Towards Unified System Modeling and Simulation with ModelicaML: Modeling of Executable Behavior Using Graphical Notations

Wladimir Schamai[1], Peter Fritzson[2], Chris Paredis[3], Adrian Pop[2]
[1]EADS Innovation Works, Hamburg, Germany
[2]PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, SE-581 83 Linköping, Sweden
[3]Georgia Institute of Technology, Atlanta, USA
wladimir.schamai@eads.net, chris.paredis@me.gatech.edu, {petfr, adrpo}@ida.liu.se

Abstract

This paper is a further step towards application of the Model-Based Systems Engineering (MBSE) paradigm, using standardized, graphical, and executable system modeling languages. It presents further development of Modelica graphical Modeling Language (ModelicaML), a UML Profile for Modelica, which enables an integrated modeling and simulation of system requirements and design (for systems including both hardware and software). This approach combines the power of the OMG UML/SysML standardized graphical notation for system and software modeling, and the modeling and simulation power of Modelica. It facilitates the creation of executable system-specification and analysis models that can simulate time-discrete (or event-based) and time-continuous system behavior.
Keywords: Modelica, ModelicaML, UML, SysML, graphical modeling, system requirements, system design.

1 Introduction

UML/SysML [2],[4] and Modelica [1] are object-oriented modeling languages. Both provide means to represent a system as objects and to describe its internal structure and behavior. SysML is a UML profile for systems modeling. It facilitates efficient capturing of relevant system requirements, design, or test data by means of graphical formalisms, crosscutting constructs and views (diagrams) on the model-data. Modelica is defined as a textual language with standardized graphical annotations for model icons, and is designed for efficient simulation of system dynamic behavior.

1.1 Paper Structure

This paper first presents the motivation and previous work done on the integration of UML/SysML and Modelica, followed by a brief description of UML/SysML, Modelica, and ModelicaML languages. Section 4 summarizes the basic mapping between UML and Modelica, which results in the ModelicaML profile, and provides examples of applications. Section 5 discusses graphical notations for Modelica behavioral concepts. Sections 6 and 7 discuss ModelicaML concepts not present in Modelica. Sections 8, 9 and 10 address the supporting modeling, code generation and simulation environment.

2 Motivation

By integrating Modelica and UML/SysML the UML/SysML's strength in graphical and descriptive modeling is complemented with Modelica's formal executable modeling for analyses and trade studies. Vice versa, Modelica will benefit from using the selected subset of the UML/SysML graphical notation (visual formalisms) for editing, reading and maintaining Modelica models.

Graphical modeling, as promoted by the OMG [13], promises to be more effective and efficient, regarding editing, human reader perception of models, and maintaining models compared to a traditional textual representation. A unified, standardized graphical notation for systems modeling and simulation will facilitate the common understanding of models for parties involved in the development of systems (i.e., system-engineers, designers, and testers; software-developers, customers or stakeholder).

Existing UML/SysML formalisms are typically translated into (and limited to) the time-discrete or event-based simulation of a system or software. This limitation disappears when Modelica comes into play. UML/SysML models will then be of a higher expressiveness and correctness, because they will become executable while covering simulation of hardware and software, with integrated continuous-time and event-based or time-discrete behavior.

3 Background and Related Work

Some research work previously done has already identified the need for integrating UML/SysML and Modelica, and has addressed integration issues to some extent. For example, [7] has identified the basic mapping of the structural constructs of Modelica to SysML. It also pointed out that the SysML Parametrics concept is not sufficient for modeling the equation-based behavior of a class. By contrast, [9] leverages the SysML Parametrics concept for the integration of continuous-time behavior into SysML models. [8] presents a concept to use SysML for integrating models of continuous-time dynamic system behavior with SysML information models representing systems engineering problems, and provides rules for graph-based bidirectional transformation of SysML and Modelica models.

The main focus of this paper is the representation of Modelica behavioral constructs using graphical notations and formalisms that are based on a subset of UML, which can be translated into executable Modelica code.

3.1 OMG Systems Modeling Language (SysML)

SysML [4] is a UML profile[1] and a general-purpose systems modeling language that enables systems engineers to create and manage models of engineered systems using graphical notations. SysML reuses a subset of UML 2.1 [2] constructs and extends them by adding new modeling elements and two new diagram types. Through these extensions, SysML is capable of representing the specification, analysis, design, verification, and validation of any engineered system.

MBSE promotes the usage of models as primary engineering artifacts. However, textual requirements are still the main vehicle for communicating and agreeing on system specification in a system development process. SysML provides mechanisms to include textual requirements into models. In doing so, traceability of textual requirements to design artifacts and test cases is facilitated.

The logical behavior of systems is captured in SysML through a combination of activity diagrams, state machine diagrams, and/or interaction diagrams. In addition, SysML includes Parametrics to support the execution of constraint-based behavior such as continuous-time dynamics in terms of energy flow. However, the syntax and semantics of such behavioral descriptions in Parametrics have been left unspecified to interoperate with other simulation and analysis modeling capabilities.

3.2 The Modelica Language

Modelica is an object-oriented equation-based modeling language primarily aimed at physical systems. The model behavior is based on ordinary and differential algebraic equation (OAE and DAE) systems combined with discrete events, so-called hybrid DAEs. Such models are ideally suited for representing physical behavior and the exchange of energy, signals, or other continuous-time or discrete-time interactions between system components.

Modelica models are similar in structure to UML/SysML models in the sense that Modelica models consist of compositions of sub-models connected by ports that represent energy flow (undirected) or signal flow (directed). The models are acausal, equation-based, and declarative. The Modelica language is defined and maintained by the Modelica Association [1] which publishes a formal specification but also provides an extensive Modelica Standard Library that includes a broad foundation of essential models covering domains ranging from (analog and digital) electrical systems, mechanical motion and thermal systems, to block diagrams for control. Finally, it is worth noting that there are several efforts within the Modelica community to develop open-source solvers, such as in the OpenModelica project [12].

3.3 ModelicaML

This paper presents the further development of the Modelica graphical Modeling Language (ModelicaML), a UML profile for Modelica. The main purpose of ModelicaML is to enable an efficient and effective way to create, read or understand, and maintain Modelica models reusing notations that are also used for software modeling. ModelicaML is defined as a graphical notation that facilitates different views (composition, inheritance, behavior) on system models. It is based on a subset of the OMG Unified Modeling Language (UML) and reuses concepts from the OMG Systems Modeling Language (SysML). ModelicaML is designed towards the generation of Modelica code from graphical models. Since the ModelicaML profile is an extension of the UML meta-model it can be used for both: Modeling with standard UML and with SysML[2].

UML/SysML provide the modeler with powerful descriptive constructs at the expense of loosely defined

[1] UML profiles allow domain-specific extensions of UML by means of stereotypes.

[2] SysML itself is also a UML Profile. All stereotypes that extend UML meta-classes are also applicable to the corresponding SysML elements.

semantics that are marked as "semantic variation points" in the UML/SysML specifications. The intention of ModelicaML is to provide the modeler with powerful executable constructs and precise execution semantics that are based on the Modelica language.

Therefore, ModelicaML uses a limited set of the UML, extends the UML meta-model (using the UML profiling mechanism) with new constructs in order to introduce missing Modelica concepts, and reuses concepts from the SysML. However, like UML and SysML, ModelicaML is only a graphical notation. ModelicaML models are eventually translated into Modelica code. Hence, the execution semantics are defined by the Modelica language and ultimately by a Modelica compiler that will translate the generated Modelica code into an executable form.

4 Representing Modelica Structural Constructs in ModelicaML

The class concept is the basic structural unit in Modelica. Classes provide the structure for objects and contain equations, which ultimately serves as the basis for the executable simulation code. The most general kind of class is "model". Specialized categories of classes such as "record", "type", "block", "package", "function" and "connector" have most of the properties of a "model" but with restrictions and sometimes enhancements.

In UML the "Class" is the main structural unit which can have behavior. A non-behavioral concept is the "DataType".

The following table summarizes the mapping of the structural Modelica constructs to UML. The details of the associated properties of the Modelica constructs are left out.

Table 1: Mapping of Modelica structural constructs to UML

Modelica	UML
package	UML::Package
model, block	UML::Class
connector, record, type	UML::DataType
component of type connector	UML::Port
variable, component	UML::Property
extends relation	UML::Generalization
connection clause	UML::Connector

The mapping listed above is specified by [11] and has been implemented as a UML profile in a the Eclipse-based open-source tool Papyrus UML [10]. Modelica constructs are represented using stereotypes (extensions of the UML meta-model) with required properties (attributes) that are specific to Modelica.

It is subject to the current implementation work of the ModelicaML editor to reflect the Modelica language wording, so that the Modelica modeler will not be forced to work with UML/SysML wording. Based on this mapping it is also possible to import existing Modelica models (or libraries) into ModelicaML models, to represent them using graphical notations and to reuse them the same way as is done in Modelica tools.

The following figures present examples of tank systems inspired from [3], sections 12.2.3, 12.2.4 and 12.2.5. The only means to represent Modelica code graphically is the Modelica *connection diagram* (see the two tanks example on the Figure 1). A Connection Diagram shows Modelica class components (typically depicted as domain specific icons with connectors) of the class and their interconnection (connect clauses) as depicted in the figure below. The graphical notation is defined by the Modelica modeler (e.g. the developer of a library) and is not standardized by the language specification; it is usually specific to the domain of application.

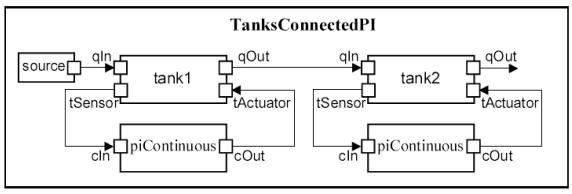

Figure 1. Two Tanks System example, [3] page 391.

The corresponding ModelicaML notation is based on the UML Composite Diagram as illustrated in Figure 2.

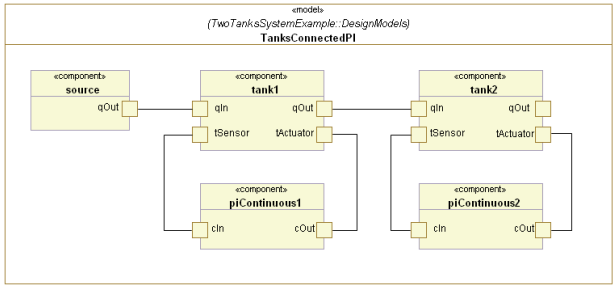

Figure 2. Example of ModelicaML notation (connections)

By contrast, UML defines different types of diagrams, which enable different visual views on the model data, such as inheritance, classes that are nested, the composition of a class or interconnection of components of a class or its defined behavior.

Moreover, the graphical notation is not specific to a domain (although it is possible to include domain specific icons into the class compartment). It is abstracted

from the domain. Thanks to such an abstracted, unified notation, engineers from different domains or disciplines will share a common understanding of the model.

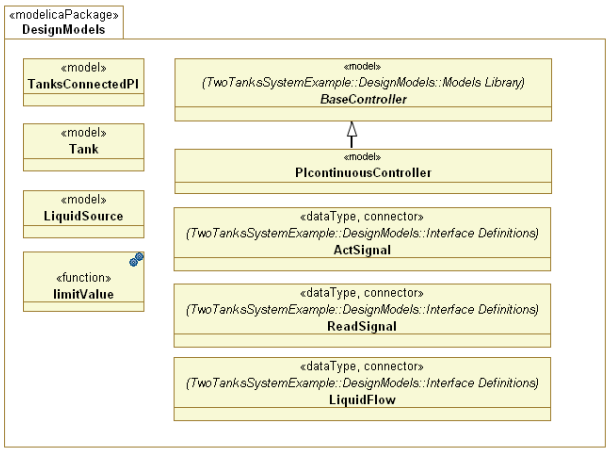

Figure 3. Example of ModelicaML notation (packages, classes)

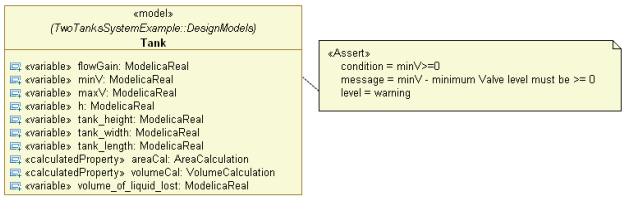

Figure 4. Example of ModelicaML notation (class, components, asserts)

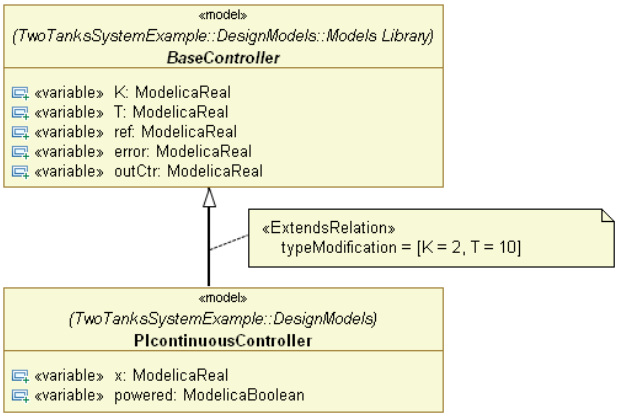

Figure 5. Example of ModelicaML notation (inheritance)

In particular the inheritance (extension) graphical representation (Figure 5) is useful if there are multiple levels of inheritance.

5 Representing Modelica Behavioral Constructs in ModelicaML

Modelica does not define any graphical notation for representing the behavior of a class. Instead, the behavior of a Modelica Class is specified by its equations or algorithm statements (including all conditional constructs) which are provided as text.

In addition to basic equations or statements Modelica defines conditional constructs, which are allowed in both equation and algorithm sections, and can have nested constructs or not.

A good match for representing conditional constructs in UML is the Activity Diagrams notation including decision nodes and conditional control flow constructs. The following figures present notations that is used for representing Modelica conditional "if-statement". This notation is used for both "if/when" statements and "if/when" equations. The execution semantics of such Activity Diagrams are the same as for the conditional statements or equations in Modelica. The conditions are evaluated at each time instance. The actions, presented on the diagram are not time-consuming activities; their execution does not take any simulation time.

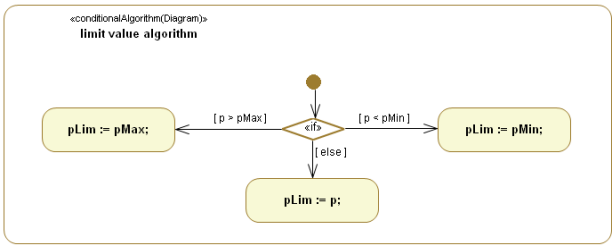

Figure 6. Conditional "if-statement" in ModelicaML

Modelica is a specific language in the context of UML/SysML. For the capturing code of specific languages UML provides opaque constructs which are defined as "A behavior with implementation-specific semantics." (see [2], p.446). In UML, any opaque construct has an attribute "language" (in our case it will be set to "Modelica") indicating how to interpret the code that is entered into the further attribute "body".

Since the UML is an object-oriented modeling language (encapsulating data and behavior), the UML meta-model defines that a classifier can have ownedBehavior (0..*). A behavior in UML can be represented by: State Machine, Activity, Interaction or OpaqueBehavior (see Figure 7).

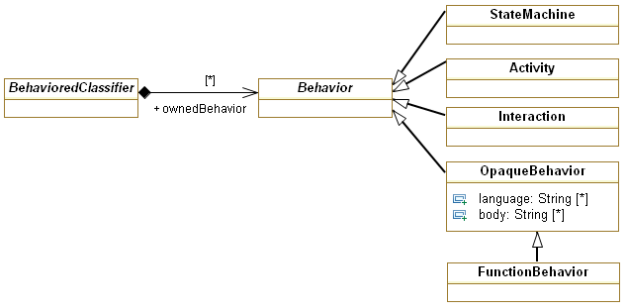

Figure 7. Extract from the UML meta model, from [2] page 426.

A Modelica model can have (0..*) (from zero to any number) of equation or algorithm sections, which corresponds to the ownedBehavior associations of a Classifier in UML. Conditional equation or algorithm statements can be modeled using a subset of the UML Activity Diagram as illustrated above. Alternatively, the modeler may use the OpaqueBehavior for capturing pure textual Modelica code as illustrated in the following figure.

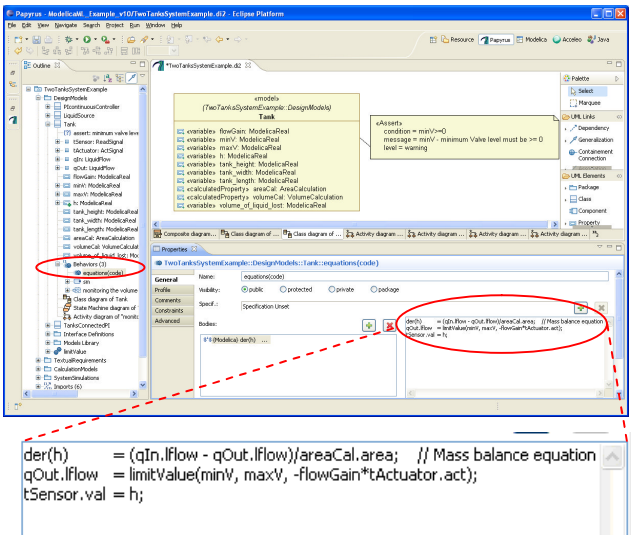

Figure 8. Modelica textual code in ModelicaML models

If conditional equations or algorithm statements are modeled using UML Activity Diagrams, the actual equations or statements are captured using UML OpaqueAction as depicted in the following figure.

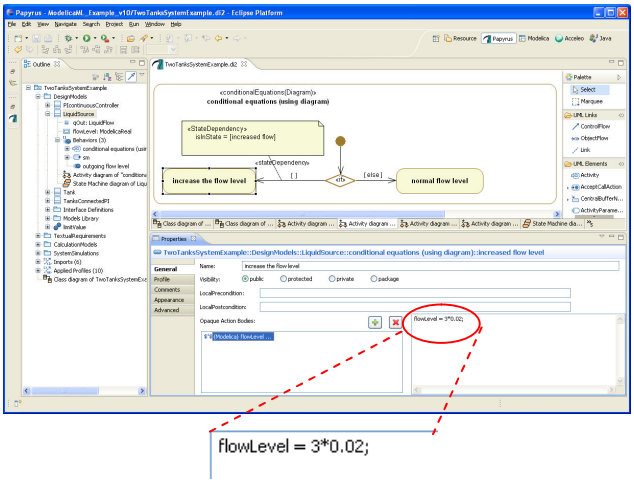

Figure 9. Modelica code in ModelicaML diagrams

[11] summarizes the mapping of the Modelica behavioral constructs to the UML in detail.

6 ModelicaML Concepts Not Provided by Modelica

UML State Machines are typically used for modeling the reactive (event-based) behavior of objects. In ModelicaML the State Machines are used for modeling explicit states or modes of a system or its components. The behavior defined by a State Machine is translated into Modelica algorithm code. Following the principles of a synchronous language the following restrictions are imposed on the semantic of the State Machines as used in ModelicaML:

- The system is at any time in a defined state (note, that the state machines include composite and parallel states, which means that it can be in multiple sub-states at the same time)
- Events and transitions between states take no simulation time. For that reason the effect actions on transitions are not allowed.
- Any behavior that is executed when the state is entered or exited takes no simulation time as well.
- Even though the system will stay in certain states for a time the Do-behavior of a state is also not time-consuming.

Consider the State Machine defined for the tank. Depending on the level of liquid in the tank (represented by the variable “h”) we can define that the tank is empty, partially filled or even in an overflow state.

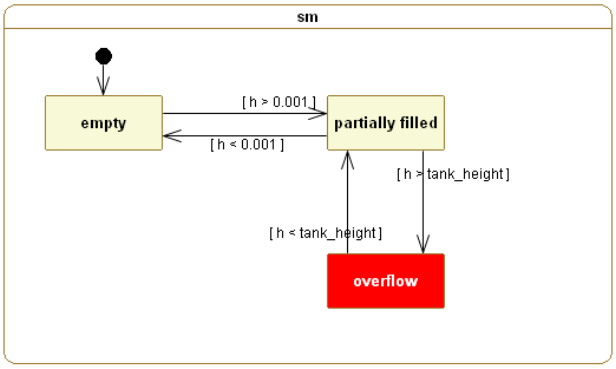

Figure 10. State Machine of the Tank

The next State Machine specifies the behavior of the controller. It shows that only if the controller is in the state "on" it will monitor or control the level of liquid in tank depending on the sensor values received.

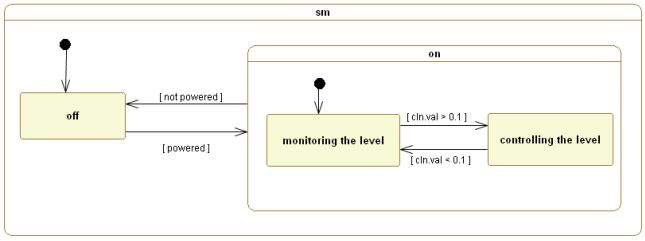

Figure 11. State Machine of the Controller

Any other behavior defined for a system can be defined as being dependent on a specific explicit state of the system. For example, the following shows how conditional equations are modeled including the dependence on the defined states. Depending on if the controller is in the state "controlling the level" it will activate (the equation is not visible on the diagram, it is: cOut.act = outCtr;) or deactivate (cOut.act = 0;) the actuator signal.

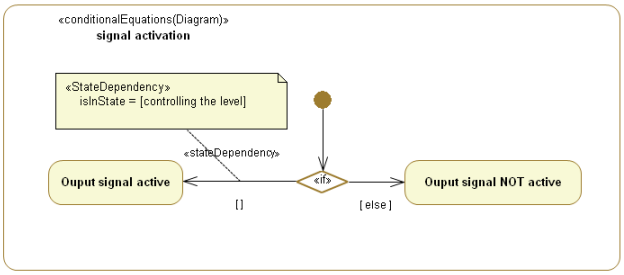

Figure 12. Example of state-dependent equations

The generation of Modelica code from StateCharts was already investigated previously, for example in [5]. Furthermore, [6] introduced the State Graph library for Modelica, which has similar power compared to StateCharts, although it has a slightly different graphical notation. ModelicaML takes a similar approach. In addition to the limitation listed above, the current version of the ModelicaML code generator does not support compound transitions (transition which cross state hierarchy borders), History, Fork/Joins, Entry/ExitPoints and ConnectionPointReference. The limitation and formal definition of the semantics for the State Machines and the Activity Diagrams (including time-consuming activities) are subject to the current ModelicaML research work.

7 Further Concepts (under investigation by ModelicaML)

Inspired by the SysML, ModelicaML reuses the concept of textual requirements within models. As in the SysML it is possible to include textual requirements into ModelicaML models and link requirements to model artifacts. This enables traceability between textual requirements and design artifacts, and supports impact analysis when requirements and/or the model change. Figure 13 illustrates how textual requirements appears graphically on diagrams in ModelicaML.

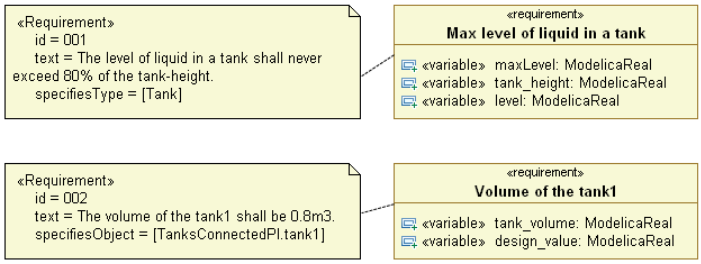

Figure 13. Example of textual requirements in ModelicaML

In contrast to SysML, requirement is defined in ModelicaML as a sub-class of the UML Class which can have behavior. It is possible to define properties and behavior (e.g. assertions) for requirements. In doing so it is possible to evaluate if a requirement is violated or not during system simulation. Our current research in this field aims at finding ways to formalize different types of requirements and to find a flexible way to associate requirements with design models. The following examples present some ideas.

Assume the following requirements to be imposed on the two tanks system:

Req. 001: The level of liquid in a tank shall never exceed 80% of the tank-height.

Req. 002: The volume of tank1 shall be 0.8 m^3.

The first requirement specifies a type: Tank in this case. In order to establish the traceability between the textual requirement and the design artifact the class Tank is referenced from the requirement inside the model using the requirement property "specifiesType". It implies that any instance of a tank must meet this requirement. In contrast, the second requirement is a

design requirement defining the required volume of tank 1. This requirement is imposed only on a particular instance of the type Tank. Therefore, the dot-notation in the requirement property "specifiesObject" is used to reference the respective instance. The "specifies…" - relations are descriptive only. They are not translated into Modelica code and do not impact the simulation.

In order to be able to evaluate these requirements during the system simulation requirements need to be formalized. In the following one possible way to do so is presented.

From the textual statement of the requirement 001 we can identify measurable properties such as: level (current level in a tank), maxLevel (80 % max. allowed level), tank_height (the height of a tank). Moreover, we can define a property indicating if the requirement is violated or not by evaluating: `level > maxLevel * tank_height`. Consider the following state machine specifying if the requirement 001 is violated or not. The second requirement is modeled in a similar way; it is not presented here.

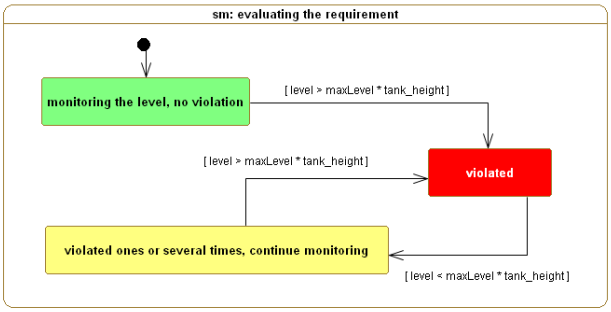

Figure 14: Example of requirements behavior

The modeled requirements can now be instantiated and their properties can be bound to the values within the corresponding design model (TanksConnectedPI in that case). In this example, the declarations for the r001_tank2 (Figure 14) are:

- `level = dm.tank1.h`
- `tank_height = dm.tank1.tank_height`

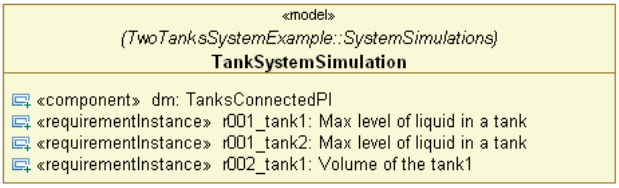

Figure 15: Instantiated design model and associated requirements

Note that requirement 001, which specifies the type Tank, is instantiated two times (because there are two tanks in the system).

Figure 16 shows the results of the evaluation (the tank_height is 0.6m in this example). The requirement 001 evaluated for the tank2 (r001_tank2) was violated two times during the simulation.

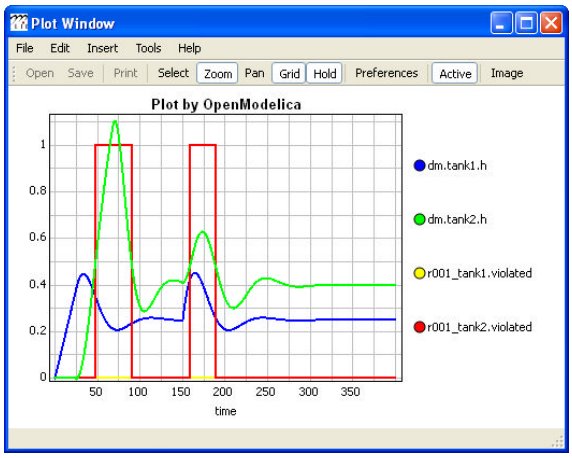

Figure 16: Example of requirements evaluation during system simulation

Similar to the concept of textual requirements, the modeller can define measures of effectiveness of models, which are used to record dedicated, measurable properties of system models during simulations and can compare them according to predefined metrics, for example, in order to select the best potential design alternative.

Our future ModelicaML research aims at developing a flexible association of requirements to multiple design alternatives in a way that requirement models can be instantiated automatically together with the associated design models in order to be evaluated during system simulation.

8 Modeling Support

Usually, when using a UML modeling tool, the model elements can be created either directly in the model browser (a tree-like representation of the classes, etc.) or using diagrams. In both cases the model data is stored in the model repository (see Figure 17).

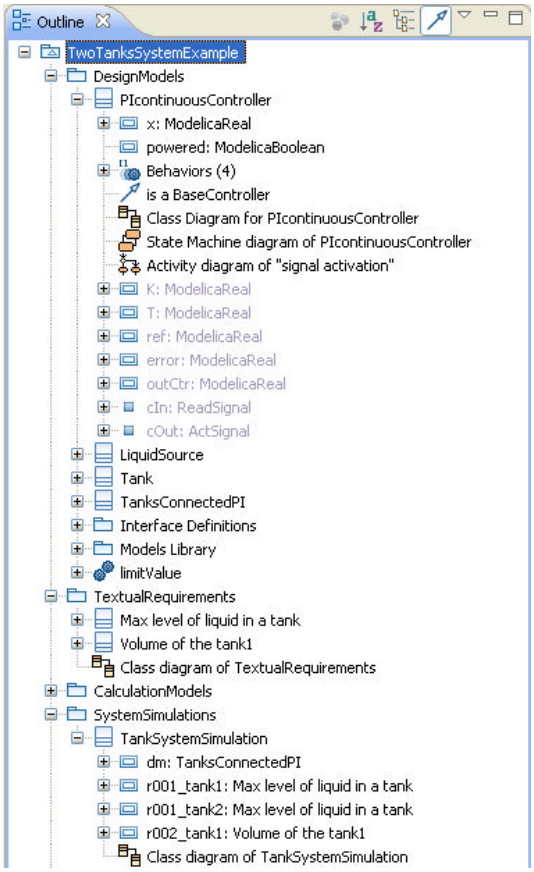

Figure 17: Example of a ModelicaML model browser

Diagrams only provide a view on a selected part of the model data. Diagrams can be used only for modeling (i.e., capturing the data), and might be deleted[3] after the data is captured. In some cases the modeler may decide to leave some diagrams for documentation or communication purposes. In this case, the modeler will need to select the data that should appear on dedicated diagrams (depending on which data can be displayed on a specific diagram type). An appropriate partitioning of the model data into different diagram and diagram types is essential in order to improve readability and to support the modeler by automatic generation and layout of diagrams. For example, the diagrams in figures Figure 3, Figure 4, Figure 5, Figure 13 or Figure 15 would not need to be modeled (and arranged visually). These can be generated from the model data.

This will prove rather difficult for the diagrams in Figure 2, Figure 10, Figure 11 or Figure 14. Those diagrams will need to be modeled (arranged visually) by the modeler. This is a good indicator to see if value is added by spending time on a diagram.

[3] Of course, any diagram can be recreated from the model data.

9 Model Validation and Code Generation

The ModelicaML code generator that generates Modelica code from the ModelicaML models is implemented using the Acceleo Eclipse Plug-In [16], which follows the MDA approach and the model-to-text recommendations of the OMG.

Presently, ModelicaML is implemented as a UML Profile that can be used in any (Eclipse-based) UML2 tool. This way the modeler needs to first create a UML element and then apply a stereotype, defined in the ModelicaML profile, in order to represent a specific concept or to introduce (or to specify) the semantics. The a*dvantage* of this approach is: it allows creating or reading ModelicaML models using any UML2 tool. *The disadvantage is*: the modeling tool GUI does not directly reflect the Modelica wording. The modeler needs to have a basic knowledge of the UML in order to know which stereotypes of the ModelicaML profile should be applied to which UML elements. Moreover, all limitations, constraints and possible inconsistencies will have to be checked and resolved before the Modelica code generation. Therefore, the ModelicaML code generator includes a validator that checks the model and informs the modeler about inconsistencies before the Modelica code is generated.

10 Simulation Support (Using OpenModelica Environment)

In addition to the convenient way of simulating a Modelica model from startTime to stopTime, in the frame of the ModelicaML research and implementation the OpenModelica Environment [12] was enhanced by interaction simulation capabilities (similar to the Interaction Library in Dymola [15] Modelica tool). It is possible to generate Modelica code directly from the ModelicaML models and to pass it to the OMC. A dedicated simulation GUI has been implemented providing the user with possibilities to interact with the Modelica model (i.e., to change parameters at runtime) and to observe the reaction of the system immediately on plots. Moreover, any additional GUI (with domain specific animations or widgets) can be implemented and connected to the simulation using the implemented OMC interactive simulation interface. This feature will support model debugging as well as the communicating and discussing of the modeled system behavior to and with any parties involved in the system development process.

11 Conclusion

This paper presents a step towards, and a proof of concept for, a unified executable system modeling language and environment using open-source UML modeling (Papyrus UML) and simulation (OpenModelica) tools.

One of our main future research activities in the field of ModelicaML will be dedicated to developing graphical notations for modeling any kind of equations or statements, as well as other constructs (e.g. type- or instance-modification) that are now captured using strings. This will avoid the refactoring of models and enable semantic analysis of the ModelicaML models.

In conclusion, UML/SysML and Modelica are complementary languages supported by two active communities. By integrating UML/SysML and Modelica into ModelicaML, we combine the very expressive, formal language for differential algebraic equations and discrete events of Modelica with the expressive UML/SysML graphical notation for requirements, structural decomposition, logical behavior, and corresponding cross-cutting constructs.

In addition, the two communities are expected to benefit from the exchange of multi-domain model libraries and the potential for improved and expanded commercial and open-source tool support.

12 Acknowledgements

This work has been supported by EADS Innovation Works, by Swedish Vinnova in the ITEA2 OPENPROD project and by the Swedish Research Council (VR).

References

[1] Modelica Association. Modelica: A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification Version 3.0, Sept 2007. www.modelica.org

[2] OMG. OMG Unified Modeling Language ™ (OMG UML). Superstructure Version 2.2, February 2009.

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

[4] OMG. OMG Systems Modeling Language (OMG SysML™), Version 1.1, November 2008.

[5] Ferreira J. A. and Estima de Oliveira J. P., Department of Mechanical Engineering, University of Aveiro, 3810 Aveiro (PORTUGAL), Department of Electronic Engineering, University of Aveiro, 3810 Aveiro (PORTUGAL), MODELLING HYBRID SYSTEMS USING STATECHARTS AND MODELICA, J. A.

[6] M. Otter, K.-E. Arz´en, I. Dressler. StateGraph-A Modelica Library for Hierarchical State Machines. DLR Oberpfaenhofen, Germany; Lund Institute of Technology, Sweden. Proceedings of the 4th International Modelica Conference, Hamburg. March 7-8, 200.

[7] Pop, A., and Akhvlediani, D., and Fritzson, P. Towards Unified Systems Modeling with the ModelicaML UML Profile. *International Workshop on Equation-Based Object-Oriented Languages and Tools. Berlin, Germany, Linköping University Electronic Press, 2007*

[8] Peak, R., McGinnis, L., Paredis, C. Integrating System Design with Simulation and Analysis Using SysML – Phase 1 Final Report. 2008

[9] Johnson, T. A. Integrating Models and Simulations of Continuous Dynamic System Behavior into SysML. M.S. Thesis, G.W. Wood-ruff School of Mechanical Engineering, Georgia Institute of Technology. Atlanta, GA. 2008

[10] Papyrus UML, www.papyrusuml.org

[11] Schamai W.. Modelica Modeling Language (ModelicaML) A UML Profile for Modelica, technical report 2009:5, EADS IW, Germany, Linkoping University, Sweden, 2009

[12] The OpenModelica Project www.ida.liu.se/labs/pelab/modelica/OpenModelica.html

[13] Object Management Group (OMG). www.omg.org

[14] Modelica Association. www.modelica.org

[15] Dymola (Dynamic Modeling Laboratory), Dynamism. www.dymola.com

[16] Acceleo, Eclipse Plug-In. www.acceleo.org/pages/home/en

Appendix: Modelica Example Code

```
connector ActSignal "Signal to actuator for setting
valve position"
  Real act;
end ActSignal;

connector ReadSignal "Reading fluid level"
  Real val(unit = "m");
end ReadSignal;

connector LiquidFlow  "Liquid flow at inlets or
outlets"
  Real lflow(unit = "m3/s");
end LiquidFlow;

partial model BaseController
  parameter Real K = 2 "Gain";
  parameter Real T(unit = "s") = 10 "Time constant";
  ReadSignal cIn "Input sensor level, connector";
  ActSignal  cOut "Control to actuator, connector";
  parameter Real ref "Reference level";
  Real error "Deviation from reference
level";
  Real outCtr "Output control signal";
equation
  error = ref - cIn.val;
  cOut.act = outCtr;
end BaseController;

function limitValue
  input  Real pMin;
  input  Real pMax;
  input  Real p;
  output Real pLim;
 algorithm
  pLim := if p>pMax then pMax
          else if p<pMin then pMin
          else p;
end limitValue;

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

model PIcontinuousController
  extends BaseController(K = 2, T = 10);
  Real  x  "State variable of continuous PI
controller";
equation
  der(x) = error/T;
  outCtr = K*(error + x);
end PIcontinuousController;

model Tank
  ReadSignal tSensor "Connector, sensor reading tank
level (m)";
  ActSignal tActuator "Connector, actuator controlling
input flow";
  LiquidFlow qIn "Connector, flow (m3/s) through input
valve";
  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; // Limits for
output valve flow
  Real h(start = 0.0, unit = "m") "Tank level";
 equation
  assert(minV>=0,"minV - minimum Valve level must be
>= 0 ");
  der(h) = (qIn.lflow - qOut.lflow)/area; // Mass
balance equation
  qOut.lflow  = limitValue(minV, maxV, -
    flowGain*tActuator.act);
  tSensor.val = h;
end Tank;

model TanksConnectedPI
  LiquidSource  source(flowLevel = 0.02);
  Tank          tank1(area = 1);
  Tank          tank2(area = 1.3);
  PIcontinuousController piContinuous1(ref = 0.25);
  PIcontinuousController piContinuous2(ref = 0.4);
 equation
  connect(source.qOut,tank1.qIn);
  connect(tank1.tActuator,piContinuous1.cOut);
  connect(tank1.tSensor,piContinuous1.cIn);
  connect(tank1.qOut,tank2.qIn);
  connect(tank2.tActuator,piContinuous2.cOut);
  connect(tank2.tSensor,piContinuous2.cIn);
end TanksConnectedPI;
```