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

Andrea Tundis[1] Lena Rogovchenko-Buffoni[2] Peter Fritzson[2] Alfredo Garro[1]

[1]Department of Computer Engineering, Modeling, Electronics, and System Sciences (DIMES), University of Calabria, Italy, {garro,atundis}@dimes.unical.it

[2]Department of Computer and Information Science (IDA), Linköping University, Sweden, {peter.fritzson,olena.rogovchenko}@liu.se

Abstract

The modeling of system requirements deals with formally expressing constraints and requirements that have an impact on the behavior of the system to enable their verification through real or simulated experiments. The need for models representing system requirements as well as for methods and techniques centered on model-based approaches able to support the modeling, evaluation, and validation of requirements and constraints along with their traceability is today greater than ever. In this context, this paper proposes a *meta-model* for modeling the requirements of physical systems. Furthermore, different approaches for integrating the modeling of system requirements in the Modelica language and their verification during the simulation are proposed and, then, evaluated and compared through a case study.

Keywords Requirements, Properties, Modeling, Assertions, Modelica, Safety, Verification, Validation

1. Introduction

In the systems engineering context, although several research activities are focused on the system design phases, there is still a lack of practices and approaches that specifically deal with the analysis, modeling, and verification of requirements in an integrated framework. One of the main open issues concerns the support provided during the design for the verification and validation (V&V) of the system under consideration. Indeed, it is crucial not only to represent in detail both the structural and behavioral design of a system, but also to ensure the proper operation of the overall system and of each individual component to guarantee their functional correctness in compliance with the requirements. Moreover in several industrial domains such as nuclear plants, medical appliances, avionics, and automotive industry, some requirements such as safety requirements must be compliant to standard specifications (see IEC 61508) and norms to allow the commercial release of a system.

In order to add support for verification and validation during the design stage of the systems engineering process we formalize a set of concepts that will allow us to model *system requirements*, in [9] called properties. In the following we will use the term *requirement*, which is defined [9] as an expression that specifies a condition that must hold true at given times and places. As a rule, their identification and definition is neither a trivial nor a unique process, and can significantly depend on the reference domain and application context. Similarly, their formalization and modeling can vary with respect to the objectives to be reached.

In general, the first step of a systems engineering process is concerned with the analysis of informal *User Requirements* (*URs*). These are typically *problem-oriented* statements and they focus on the required capabilities. Thus, they need to be converted into *solution-oriented* statements. The *System Requirements* (*SRs*) are then derived by decomposing the *URs* into sets of basic required functionalities. *SRs* form the basis for the subsequent system functional analysis and design synthesis phases [8]. In particular, in the System Design phases, *SRs* are used to define both the structure and the behavior of the System under development. Specifically, in an equation-based context, the behavior of each system component, as well as the behavior of the entire system, is represented by a set of equations defined using component attributes (such as variables, parameters and constants).

Starting from the *SRs* and according to the defined System Design (*SD)*, additional *mechanisms* called *Requirement Assertions* can be defined in order to verify as well as to trace through the simulation the fulfillment of the *SRs*. Indeed a *requirement assertion* can be associated with a real system, subsystem, equipment or component, or with a model of the real system, subsystem or component and it defines what the system should guarantee, or the validity domain for the behavior of the system. In particular, in our context a *requirement* is represented by an *assertion* that is related to a specific *physical component* and which exploits the attributes of the component in order to verify and trace the fulfillment of some *SRs* related to a specific *component*. It worth to notice that while *user* and *system requirements* (both

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

EOOLT 2013 website: http://www.eoolt.org/2013/

functional and non-functional) are used in the analysis and design phases for the development of the system under consideration; formalized requirements as *requirement assertions* are exploited during the verification phases for evaluating if the system requirements are satisfied by a specific system design model. Consequently, an appropriate approach to define formalized requirements along with the possibility to retrieve information about their status is crucial for the overall development process.

Few works are currently available addressing the modeling of requirements which was one of the goals addressed in ModelicaML [11] during the OPENPROD project [4]. Specifically, our proposal is strongly related to: (i) [9] in which the representation of the requirements is closely bound and restricted to the exploitation/implementation of a software library; (ii) [14, 15] where the communication processes and evaluation mechanisms among requirements, in order to enable the propagation of assessments among them, are not properly dealt. Furthermore, well-known simulation environments exploit assertions to verify system requirements; for example, MathWorks Matlab/Simulink provides assertions and bound checking blocks as configurable components. However, they are able to face only a limited set of specific aspects (e.g. zero/nonzero signal, threshold values). In this context, our aim is twofold: (i) to develop a comprehensive approach for the definition and modeling of requirements of a physical system in a clear and well-defined way, (ii) to define a mechanism to enable their traceability in order to support the verification process through simulation. To address these issues, a *meta-model* to represent system requirements along with some different solutions to model them are described in an equation-based context. On the basis of this meta-model, several *extensions* of the Modelica language [3] (an object-oriented modeling language to describe physical systems by differential, algebraic and discrete equations), to model requirements in a more flexible way, are introduced.

In Section 2 the proposed meta-model is described, in Section 3 both its use and its possible integration in the Modelica context are illustrated along with some notation extensions, in Section 4 a case study for the evaluation of the various approaches is presented and discussed whereas in Section 5 conclusions are drawn and future works outlined.

2. A Meta-model for representing System Requirements as RequirementAssertions

The concepts required for modeling system requirements are clearly identifiable and their representation can be generalized. For this purpose we define formal meta models [1].

Even though the notions of model and meta-model are crucial when we talk about representation and modeling, often these terms generate confusion, so it is necessary to clarify the difference between them and the context for the use of each of them.

Firstly we can define the concept of *subject* as the main thing that we want to think/reason about and on which we perform experiments. This usually belongs to the real world. To solve a problem we construct a simplified representation of the subject, called *model*, to which different experiments can be applied, in order to answer questions aimed at the subject. Since a model captures only a part of the complete subject, it is possible to define many models which represent the same subject but that are able to capture different characteristics, aspects, variables and parameters. In order to perform reasoning on a model it is necessary to know exactly which variables are available, furthermore, it is necessary to know the structure of the model. Such information can be expressed through *meta-data* by defining a higher abstraction level called *meta-model*. Hence, a meta-model is a model that defines the structure of valid models (see Figure 1).

In the following the definition and description of the proposed *meta-model* (see Figure 2) is provided. It is a combination of two main parts: the *Physical Meta-Model* (in the left-side) and the *Requirement Meta-Model* (on the right-side).

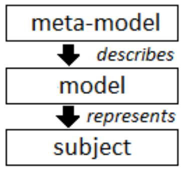

Figure 1. *Meta-model*, *model* and *subject* abstraction levels.

As previously stated, before defining *System Requirements*, it is necessary to build a representation of the physical model. Thus, the *meta-data* of the *Physical Meta-Model* are used to describe and to represent one among all the possible physical models of a specific actual system, whereas the *meta-data* of the *Requirement Meta-Model* are exploited to represent *System Requirements* in terms of *requirement assertions* by defining a possible *requirement-model* on a specific physical model representation.

Starting from the *Physical Meta-Model* side, the main concept is the *Attribute,* which represents a characteristic (i.e. temperature, pressure, level of liquid, age) of an entity (i.e. a system, a sub-system, a component); in the proposed meta-model, it is defined by (i) a *Name* (by which it is referred) (ii) a *Type* (type of value which is expected), (iii) a *Value* (a possible value among all the range of values related to a specific *Type*) and (iv) (optionally) a *Unit* of measure. Each *Attribute* is associated with one specific *Variability* which in turn can be (i) *Constant* which means that its *Value* never changes, (ii) *Variable* which means that its *Value* depends on other attributes, (iii) *Parameter* which means that its *Value* can be properly tuned. Moreover, each *Attribute* has to specify its access level called *Visibility* which, according to the meta-model, could be either *Private,* if accessible only internally to the component in which it has been defined, *Protected*, if accessible by the descendants, or *Public*, if accessible externally.

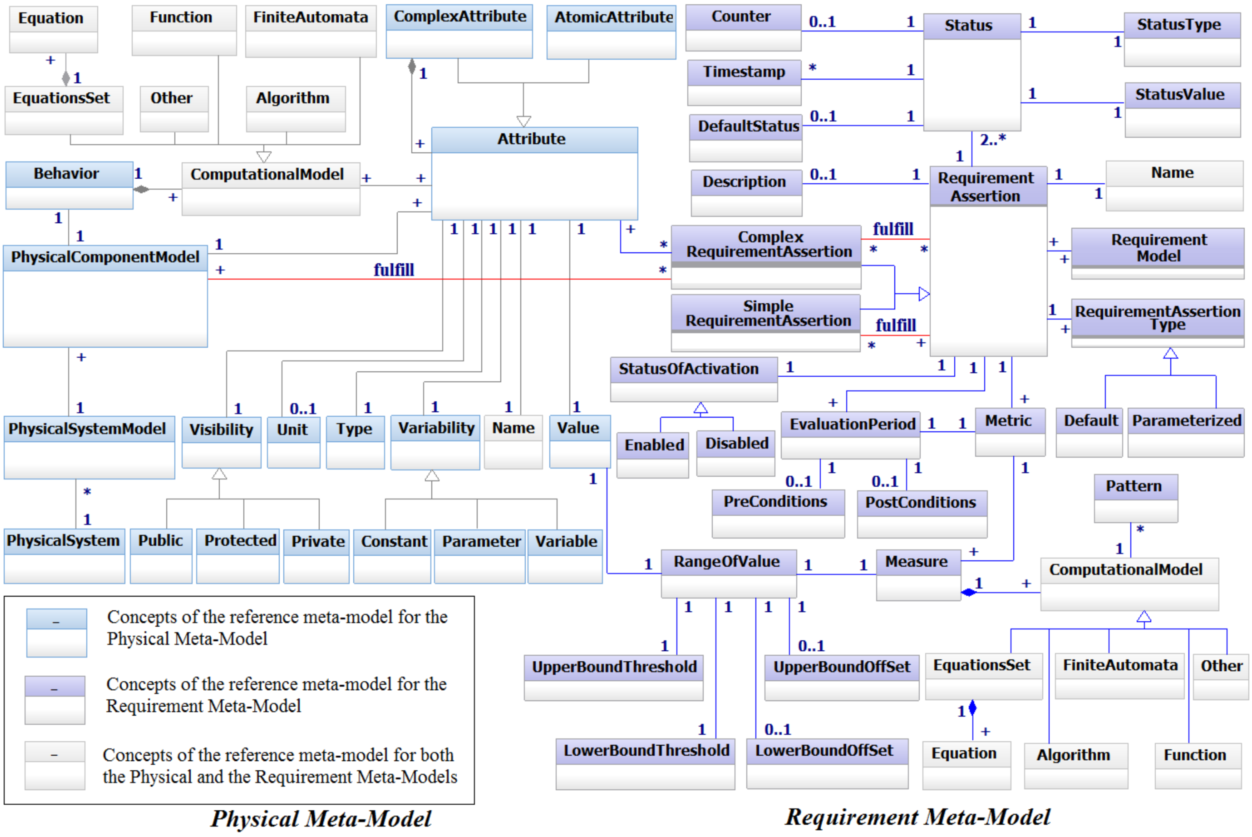

Figure 2. A meta-model for modeling *System Requirements*.

An *Attribute* can be (i) an *AtomicAttribute*, which means it cannot be further decomposed, (ii) a *ComplexAttribute,* that is, composed by other attributes.

A *ComputationalModel*, which could be represented through an a *Algorithm*, a *FiniteAutomata* (e.g. Timed Automata, Hybrid Automata, etc.), a *Function*, an *EquationsSet* (i.e. a set of *Equation* concepts), or by their combination as well as by *Other* kinds of computational models, defines the behavior of a *PhysicalComponentModel*. An *Attribute* has to belong at least to one *ComputationalModel* as well as a *ComputationalModel* has to use at least one *Attribute*. One or more *PhysicalComponentModels* compose a *PhysicalSystemModel,* which in turn is one of the many possible models to describe an actual system called *PhysicalSystem*.

While the *meta-data* on the left side of the figure is used for the description of the physical model, moving to the right side of the *meta-model*, we can see the concepts used for the modeling of *System Requirements*. Among these, the main concept is the *RequirementAssertion,* which is used to describe a *Requirement* of a system. A *RequirementAssertion* can be (i) a *SimpleRequirementAssertion*, that means it doesn't receive any input from any *PhysicalComponentModel*, (ii) a *ComplexRequirementAssertion,* which is connected directly to at least one *Attribute* and to one *PhysicalComponentModel*; this means that a *ComplexRequirementAssertion* is based on at least a *PhysicalComponentModel* and it is able to receive one or more input values coming from several attributes of the physical model; moreover, a *RequirementAssertion* (*SimpleRequirementAssertion* or *ComplexRequirementAssertion)* could be defined in terms of other *RequirementAssertions* whereas on a single *PhysicalComponentModel*, different *RequirementAssertions* can be defined.

According to the meta-model a *RequirementAssertion* belongs at least to one possible *RequirementModel* as well as a *RequirementModel* has to define at least one *RequirementAssertion*; each *RequirementAssertion* being characterized by:

- a *Name* and a possible *Description* in a text format by using the natural language;

a *RequirementAssertionType* which specifies the type of the role played by the *RequirementAssertion*; in particular a *RequirementAssertion* can have (i) a *Default* behavior type: it is allowed only to monitor a *PhysicalComponentModel* without influencing its evolution; (ii) a *Parameterized* behavior type: it is able to alter the value of a *PhysicalComponentModel* and influence its evolution (the *RquirementAssertion* has both *read* and *write* capabilities);

- at least two *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.* The first concept defines the type of value that a state can take (i.e. a Boolean type, a real type, etc.) whereas the second one represents the value which is related to a specific *StatusType* (such as True/False for a Boolean or NotEvaluated/Satisfied/NotSatisfied for a three valued logic, etc.). Each *Status* could be associated to both 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. Moreover, a status can be defined as a *DefaultStatus* (useful, for example, in the initialization phase when none value is still provided to the *RequirementAssertion*). A *RequirementAssertion* has a *StatusOfActivation*, that means it can be *Enabled* and *Disabled* in order to decide if it takes/doesn't take part in a specific scenario or simulation run.

- at least one *EvaluationPeriod* to indicate when the *RequirementAssertion* has to be evaluated according to possible *PreConditions* and *PostConditions* that could be based on temporal values or on values coming from *Attributes*. Moreover for each *EvaluationPeriod* a *Metric* must be associated.
- at least a *Metric* to describe the objective to be verified for which the *RequirementAssertion* has been defined (e.g. Mean Time To Failure for the Reliability); each metric has to define a way which objectively allows its evaluation in terms of *Measure* (e.g. the number of failures in a period of time to measure the Mean Time To Failure). Specifically, a *Measure* can be expressed by adopting an appropriate *ComputationalModel*; moreover, one or more *Patterns* could be applied for representing such *ComputationalModels* when a sort of recurrent structure occurs (e.g. a threshold pattern, a derivative pattern, a delay pattern, etc.). Furthermore, each measure should define a *RangeOfValue*, within the *Value* of the *Attribute* which is related to, in which it is valid. Such *RangeOfValue* is specified by: (i) a *LowerBoundThreshold*: minimum value of validity in the range; (ii) *UpperBoundThreshold*: maximum value of validity in the range; moreover, further thresholds as *LowerBoundOffSet* and *UpperBoundOffSet* can be exploited when the *Value* of a *RequirementAssertion* is respectively below/above the *LowerBoundThreshold* and *UpperBoundThreshold* for a limited time.

RequirementAssertions can describe the state and the *intended behavior* [6, 7] of *PhysicalComponentModels,* i.e. the expected behavior for which components are designed. Both *Physical Meta-Model* and *Requirement Meta-Model* are jointly exploited to describe the overall model (hereafter called *Extended System Design – ESD*) of an actual system.

To further clarify the meta-model above described, a simple exemplification is provided below, where some of the above described concepts are exploited in order to define an *requirement model* upon a *physical model* in compliance with the proposed meta-model.

The *PhysicalSystem* under consideration is a Water System whose model, i.e. one among all possible *PhysicalSystemModels,* called WaterSystemModel is simply composed by a single *PhysicalComponentModel* of a Tank. The Tank is modeled through different *Attributes* such as the current level of liquid *levelInTank* as well as the height of the tank *tankHeight* (both as a Real *type* and *unit*="m"). Such attributes can be accessed externally (*Public Visibility*), whereas other *Attributes* can be used by the descendants of the Tank (*Protected Visibility*). All those *Attributes* (both with *Public* and *Protected Visibility*) are exploited into a *ComputationalModel* which is defined through different equations (*EquationsSet*) in order to model the *Behavior* of the *Tank*.

Let us assume to define a *RequirementModel* on this specific *PhysicalSystemModel* (the above described WaterSystemModel)~~,~~ in order to verify the following *RequirementAssertion* of a Tank (hereafter we refer to the model of the Tank), whose *Description* is: "The level of liquid in the tank shall never exceed 80% of the tank height" and its *Name* is "LevelOfLiquidInTank". According to the meta-model the status of activation (*StatusOfActivation*) of this *RequirementAssertion* is enabled (*Enabled*) for all the simulation time, and its evaluation period (*EvaluationPeriod*) has a duration equal to the duration of the simulation run without further specific *PreConditions* or *PostConditions*. The *Status* of the *RequirementAssertion* has a *StatusType* set to Boolean, consequently, the allowed status value (*StatusValue*) will range between true and false (or satisfied and notSatisfied).

The fulfillment of this *RequirementAssertion* is defined by a metric (*Metric*) based on the current level of fluid in the Tank, which is measured (*Measure*) as a percentage according to the maximum height of the tank. Consequently, the definition of the *RequirementAssertion* exploits the *levelInTank* and *tankHeight* that are both two *Public Attributes* of the Tank, moreover, an internal parameter, equal to 0.8, is used to express the percentage. Finally, this *Measure* is expressed by adopting as *ComputationalModel* a set of equations (*EquationsSet*). In particular, in this case by a single *Equation,* which is defined according to a threshold *Pattern* (e.g. levelInTank<0.8*tankHeight); a fragment of the possible Modelica (psedo) code is reported below.

```
requirement LevelOfLiquidInTank
  Real levelInTank(unit="m");
  Real tankHeight(unit="m");
  parameter Real limit (start=0.8);
equation
  levelInTank<limit*tankHeight;
end LevelOfLiquidInTank;
```

In the following section some approaches for modeling *System Requirements* through *RequirementAssertions*, based on the presented meta-model, are proposed.

3. Extending the Modelica language for Modeling System Requirements

In this Section different approaches for modeling *system requirements* and how they can be used to verify the intended behavior of the system and validate it through simulation are described. All the approaches are equation-based and, in particular, based upon the Modelica language and ModelicaML (Modelica Modeling Language).

Modelica is a language for equation-based object-oriented mathematical modeling of physical systems (e.g., systems containing mechanical, electrical, electronic, hydraulic, thermal, control, electric power components) with *acausal* features, which allows defining models in a declarative manner [3].

ModelicaML is an UML profile, which is based on the SysML/UML profile and reuses its artifacts required for system specification. ModelicaML reuses several diagrams types from SysML without any extension, extends some of them, and also provides several new ones. ModelicaML diagrams are grouped into four categories: *Requirement, Structure, Behavior* and *Simulation* [13].

Although both Modelica and ModelicaML are expressly designed for modeling systems in a coherent framework based on an equation approach, they do not yet provide concepts to be used in order to represent and trace the occurrence of dysfunctional/abnormal behavior (such as faults and failures), that is to say, an observable deviation from the intended behavior at the system boundary [2, 6, 7].

In this perspective, the exploitation of the *meta-model* presented in the previous Section can be used to enrich both the Modelica language and ModelicaML to provide them with the capability of modeling system requirements and to enable model checking. In particular, different approaches are proposed and discussed in the following subsections based on the two main concepts of *requirement assertion* (see Section 2) and *fulfill* and some variants of them.

3.1 Approach A

In this approach the formal concepts of *requirement* and *fulfill* are defined as follows:

- *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* (according to the *SRs* and the *SD*).
- *fulfill:* which expresses the entailment relationship between *PhysicalComponentModels* and a *requirement,* as well as among *requirements*. Moreover, it provides the propagation process of an assessment among *RequirementAssertions*.

An example model, which illustrates these concepts, is shown in Figure 3. In particular, after the declaration of the instances of both *PhysicalComponentModels* and *RequirementAssertions* their relationship is established according to the following *five connection-rules*:

1. the connections enabled through the *connect* construct among *PhysicalComponentModels* are defined to build the *SD* of the *PhysicalModel;*
2. the connections enabled through the *connect* construct among a *PhysicalComponentModel* and an *RequirementAssertion* are used to provide outputs coming from *PhysicalComponentModels* in input to *RequirementAssertions.*
3. the exploitation of the *fulfill* keyword is used to define which instance of an *RequirementAssertion* has to be satisfied/related from at least one specific instance of a *PhysicalComponentModel.*
4. the exploitation of the *fulfill* keyword is used among *RequirementAssertions* to enable the propagation mechanisms of assessment among them;
5. If *A1,..,An* are *RequirementAssertions* and *C1,..,Cm* are *PhysicalComponentModels*, then we can define *(A2,..,An,C1,..,Cm)fulfill(A1)*, where *A1* is satisfied if and only if *C1,..,Cm* satisfy *A1* as well as *A2,..,An* are all satisfied (*fulfill* follows the rule of the And logic).

As we can see in Figure 3, the *connect* construct, which is already available in the Modelica language, is used not only among *PhysicalComponentModels* but also between a *RequirementAssertion* and a *PhysicalComponentModel*. Even though the *connect* construct allows to define connections among attributes of two or more components in an acausal way [3], in this approach some restrictions are defined on it. As an example, the connection is only able to provide inputs from a physical component towards a *RequirementAssertion*. The reason for such a restriction is to prevent a *RequirementAssertion* from providing input to a *PhysicalComponentModel* and consequently affecting its behavior.

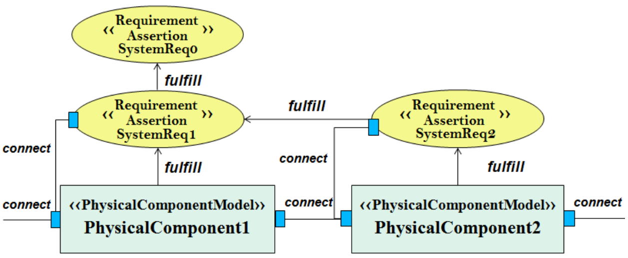

Figure 3. A verification model based on *requirement assertion* and *fulfill*.

3.2 Approach B

Whilst the above mentioned approach allows to model requirements in a simple and intuitive fashion, with the help of a minimal set of new concepts (i.e. *requirement assertion* and *fulfill*), the addition of extra connections between *requirement assertions* and components through *connect*, could make the *ESD* overly verbose and difficult to read from a visual representation point of view, thus complicating the maintainability of the source code.

Therefore, an alternative approach is a variant of the previous one in which along with the keyword *requirement*, another concept (and another keyword) called *On,* which is only visible in the source code of a *RequirementAssertion,* is introduced. Similar to the *extends* construct, but with some restrictions on the inherited elements, the *On* keyword enables a *requirement* to be defined on specific *PhysicalComponentModels.* Such a *requirement* will inherit the *attributes* on which it will carry out the processing.

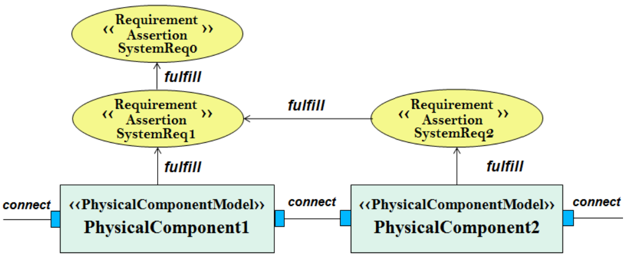

Figure 4. Modeling Requirements using the On construct.

The process to build the *ESD* follow the *five-connection-rules*, which have been described in Section 3.1 except for the rule number 2; in this way:

- it allows to avoid the exploitation of extra *connect* (between *PhysicalComponentModels* and *RequirementAssertions*) in order to provide input values coming from *constants*, *parameters* or *variables* of physical components towards an *requirement*. Indeed, such relationships are established during the definition of the *RequirementModel* through the exploitation of the *On* keyword;
- it allows to avoid of having too many connections into a graphical representation, as it is in Figure 4, by also reducing the lines of the source code of the *Extended System Design.*

The concept of *fulfill* is that explained in the previous section.

3.3 Approach C

Often, it is necessary to have additional mechanisms for generating dysfunctional/abnormal behavior in a physical component, so as to assess the consequences on the whole system.

To this end, approach C proposes the possibility of altering the values of the components starting from the B approach and adding the new notions of *tester entity/component entity* and the *supersede* keyword. The *tester entity* can be seen as a specific component that is defined on a *PhysicalComponentModel* and which is able to generate outputs (e.g. signals, events or values) according to specific functions and inject them into the *PhysicalComponentModel* in order to alter its intended/nominal behavior (expected values). The *supersede* keyword enables the mechanism to create a reference between an instance of a *tester entity* and an instance of a *PhysicalComponentModel*. In particular, the following rules define the semantics of the *supersede* keyword and how to use it:

1. the exploitation of the *supersede* keyword is used to define which specific instance of a *PhysicalComponentModel* could be compromised by which specific instance of a *Tester component*;
2. If *T1,..,Tn* are *Tester components* and *C* is a *PhysicalComponentModel*, then we can define *(T1,..,Tn)supersede(C)*, where the operation work of *C* could be influenced only by one among the *T1,..,Tn Tester components* (*supersede* follows the rule of the XOr logic).

RequirementAssertions can monitor the occurrence of abnormal/dysfunctional behavior in physical components; the *fulfill* relationship is exploited by the *RequirementAssertion* to check the impact and the consequently propagation of possible unexpected values in a component on other components (see Figure 5). The *On* keyword enables both *RequirementAssertions* and *Tester components* to have access directly to the attributes of the physical component models on which they are defined.

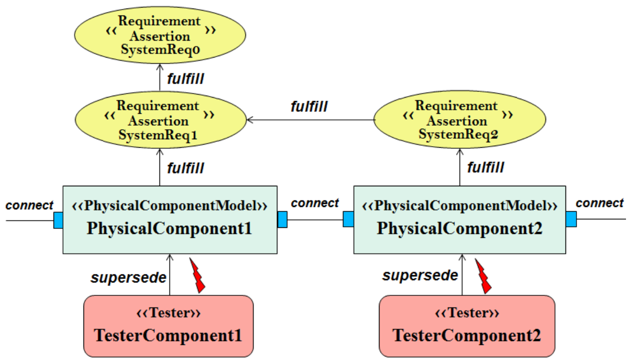

Figure 5. Requirements and Tester component for the dysfunctional behavior analysis.

4. A case study

In this Section, a case-study is first described and then used to evaluate some of the solutions which have been proposed in the previous Section; for this purpose, both the ModelicaML diagrams and the Modelica code are presented; finally, the pros and cons of each solution are discussed.

4.1 System Description

The possible implementation of the previously presented approaches along with the significant reduction of programming and implementation efforts to model *system requirements* as well as the increased readability, are demonstrated through a typical case study of a Tank System.

The Tank System is composed by four main physical components: a *Source* component, a *Tank* component, a *LevelController* component and a *Sink* component. The *Source* component produces a flow of liquid, which is taken in input by the *Tank* component. The *Tank*, which is managed through the *LevelController* component, provides in output a liquid flow according to the control law defined by the *LevelController*. The *Sink* is the component where a part of liquid is sent.

After an analysis of the *URs*, the following main *SRs* (and many others) have been identified:

- *System_Requirement_1*: the system has to be composed by one Source Component, one Sink Component, at least one Tank Component and at least one LevelCotroller Component;
- *System_Requirement_2:* each tank has to provide one port called *qIn* in order to receive flow from another possible Tank Component (or from the Source component if it is the first Tank component in the chain);
- *System_Requirement_3:* each tank has to be connected to its own LevelController component;
- *System_Requirement_3_1*: each Tank component has to provide a port called *tSensor* in order to provide signal to the LevelController component;
- System_Requirement_*4*: the Source component has to provide a flow port called *qOut*;
- *System_Requirement_4_1*: the liquid flow produced by the Source component has to be equal three times the initial flow after 150 seconds;
- *System_Requirement_5_1*: the liquid flow produced by the Source component should be less than 10 m^3/s.
- *System_Requirement_5_2*: the role of the LevelController should be verified by exploiting both the *h* level from the Tank component and the *qOut* flow.
- *System_Requirement_*5_3: the validity of both the *tActuator* (Out-flow) and the *outFlowArea* values should be checked according to a specified function;
- *System_Requirement_5_4*: both the *h* level and the *tSensor* should provide the same values;
- *System_Requirement_5_5*: the *h* level coming from the Thank should be checked according to a specified function.

Starting from the SRs above described, the SD of the Tank System has been defined as shown in Figure 6, whereas in the following, a fragment of the Modelica code used to implement the Tank System is reported.

```
package PhysicalComponentModel
model Source;
  LiquidFlow qOut;
  parameter Real flowLevel=0.02;
equation
  qOut.lflow = if time>150 then
  3*flowLevel else flowLevel;
end Source;
model Tank
  ReadSignal tSensor;
  ActSignal tActuator;
  LiquidFlow qIn;
  LiquidFlow qOut "Connector, flow (m3/s)
  through output valve";
  parameter Real area(unit="m2")=0.5;
  parameter Real flowGain(unit="m2/s")=
  0.05;
  parameter Real minV = 0,maxV = 10;
  Real actuatorControllerV;
  Real outFlowArea(unit="m");
  Real h(start=0.0, unit="m");
equation
  der(h)=(qIn.lflow-qOut.lflow)/area;
  actuatorControllerV=flowGain*
  tActuator.act;
  qOut.lflow = LimitValue(minV, maxV,
  actuatorControllerV);
  tSensor.val=h;
  outFlowArea=-qOut.lflow/flowGain;
end Tank;
…
end PhysicalComponentModel;
```

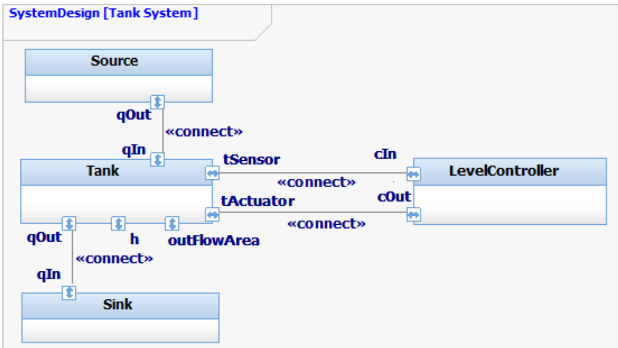

Figure 6. The System Design (*SD*) of the Tank System.

Moreover, starting from the *SD* of the Tank System, the following *Requirement Assertions* have been defined; they should be represented and verified during simulation in order to ensure the proper operation of the system. In the next subsections some of the proposed approaches are applied for the modeling of requirements.

4.2 Exploiting the A Approach

In this example, starting from the *System Requirements* specified in the previous subsection, a set of *RequirementAssertions* can be defined on the *SD* of the Tank System by exploiting the Approach A; in particular:

- *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).
- *RequirementAssertion_2*: *ControlOutFlow*, which takes in input the *h* level from the Tank component and the *qOut* flow to validate the role of the *LevelController*; moreover, to be valid it must be fulfilled by both the *RequirementAssertion_3* and the *RequirementAssertion_4*.
- *RequirementAssertion_3*: *ActuateOutFlow*, which takes in input both the *tActuator* (Out-flow) and the *outFlowArea*, checks if the *outFlowArea* value is proportional at the *tActuator* signal.
- *RequirementAssertion_4*: *SenseLevel*, which takes in input both the *h* level and the *tSensor*, checks if the sensor output is equals to the *h* level (i.e. lLevel=sensorOuput).
- *RequirementAssertion_5*: *ControlLevel*, which takes in input the *h* level coming from the Tank component,

checks if hLevel<9 and hLevel>5; moreover, to fulfill the *RequirementAssertion_5*, both the state of *RequirementAssertion_1* and of *RequirementAssertion_2* have to be satisfied.

Figure 7 shows an example of ModelicaML-based notation for the different concepts. In the following, some code fragments of the *RequirementModel* and, in particular, the implementation of *RequirementAssertion_1* and of *RequirementAssertion_5,* introducing the new keyword *requirement*, are reported.

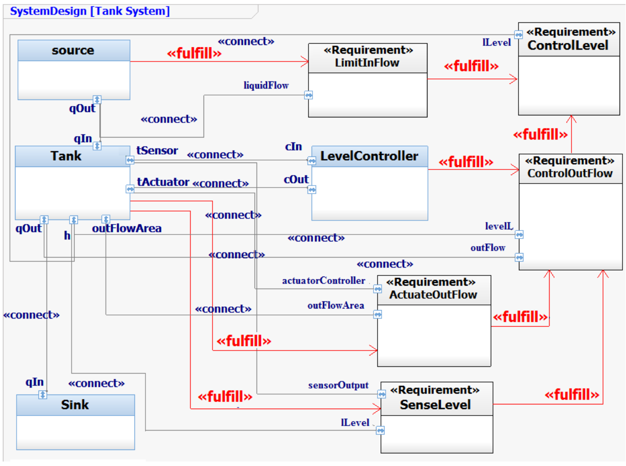

Figure 7. Approach A for modeling requirements of the Tank System.

```
package RequirementModel
requirement Requirement1
  Real liquidFlow; "qOut of Source"
  parameter Real maxLevel=10;
equation
  if liquidFlow<=maxLevel then
    Status.satisfied;
    …
end Requirement1;
…
requirement Requirement5
  Real lLevel;
  parameter Real Lmin=5, Real Lmax=9;
equation
  if lLevel<Lmax and lLevel>Lmin then
    …
end Requirement5;
end RequirementModel;
```

In the snippet of code shown subsequently, both the *PhysicalSystemModel (or SD)* and the *RequirementModel* are composed.

```
model ExtendedSystemDesign
  //PhysicalComponentModels
  Source source;
  Tank tank1(area=1);
  …
  //RequirementComponents
  Requirement1 limitInFlow;
  …
  Requirement5 controlLevel;
equation
  //Connection among PhysicalComponents
  connect(source.qOut,tank1.qIn);
  …
  //fulfill connections
  (source)fulfill(limitInFlow);
  (tank1)fulfill(actuateOutFlow);
  (tank1)fulfill(senseLevel);
  (limitInFlow,controlOutFlow)
  fulfill(controlLevel);
  (levelController,actuateOutFlow,
  senseLevel)fulfill(controlOutFlow);
  //connection between physical
  //components and requirements
  connect(tank1.h,controlLevel.L);
  connect(tank1.h,senseLevel.lLevel);
  connect(source.qOut,limitInFlow.
  liquidFlow);
  …
end ExtendedSystemDesign;
```

By adopting this approach, the *RequirementModel* is completely decoupled from the *PhysicalSystemModel* of the system under consideration. Indeed, a *requirement model* only requires input values of specific types, regardless of the type and the number of components that the values come from. This means that a *requirement model* could be re-used to validate physical components belonging to different *SD*, although the semantics of such physical components could be completely different. The link between the *RequirementModel* and *PhysicalSystemModel*, occurs only in the *ESD*, through the *fulfill* relationships which govern the assignment of a component to a requirement, while the inputs to be sent to the requirement are provided by the *connect* construct.

4.3 Exploiting the B Approach

In this example the Approach B is exploited to represent the same Tank System including the *RequirementAssertions* described in the previous subsection. Figure 8 shows the related ModelicaML-based notation of such a modeling approach. As we can see, the diagram is less crowded with connections and consequently easier to read.

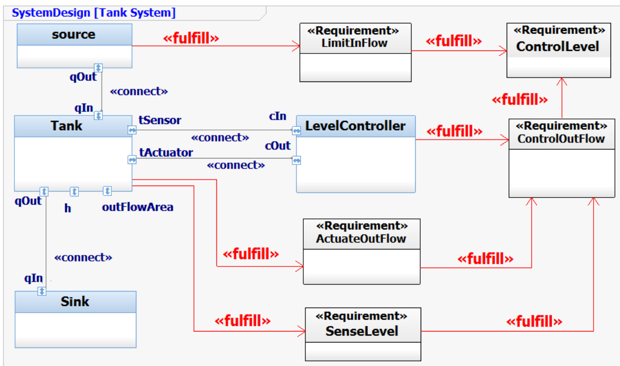

Figure 8. Approach B for modeling requirements of the Tank System

As it is shown in the next code fragments illustrating the source code of *Requirement_Assertion_1* and of *Requirement_Assertion_5*, both the keyword *requirement* along with the *On* keyword are combined for the definition of each *requirement*. Specifically, starting from the *Source* model, *Requirement1* is defined on it; this means that *Requirement1* is able to use (read-only) all the

Public attributes, which have been defined by the *Source* model. In particular, the *qOut* attribute of the *Source* model can be used by *Requirement1* without further referencing or connections with the *Source* model.

```
package RequirementModel
requirement Requirement1 On Source
  parameter Real maxLevel=10;
equation
  if Source.qOut<=maxLevel then
    Status.satisfied;
  else
    Status.notSatisfied;
  end if;
…
end Requirement1;
…
requirement Requirement5 On Tank
  parameter Real Lmin=5, Lmax=9;
equation
  if Tank.h<Lmax and Tank.h>Lmin then
  …
end Requirement5;
end RequirementModel;
```

As for the previous example a fragment of source code combining both the *PhysicalSystemModel* and the *RequirementModel* is presented. As we can see, no connections which use the *connect* construct between a *PhysicalComponentModel* and a *requirement component*, are present in the source code of the *ESD* model.

```
model ExtendedSystemDesign
  //PhysicalComponentModels
  Tank tank1(area=1);
  Sink sink1;
  …
  //RequirementComponents
  Requirement1 limitInFlow;
  …
  Requirement5 controlLevel;
equation
  //Connections among PhysicalComponents
  connect(source.qOut,tank1.qIn);
  …
  //fulfill relationships
  (source)fulfill(limitInFlow);
  (tank1)fulfill(actuateOutFlow);
  (tank1)fulfill(senseLevel);
  (levelController,actuateOutFlow,
  senseLevel)fulfill(controlOutFlow);
  (limitInFlow,controlOutFlow)
  fulfill(controlLevel);
end ExtendedSystemDesign;
```

By adopting this approach, the *RequirementModel* is not completely decoupled from the *PhysicalSystemModel* (this make *requirement assertions* less flexible and less reusable) as it knows *Public Attributes* that are defined in the *PhysicalSystemModel*. On the other hand, it allows for a more immediate exploitation making the *ESD* model easier to read by hiding the details of the matching between the *PhysicalSystemModel* and the *RequirementModel*. Indeed, as it is shown both in Figure 9 and through the code of the *ESD*, only the *fulfill* relationships are visible, while the *connection* (through the *connect* construct) among *PhysicalComponentModels* and *RequirementAssertions* are not part of the *ESD*.

4.4 Exploiting the C Approach

The Approach C is adopted to model the previously described *requirement assertions* on the Tank System. Additionally, the possibility of modeling entities that alter the intended behavior of components, and consequently of the system, is taken into account by exploiting *tester entities/components.*

In this section, three *tester components* have been defined in order to illustrate their use:

- *AlterSourceFlow* and *AlterSourceFlow2* on the *Source* component, respectively producing the double of the liquid in the first case and producing a negative value of liquid in the second case.
- *AlterOut* on the *Tank* component, where the LimitValue function has been removed from the behavior of the tank.

In the following, some code fragments describing the *TesterModel* and, specifically, the source code of the *AlterSourceFlow* and of the *AlterOut* are reported.

```
package TesterModel
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;
end TesterModel;
```

As we can see in the source code below, the link between *PhysicalSystemModel* and *TesterModel* is defined in the *ESD* through the keyword *supersede.* In Figure 9 a ModelicaML-based notation for such a modeling approach, introducing both *Requirement* and *Tester components* as well as physical components is depicted.

```
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;
```

It is worth noting that one possible variant of the Approach C consists in defining the relationships between a *PhysicalComponentModel* and a *Tester component* in the *ESD* by using the construct *connect*, in order to avoid the exploitation of the *On* keyword during the definition of the tester components in the *TesterModel*. By adopting this version (similar to the A Approach), the *PhysicalSystemModel* will be completely decoupled from both the *RequirementModel* and the *TesterModel*.

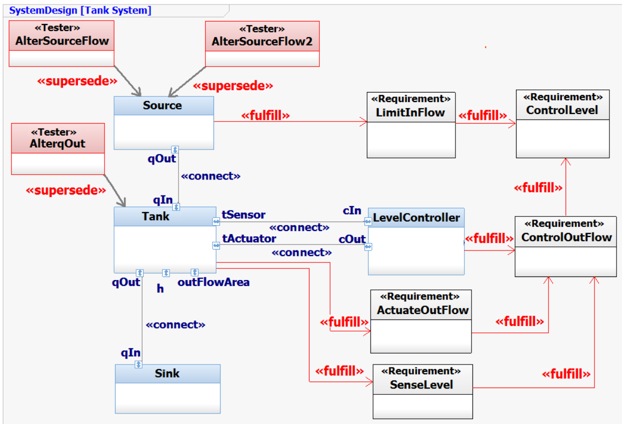

Figure 9. Approach C for modeling requirements of the Tank System.

5. Conclusions and future works

The paper focused on the modeling of requirements in an equation-based context. In particular, a reference *meta-model* for representing *System Requirements* in terms of *RequirementAssertions* has been defined. Then, three different approaches for the modeling of *System Requirements* that adhere to the proposed *meta-model*, have been outlined. All of them aim to provide support for model verification by defining extensions of the Modelica language, and, one of them also aim to extend such model verification by supporting the modeling of system failures and thus allowing to analyze the behavior of the system in presence of faults.

Finally, the exploitation of the proposed approaches in a case study concerning a Tank System has allowed to compare their advantages and disadvantages as well as to appreciate their effectiveness and usability in the system modeling phases.

This work is part of an ongoing research project (MODRIO project – ITEA 2) [10] aiming at developing a model-based approach for system requirements verification and fault tree analysis through Modelica extensions for Requirements modeling and Safety analysis.

Ongoing research efforts are devoted to improving the proposed approaches through both their implementation in *OpenModelica* [12] and their integration in a full-fledged Systems Engineering development process [5] along with an extensive experimentation in the analysis of systems in different application domains such as automotive, railway, avionics and energy.

Acknowledgements

This work has been supported by ITEA 2 MODRIO project. Andrea Tundis has been supported by a grant funded in the framework of the “POR Calabria FSE 2007/2013”.

References

[1] T. Clark, P. Sammut, and J. Willans. Applied metamodelling: a foundation for language driven development (Second Edition), 2008.

[2] R. Cressent, V. Idasiak, F. Kratz, and P. David. Mastering safety and reliability in a model based process. *Proc. of the Reliability and Maintainability Symposium (RAMS)*, Lake Buena Vista (FL, USA), January 2011.

[3] P. Fritzson, Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, Wiley IEEE Press, 944 pages, February 2004.

[4] P. Fritzson. Integrated UML-Modelica Model-Based Product Development for Embedded Systems in OPENPROD. *Proc. of the 1st Workshop on Hands-on Platforms and tools for model-based engineering of Embedded Systems (Hopes’2010),* Paris, June 15, 2010.

[5] A. Garro and A. Tundis. Enhancing the RAMSAS method for Systems Reliability Analysis through Modelica. *Proc. of the 7th Workshop on Model-Based Product Development (MODPROD)*, Linköping (Sweden), 5-6 February, 2013.

[6] A. Garro and A. Tundis. Modeling and Simulation for System Reliability Analysis: The RAMSAS Method. *Proc. of the 7th IEEE International Conference on System of Systems Engineering (IEEE SoSE)*, Genova (Italy), July 16-19 2012.

[7] L. Grunske and B. Kaiser. Automatic Generation of Analyzable Failure Propagation Models from Component-Level Failure Annotations. *Proc. of the 5th Int. Conf. on Quality Software (QSIC)*, Melbourne (Australia), September 2005.

[8] H. P. Hoffmann. System Engineering Best Practices with Rational Solution for Systems and Software Engineering. February 2011. http://www.ibm.com/.

[9] A. Jardin, D. Bouskela, T. Nguyen, N. Ruel, E. Thomas, R. Schoenig, S. Loembé and L. Chastanet. Modelling of System Properties in a Modelica Framework. *Proc. of the 8th International Modelica Conference*, TU Dresden, March 20-22, 2011.

[10] ITEA 2 Projects: MODRIO - http://www.itea2.org/.

[11] F. Liang, W. Schamai, O. Rogovchenko, S. Sadeghi, M. Nyberg and P. Fritzson. Model-based Requirement Verification: A Case Study. *Proc. of the 9th International Modelica Conference (Modelica'2012)*, Munich (Germany), September 3-5, 2012.

[12] OpenModelica - Open Source Modelica Consortium (OSMC) - https://www.openmodelica.org/.

[13] OpenModelica Project: ModelicaML - A UML Profile for Modelica. www.openmodelica.org/modelicaml.

[14] W. Schamai, P. Fritzson, C.J.J. Paredis, P. Helle. ModelicaML Value Bindings for Automated Model Composition. *Proc. of the Symposium on Theory of Modeling and Simulation (DEV’12),* Orlando, FL (USA) March 26-29, 2012.

[15] W. Schamai, P. Helle, P. Fritzson, and C. Paredis. Virtual Verification of System Designs against System Requirements. *Proc. of 3rd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES’2010)*, Oslo (Norway), October 4, 2010.