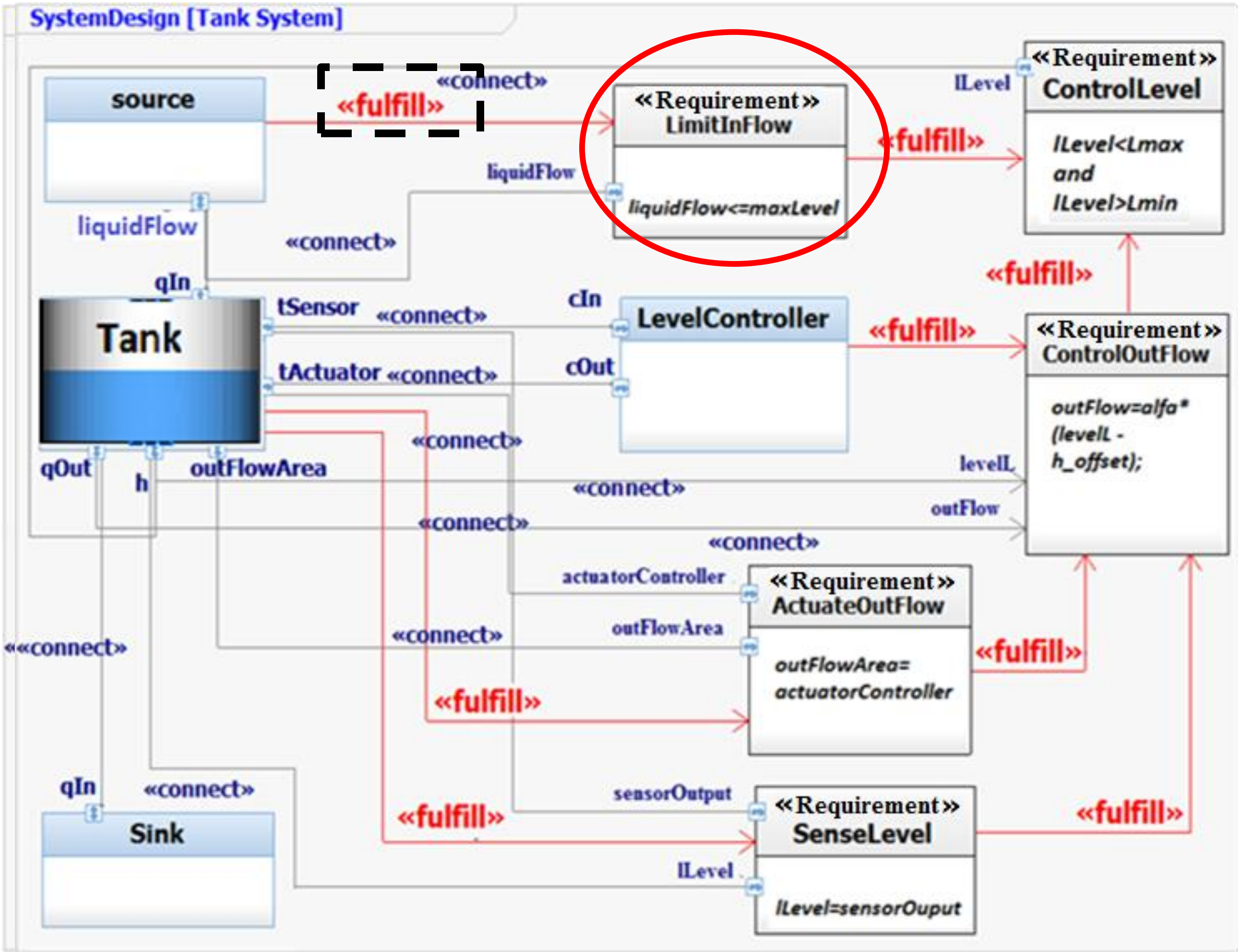
- They are based on the two main concepts of *RequirementAssertion* and *Fulfill* as stated into the proposed meta-model.

Approach A



Exploiting the Approach A: a Case Study

An example of scenario



How does the Source Code look like?

```
package PhysicalComponentModel
model Source;
  LiquidFlow qOut;
  parameter Real flowLevel=0.02;
equation
  qOut.lflow = if time>150 then
  3*flowLevel else flowLevel;
end Source;
```

- *RequirementAssertion_1: LimitInFlow*, which takes in input the value of the *qOut* port of the Source component. It is satisfied if the liquid flow produced by the Source component is less than a specific “maxLevel” (i.e. liquidFlow<=maxLevel, in our case maxLevel =10).

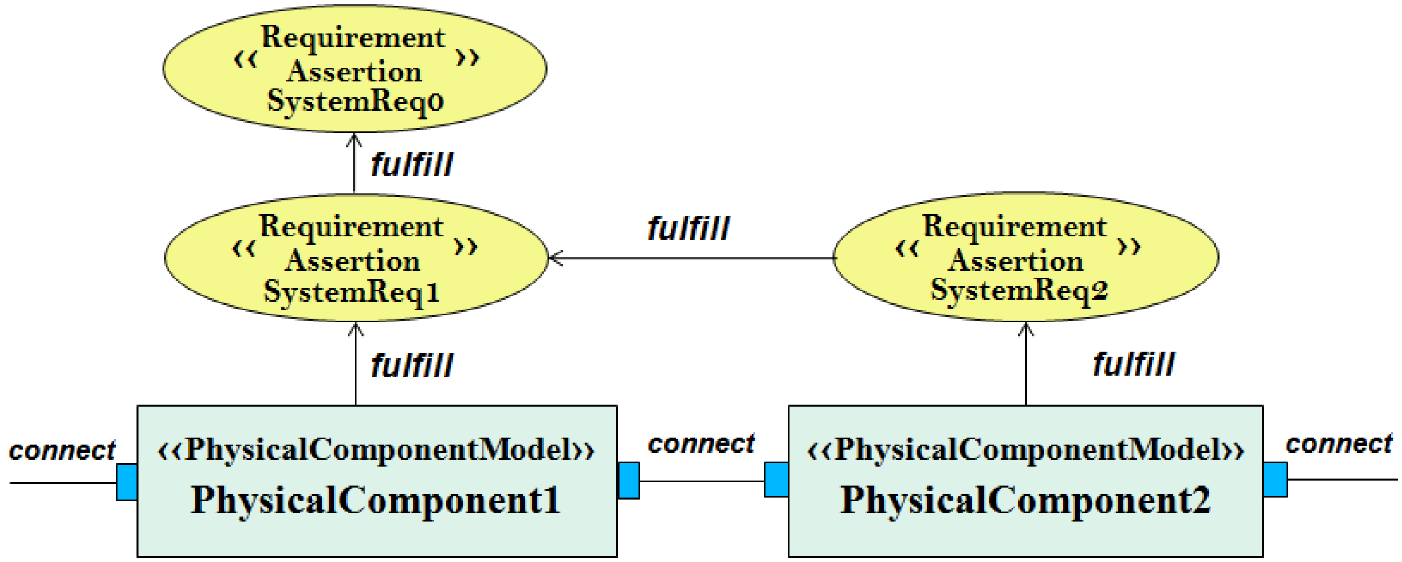
```
requirement Requirement1
  Real liquidFlow; "qOut of Source"
  parameter Real maxLevel=10;
equation |
  if liquidFlow<=maxLevel then
    Status.satisfied;
end Requirement1;
```

```
model ExtendedSystemDesign
  //PhysicalComponentModels
  Source source;
    …
  //RequirementComponents
  Requirement1 limitInFlow;
  …
equation
  //Connection among PhysicalComponents
  connect(source.qOut,tank1.qIn);
  …
  //fulfill connections
  (source)fulfill(limitInFlow);
  (limitInFlow,controlOutFlow)
  fulfill(controlLevel);
  (levelController,actuateOutFlow,
  senseLevel)fulfill(controlOutFlow);
  //connection between physical
  //components and requirements
  connect(source.qOut,limitInFlow.
  liquidFlow);
  connect(tank1.h,senseLevel.lLevel);
  …
end ExtendedSystemDesign;
```

Approaches for Modeling System Requirements in Modelica

Approach B: It is a variant of the previous approach to avoid the exploitation of the construct of "connect" between *RequirementAssertion component* and *Physical Component.*

Beside to keyword ***requirement***, the ***On***-keyword is introduced. "*On*" allows a *RequirementAssertion* to be defined on a specific model and by inheriting their attributes, on which it will carry out processing.



How does the Source Code look like?

```
package PhysicalComponentModel
model Source;
  LiquidFlow qOut;
  parameter Real flowLevel=0.02;
equation
  qOut.lflow = if time>150 then
  3*flowLevel else flowLevel;
end Source;
```

- *RequirementAssertion_1*: *LimitInFlow*, which takes in input the value of the *qOut* port of the Source component. It is satisfied if the liquid flow produced by the Source component is less than a specific "maxLevel" (i.e. liquidFlow<=maxLevel, in our case maxLevel =10).

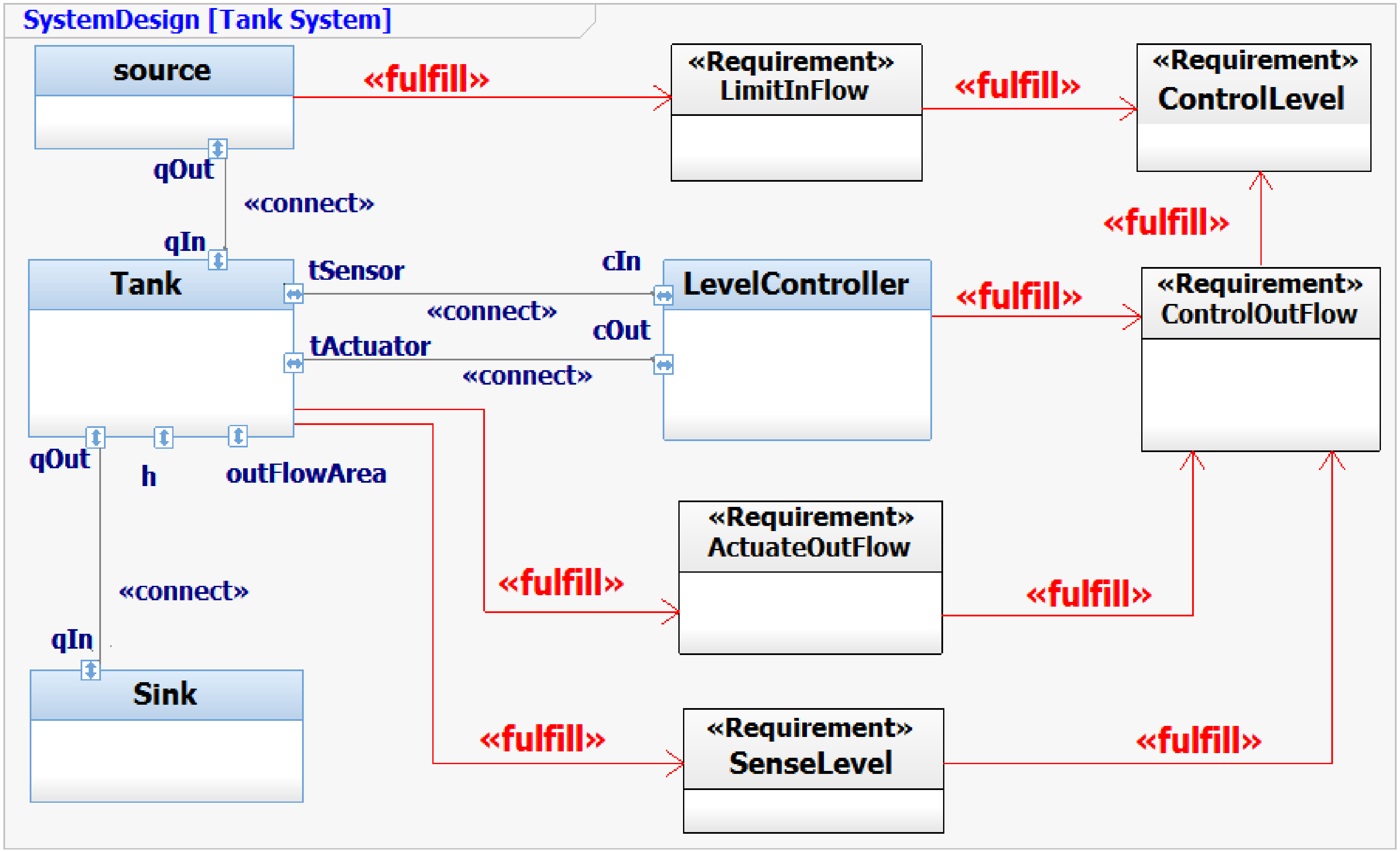
```
package RequirementModel
requirement Requirement1 On Source
  parameter Real maxLevel=10;
equation
  if Source.qOut<=maxLevel then
    Status.satisfied;
   ...
end Requirement1;
...
```

```
model ExtendedSystemDesign
  //PhysicalComponentModels
  Source source;
  ...
  //RequirementComponents
  Requirement1 limitInFlow;
  ...
equation
  //Connections among PhysicalComponents
  connect(source.qOut,tank1.qIn);
  ...
  //fulfill relationships
  (source)fulfill(limitInFlow);
  (levelController,actuateOutFlow,
  senseLevel)fulfill(controlOutFlow);
  (limitInFlow,controlOutFlow)
  fulfill(controlLevel);
end ExtendedSystemDesign;
```

NO

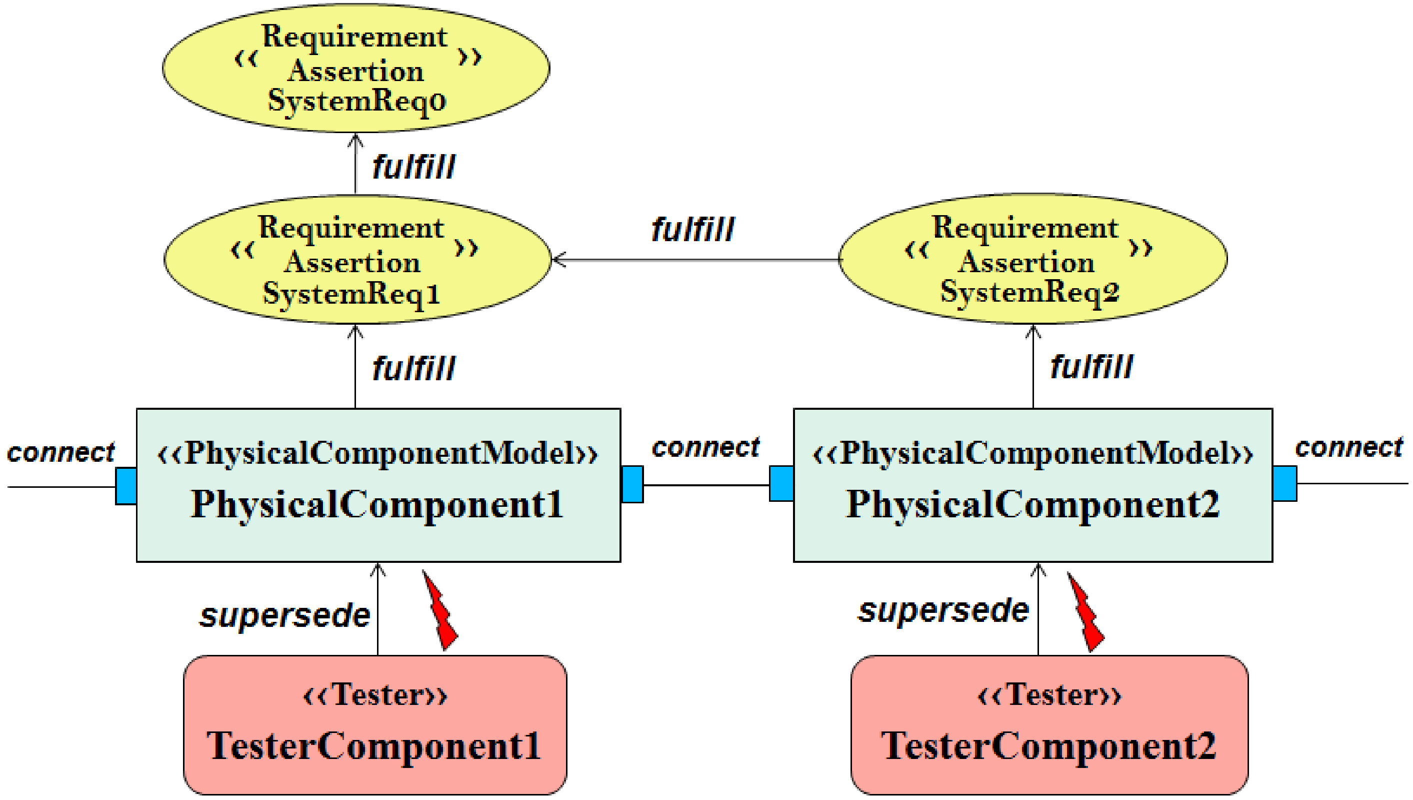
connect (PhysicalComponent, RequirementComponent)

Exploiting the Approach B: a Case Study

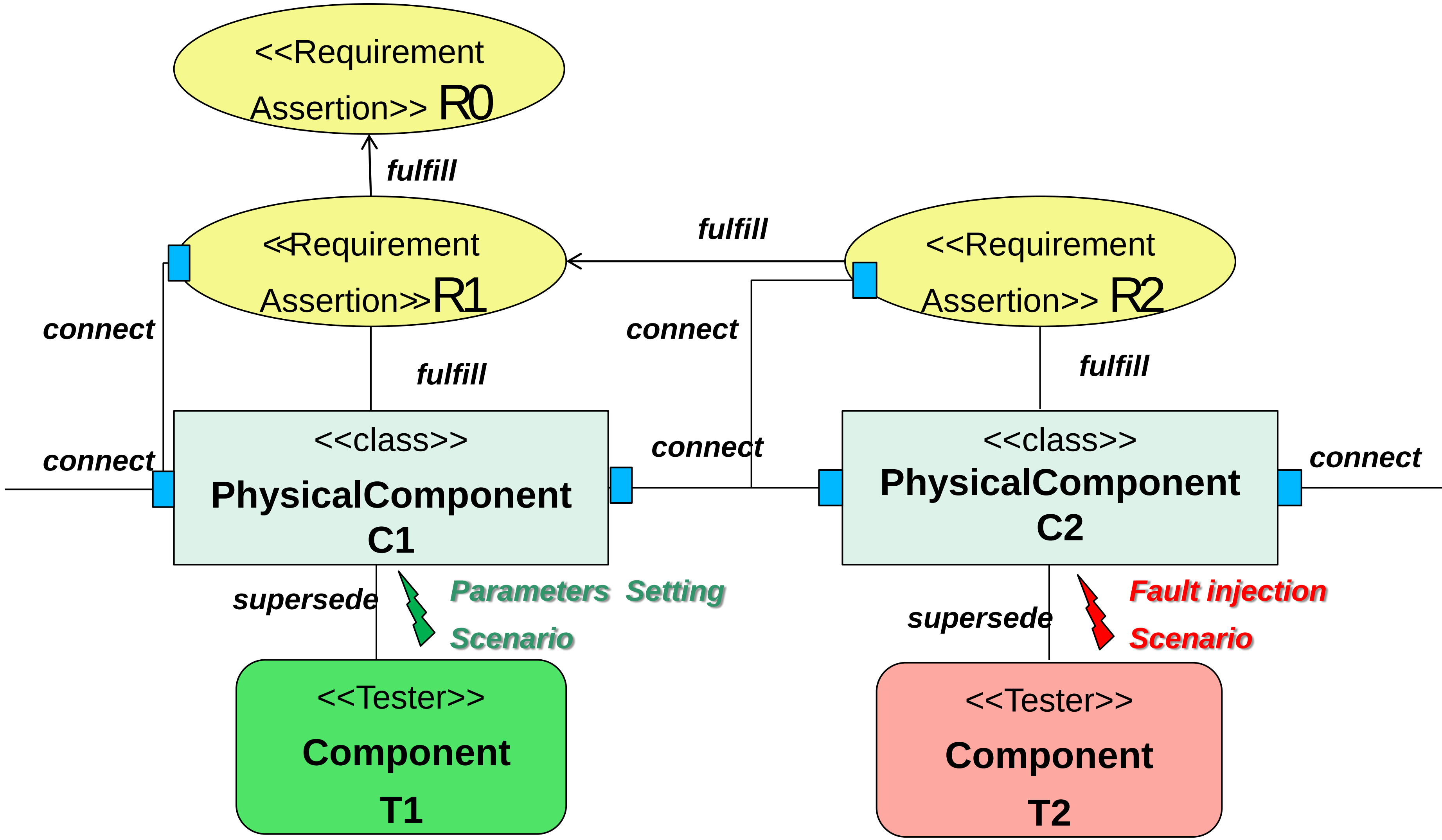


Modeling Dysfunctional Behavior and Scenarios

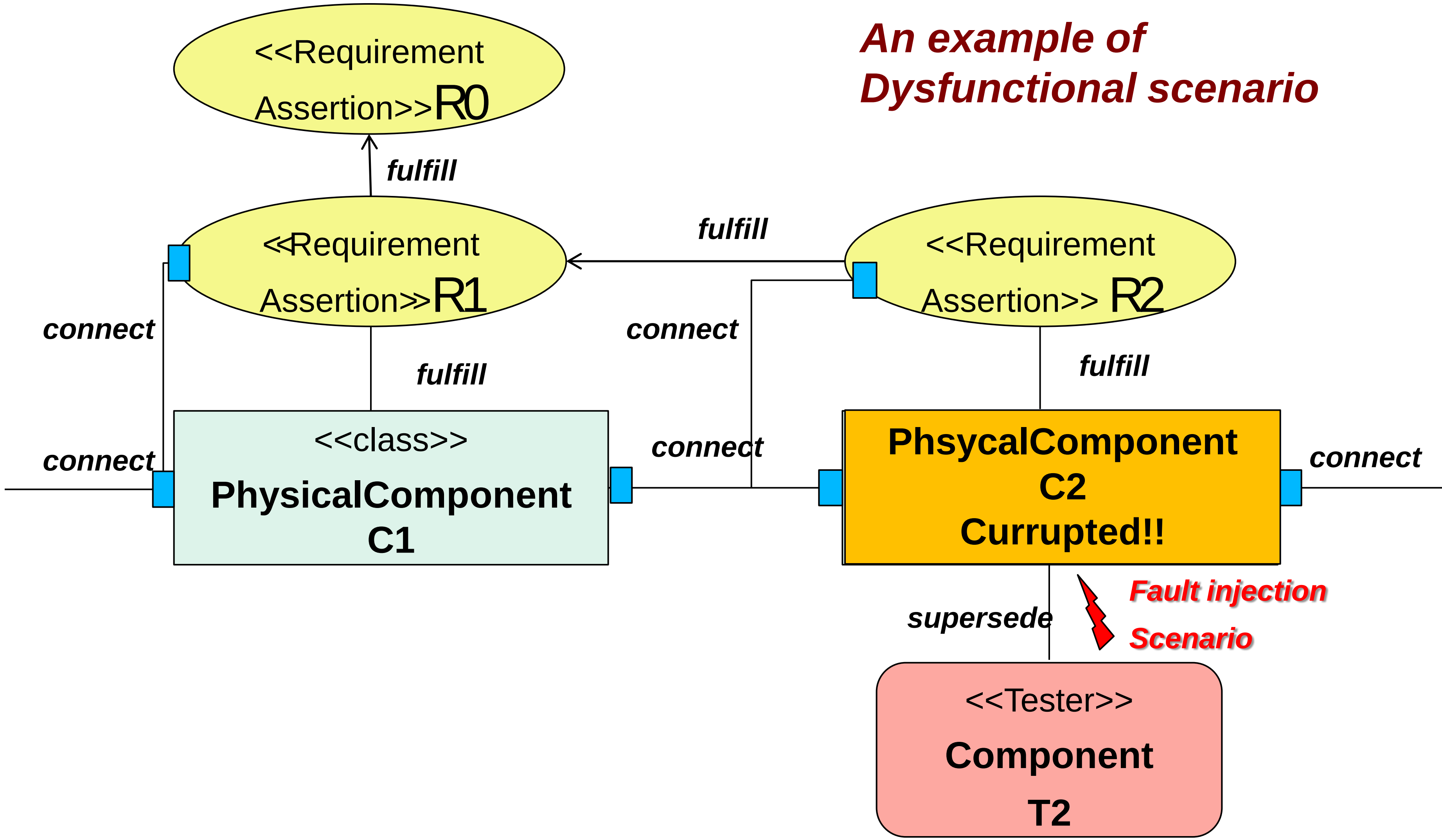
Approach C: This approach takes into account the possibility of considering the feature of altering the values, as well as provide parameters setting of particular scenarios, of the components by extending the previous approaches.



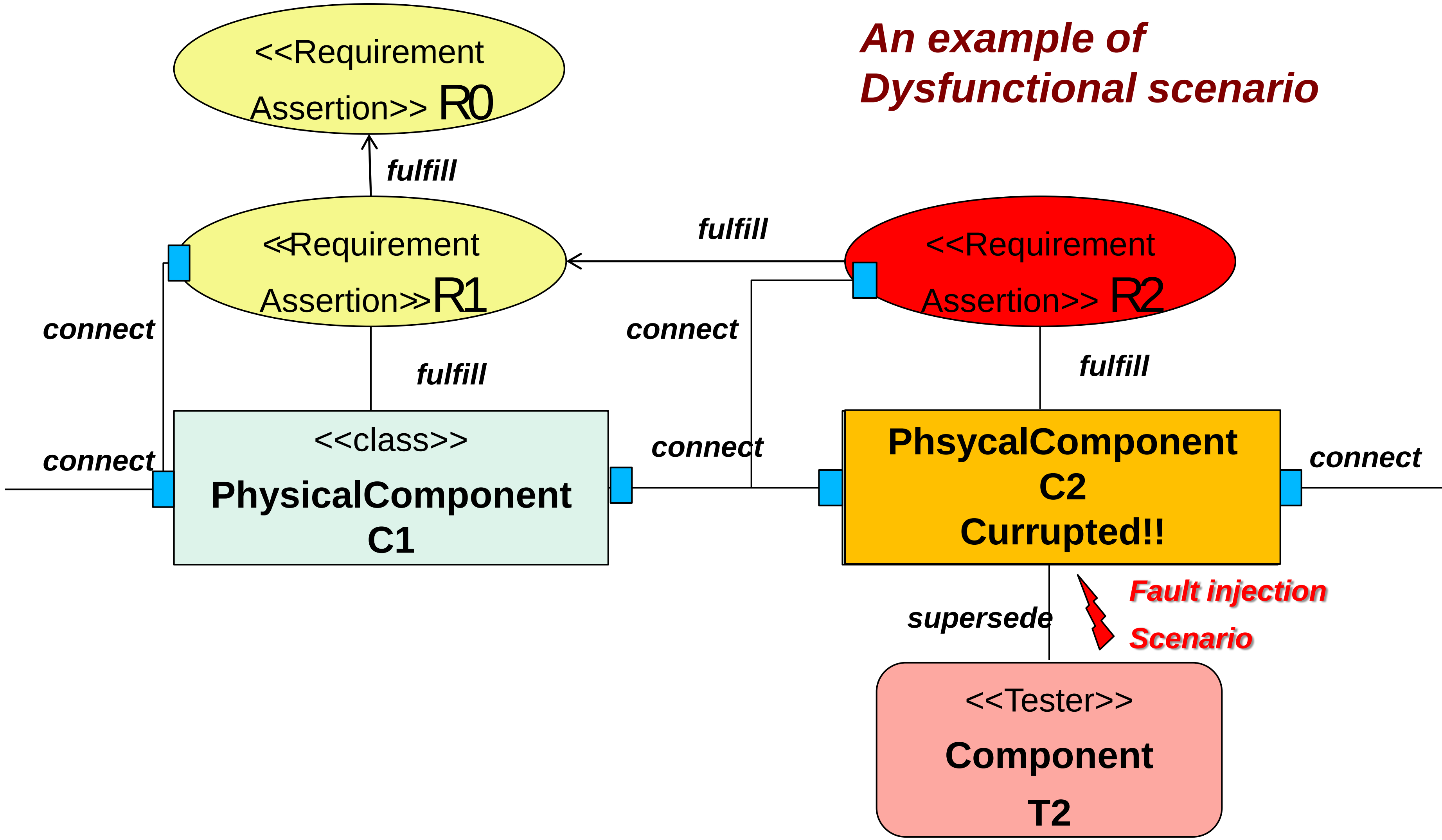
Modeling Dysfunctional Behavioral and Scenarios



Dysfunctional Behavioral Modeling

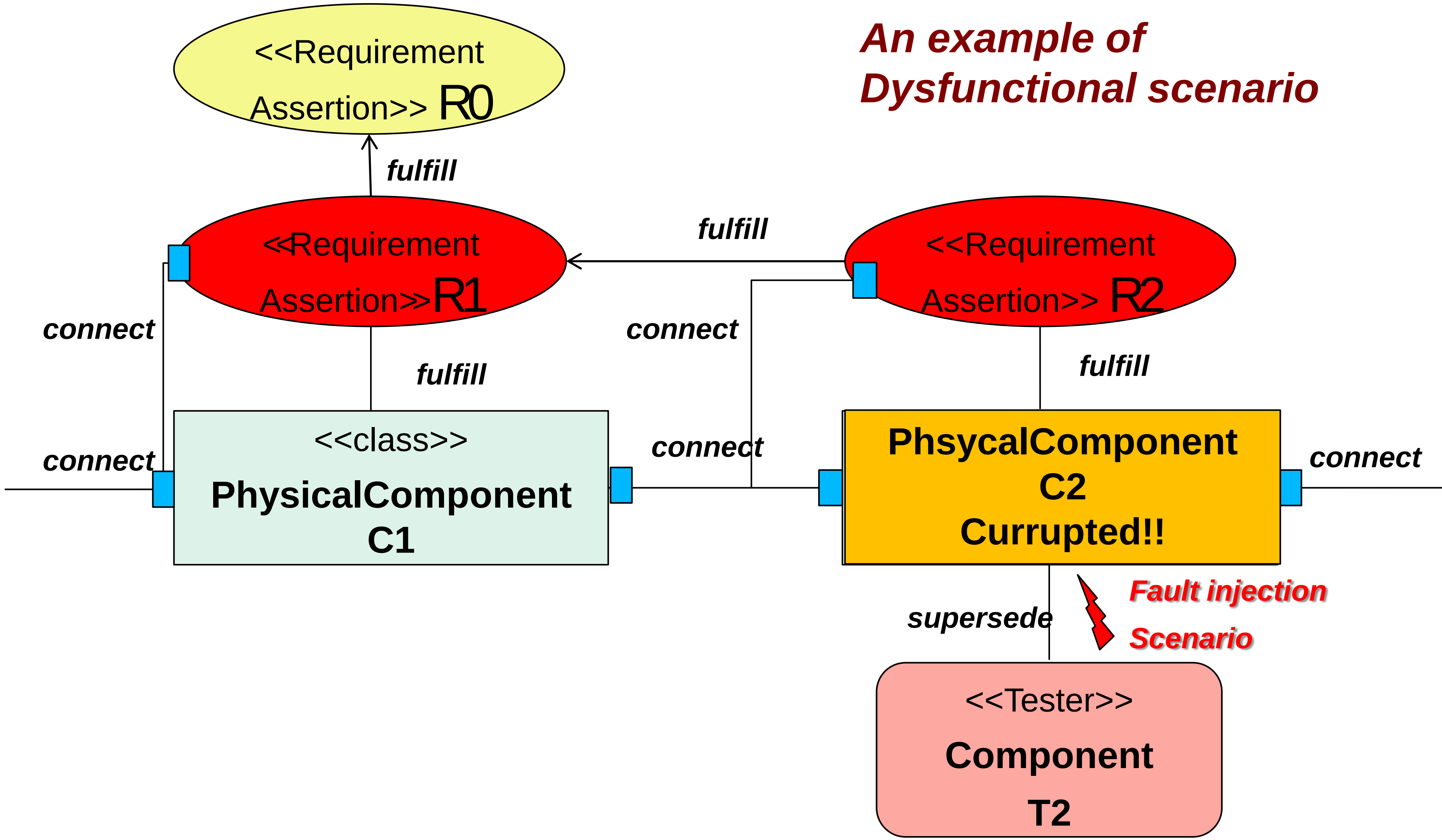


Dysfunctional Behavioral Modeling



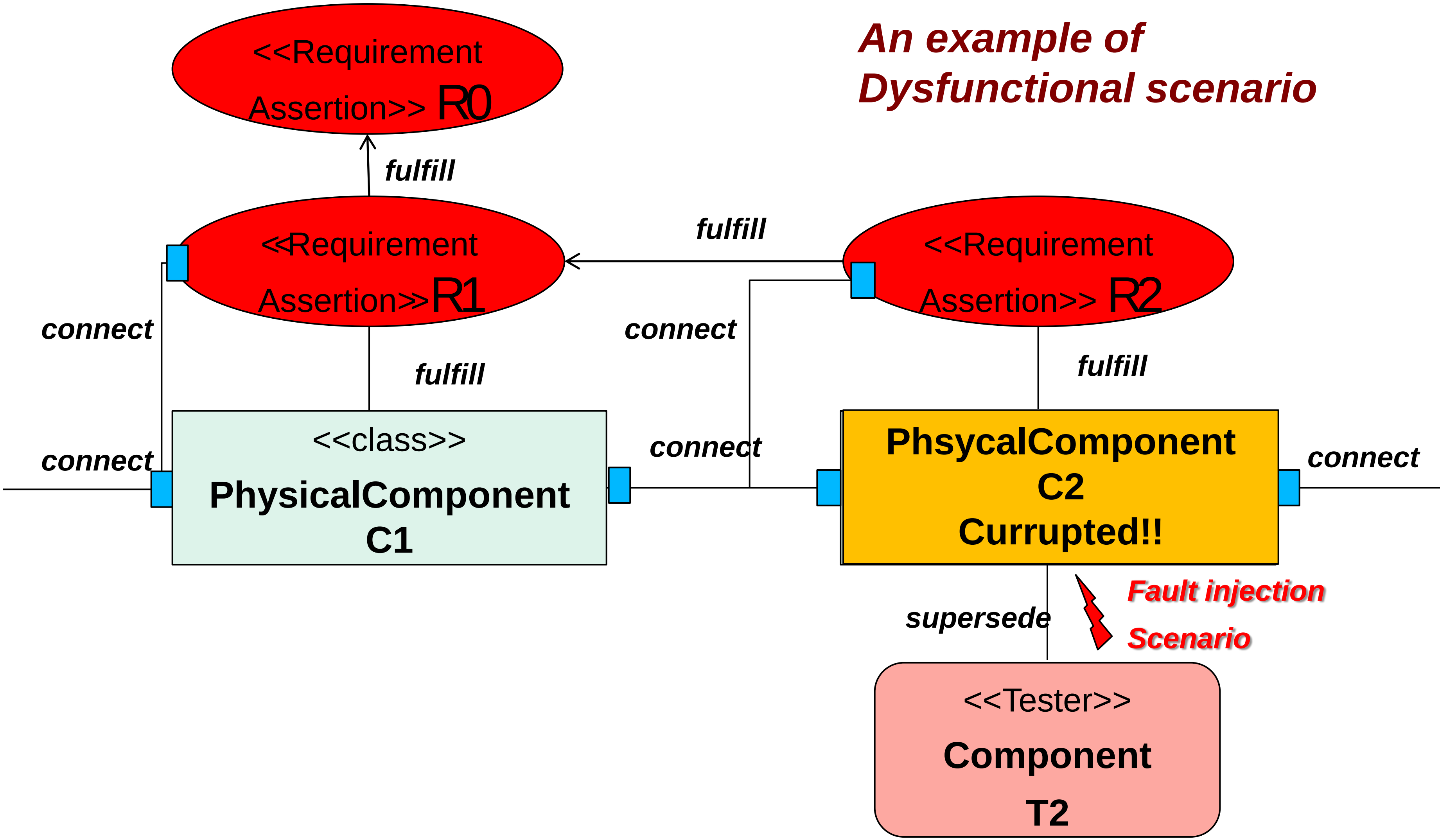
Dysfunctional Behavioral Modeling

An example of Dysfunctional scenario



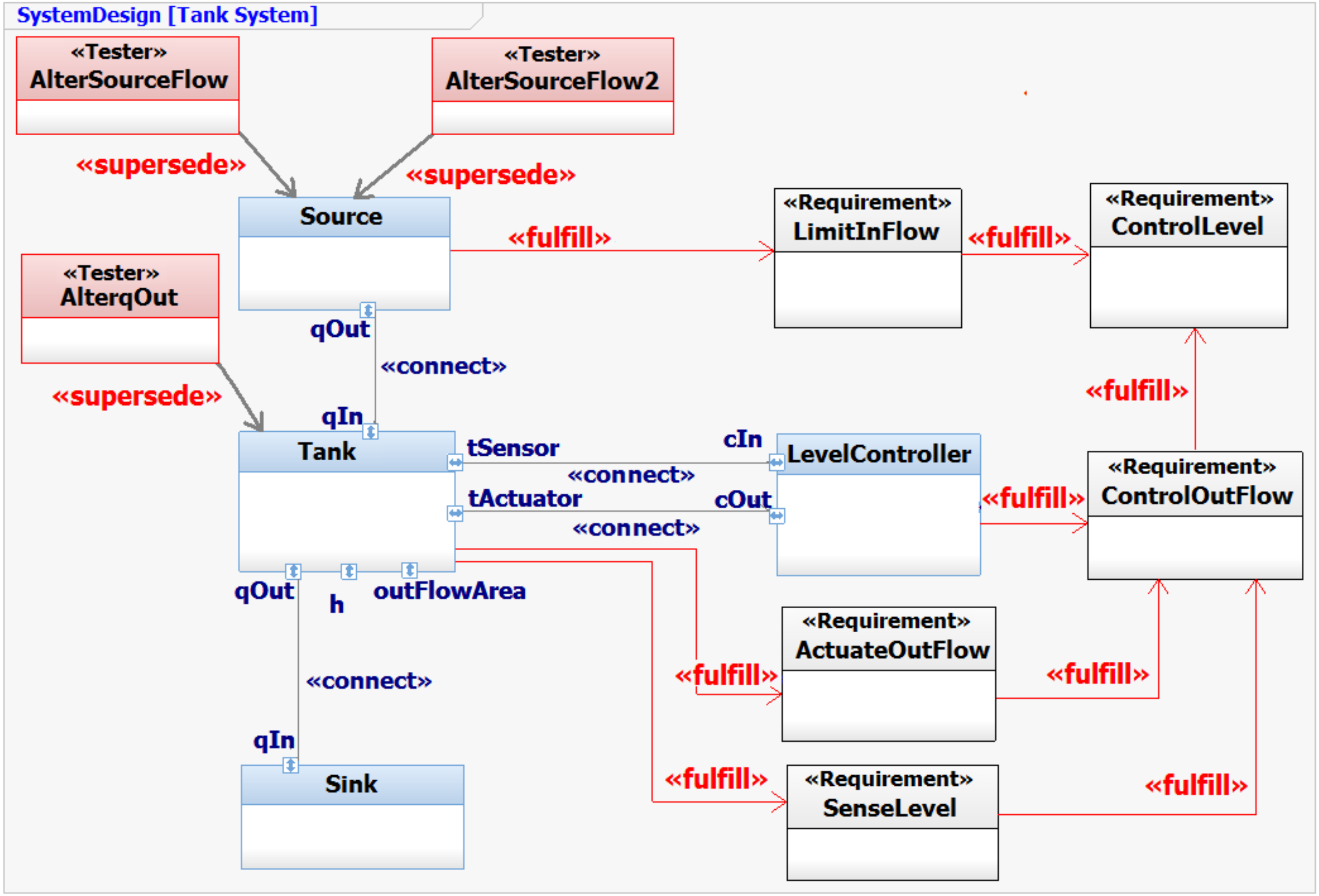
Dysfunctional Behavioral Modeling

An example of Dysfunctional scenario



Exploiting the Approach C: a Case Study

- **3 Tester components**



How does the Source Code look like?

```
package PhysicalComponentModel
model Source;
  LiquidFlow qOut;
  parameter Real flowLevel=0.02;
equation
  qOut.lflow = if time>150 then
  3*flowLevel else flowLevel;
end Source;

tester AlterSourceFlow On Source
  parameter Real flowLevel=0.04;
  …
equation
  qOut.lflow=flowLevel;
end AlterSourceFlow;

tester AlterOut On Tank
  …
equation
  actuatorControllerV=-
  flowGain*tActuator.act;
  qOut.lflow = actuatorControllerV;
  tSensor.val = h;
  outFlowArea=-qOut.lflow/flowGain;
end AlterOut;
```

```
model ExtendedSystemDesign
  //PhysicalComponentModels
  Tank tank1(area=1);
  Source source;
  …
  //RequirementComponents
  …
  //TesterComponents
  AlterSourceFlow alterSourceFlow;
  AlterSourceFlow2 alterSourceFlow2;
  AlterOut alterOut;
equation
  //supersede relationships
  (alterSourceFlow,
  alterSourceFlow2)supersede(source);
  (alterOut)supersede(tank1);

  //fulfill relationships
  …
end ExtendedSystemDesign;
```

Conclusions and Future perspectives

Contribution:

- ✓ A reference ***Meta-model*** for representing System Requirements in terms of ***RequirementAssertions*** has been defined.
- ✓ Possible extensions of the **Modelica** language: new concepts and keywords, such as ***requirement*** and ***fulfill*** for supporting the verification of models as well as ***supersede*** and ***tester*** for parameters setting of scenarios, have been introduced.
- ✓ Three ***approaches*** for the modeling of System Requirements that adhere to the proposed meta-model, have been outlined.

Ongoing and future works:

- ➢ Improvement of ***Modelica extensions*** and their implementation in ***OpenModelica*** for *the verification* and *validation of models*;
- ➢ Definition and implementation of ***OpenModelica API*** for enabling ***Fault Tree Analysis***;
- ➢ Definition of a ***Methodology*** for supporting the Modeling and the Validation process of physical systems.

Thank you!

atundis@dimes.unical.it