Bachelor Thesis

Parham Vasaiely

Interactive Simulation of SysML Models using Modelica

EADS Deutschland GmbH

Faculty of Engineering and Computer Science
Department Computer Science

Fakultät Techchnik und Informatik
Studiendepartment Informatik

Parham Vasaiely

Interactive Simulation of SysML Models using Modelica

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Olaf Zukunft
Zweitgutachterin : Prof. Dr. Bettina Buth

Abgegeben am: 24.08.2009

Parham Vasaiely

Title of the thesis

Interactive Simulation of SysML Models using Modelica

Keywords

UML, SysML, Modelica, Simulation, Interactive, System, Model based Engineering, Systems Engineering

Abstract

The International Council on Systems Engineering (INCOSE) identified Model-Based Systems Engineering as a key driver for effective and efficient system development in the future. System simulation using models is widely used for analysis, communication or training purposes. This thesis presents an approach for user-interactive simulation of system models which are created using the graphical Systems Modelling Language (SysML) and translated into executable Modelica models. A software prototype based on the OpenModelica environment will be developed and demonstrates the application on a concrete example.

Parham Vasaiely

Thema der Bachelorarbeit

Interaktive Simulation von SysML Modellen unter Verwendung von Modelica

Stichworte

UML, SysML, Modelica, Simulation, Interaktive, System, Modell basierte Entwicklung, System Entwicklung

Kurzzusammenfassung

International Council on Systems Engineering (INCOSE), erkannte die Modellbasierte System Entwicklung als eine effektive und effiziente Schlüsseltechnik für die zukünftige Entwicklungen von Systemen. Die Simulation von Systemen wird meist zu Analyse-, Kommunikations- oder Einweisungs- Zwecken verwendet. In dieser Arbeit wird ein Ansatz zur interaktiven Simulation von System Modellen, welche unter Verwendung der graphischen Systemmodellierungssprache SysML erzeugt und in ausführbare Modelica Modelle übersetzt wurden, präsentiert. Ein Software Prototyp, welches auf der OpenModelica Umgebung basiert, wird entwickelt und eingesetzt um ein konkretes Beispiel der Anwendung zu demonstrieren.

Acknowledgments

First of all I must thank my family because of they support and love. My mom, Soudabeh, has always believed in me and her positivity is my moving spirit. Through the good times and the bad times, she has been my most important source of support.

I must thank EADS Innovation Works and Wladimir Schamai, my advisor at EADS, he is a very competent engineer and it was a pleasure to work with him.

Also, I appreciatively acknowledge the support of Lawrence Harris, from the Technical English Language Services (http://www.tels.de), for supporting my technical English spelling and his effort to correct my thesis.

Finally, I am grateful for the academic software licenses provided by Microsoft, IBM, Object Refinery Limited.

Table of Contents

I. List of Figures ... 7
II. List of Tables ... 9
III. Glossary ... 10
1. Introduction ... 11
1.1. Background ... 11
1.2. Objective of the thesis ... 12
1.3. Thesis Structure ... 13
2. State of the Art ... 14
2.1. Modelica – An Overview ... 14
2.1.1. The Modelica application area ... 14
2.1.2. Modelling and Simulation Tools for Modelica ... 14
2.1.2.1. Dymola ... 15
2.1.2.2. MathModelica ... 16
2.1.2.3. OpenModelica ... 16
2.2. The Systems Modelling Language ... 19
3. Demonstration System ... 21
3.1. The Two Tanks System ... 21
4. Translation of a SysML model to a Modelica model ... 23
4.1. Mapping of SysML to Modelica ... 23
4.1.1. Model transformation ... 23
4.1.2. Additional Stereotypes ... 26
4.1.3. SysML Parametric to Modelica Equation ... 28
4.2. TanksConnectedPI System in SysML ... 29
4.2.1. System structure with SysML Block Definition Diagram and SysML Internal Block- Diagram ... 33
4.2.2. Block Definition Diagrams of the constraint blocks ... 36
4.2.3. Parametric Diagrams of the parametrics structure ... 37
5. Interactive Simulation Runtime ... 41
5.1. OpenModelica Interactive ... 42
5.1.1. The OpenModelica Subsystem ... 43
5.1.1.1. OpenModelica Subsystem Service Interface ... 44
5.1.2. The OpenModelica Interactive Subsystem ... 44
5.1.2.1. OMI::Control ... 44

5.1.2.2. OMI::ResultManager 45
5.1.2.3. OMI::Calculation 48
5.1.2.4. OMI::Transfer 48
5.1.3. Communication Interface (Architecture) 49
5.1.3.1. Communication 49
5.1.3.2. Operation Messages 50
5.1.4. OpenModelica Interactive Structure and Behaviour 52
5.1.5. Testing of the OpenModelica Interactive simulation runtime 56
5.1.5.1. Back to Back Tests 56
6. Interactive Graphical User Interface 58
6.1. Simulation configuration 58
6.2. Simulation Environment 59
7. Conclusions and Future Work 62
7.1. Conclusions 62
7.2. Future Work 62
IV. References 64
V. Appendix 67

I. List of Figures

Figure 2-1 Dymola system modelling (left) and plot of simulation results (right) 15
Figure 2-2 MathModelica system modelling (left) and plot of simulation results (right) 16
Figure 2-3 OpenModelica (1.4.5) System overview architecture....................................... 17
Figure 2-4 OMC generated executable program to simulate a Modelica model................ 18
Figure 2-5 OM Simulation Runtime main components and their dependencies................ 18
Figure 2-6 Relationship between SysML and UML 19
Figure 2-7 SysML Diagram Types.. 20
Figure 3-1 Two tanks with proportional–integral continuous controllers connected together 21
Figure 3-2 TanksConnectedPI structure diagram... 22
Figure 3-3 Plot of simulation results from the levels of tank1 and tank2 22
Figure 4-1 The created Stereotypes in Rhapsody... 27
Figure 4-2 TwoTanks Package Structure 29
Figure 4-3 Tank Block 29
Figure 4-4 LiquidSource Block 30
Figure 4-5 BaseController Block 30
Figure 4-6 PIcontinuousController Block 31
Figure 4-7 TanksConnectedPI Block.. 31
Figure 4-8 ReadSignal FlowSpecification.. 32
Figure 4-9 ActSignal FlowSpecification 32
Figure 4-10 LiquidFlow FlowSpecification 32
Figure 4-11 BBD TanksConnectedPI 33
Figure 4-12 Inheritance between BaseController and PIcontinuousController 33
Figure 4-13 IBD TanksConnectedPI 34
Figure 4-14 BDD Tank Constraints 36
Figure 4-15 BDD BaseController Constraints.. 36
Figure 4-16 BDD PIcontinuousController Constraints 36
Figure 4-17 BDD LiquidSource Constraints 37
Figure 4-18 PAR Tank... 37
Figure 4-19 PAR BaseController and PIcontinuousController... 39
Figure 4-20 PAR Outgoing flow level of the LiquidSource... 40
Figure 5-1 OpenModelica Interactive System Architecture Overview 43

Figure 5-2 Pseudo code of push and pull in SRDF 48
Figure 5-3 UML-Structure OM and OMI with some attributes and methods...................... 52
Figure 5-4 UML-Seq Handshake, model initialization and set Transfer filter mask 53
Figure 5-5 UML-Seq Simulation start 53
Figure 5-6 UML-Seq Calculation phase 54
Figure 5-7 UML-Seq Transfer to client phase 54
Figure 5-8 UML-Seq Change Value of a parameters 55
Figure 5-9 Plot of Simulation Results Tank1.h and Source.qOut.lflow 56
Figure 6-1 Simulation Configuration Tool 58
Figure 6-2 Simulation control center 59
Figure 6-3 Selection of properties to display on plot 60
Figure 6-4 New plot to display tank1.h and tank2.h 60
Figure 6-5 Live plot of tank1.h and tank2.h 61

II. List of Tables

Table 4-1 SysML Package à Modelica Package ... 24
Table 4-2 SysML Block à Modelica Block ... 24
Table 4-3 SysML Attribute à Modelica Variable ... 24
Table 4-4 SysML FlowSpecification à Modelica Connector ... 24
Table 4-5 Atomic Flow Port Node à Instance of connector ... 24
Table 4-6 SysML Connector à Modelica Connection ... 25
Table 4-7 SysML Flow (FlowDirection) à Modelica Causality of connector instance ... 25
Table 4-8 SysML Inheritance (Gen/Spec) à Modelica extends ... 25
Table 4-9 SysML Datatype Double à Modelica Datatype Real ... 25
Table 4-10 SysML Stereotype <<variable>> for Modelica variability and unit ... 26
Table 4-11 SysML Stereotype <<extendsRelation>> for Modelica modification of inherit variable values ... 26
Table 4-12 SysML Stereotype <<abstract>> for Modelica partial ... 27
Table 4-13 SysML Stereotype <<composite>> for Modelica instance modification ... 27
Table 4-14 SysML Parametric elements ... 28
Table 5-1 OMI server and client components ... 50
Table 5-2 GUI server and client components ... 50
Table 5-3 Available messages from a GUI to OMI (Request-Reply) ... 51
Table 5-4 Available messages from OMI::Control to GUI ... 51
Table 5-5 Available messages from OMI::Transfer to GUI ... 51
Table 5-6 source.flowLevel values for a Back to Back Test ... 56
Table 5-7 Results of the Back to Back Test ... 57

III. Glossary

BBD	Block Definition Diagram
DASSL	Differential/Algebraic System Solver
EADS	European Aeronautic Defence and Space
GUI	Graphical User Interface
IBD	Internal Block Diagram
INCOSE	International Council on Systems Engineering
IS	Interactive Simulation
MBSE	Model-Based Systems Engineering
OM	OpenModelica
OMC	OpenModelica Compiler
OMG	Object Management Group
OMI	OpenModelica Interactive
PAR	Parametric Diagram
PI	proportional–integral
SysML	Systems Modelling Language
UP	Unified Software Development Process
XML	Extensible Markup Language

1. Introduction

1.1. Background

The International Council on Systems Engineering (INCOSE) [19] identified Model-Based Systems Engineering (MBSE) [11] as the key driver for effective and efficient system development in the future. One of the key MBSE drivers identified was the need for a standardized notation for description of system requirements or design at any level of abstraction. However, in the operational field was quickly realised that a comprehensive simulation of systems (e.g. for the purpose of system analysis, validation and verification) is the main beneficial part of an MBSE approach.
In the development of complex systems multiple engineering disciplines are involved each using its own formalisms and tools to develop their own parts of the system.
Examples of complex systems are Robotics, Automotive, Aircraft and Biomechanics.

OMG Systems Modelling Language (SysML) [20] was developed in order to support effective communication among the parties involved by means of a standardized graphical notation. Since SysML does not include an action language it is up to the tool vendor to select an appropriate one and to make SysML models executable. For example, the COTS SysML tool Rhapsody (IBM) provides code generation from SysML models and enables interactive simulation of models. In turn, Modelica [21] is a well-defined object oriented modelling language which is dedicated to the simulation of physical systems.

For comparison MATLAB/Simulink [9] is widely used in industry for modelling and simulation of systems.
However, there are essential differences when compared with SysML/Modelica:

- MATLAB/Simulink does not have a standardized graphical notation whereas SysML has a standardized general purpose graphical notation for modelling different views of the system definition.
- MATLAB/Simulink is based on the signal-oriented paradigm (also referred to as block-oriented), which always forces a causal dependence between inputs and outputs of a block when solving equation systems. This is different to Modelica which follows the equation-based modelling paradigm and enables acausal modelling. This approach is more suitable for physical systems modelling.

- MATLAB/Simulink does not support inheritance-concepts for classification of components in order to enable their reuse. Both, SysML and Modelica provide such capabilities.

Putting together SysML and Modelica gives a powerful combination for modelling and simulation of complex systems at any stage of system development.

1.2. Objective of the thesis

Current activities inside the OMG SysML address integration of SysML with Modelica in order to combine the graphical modelling capability of SysML with the simulation power of Modelica. This research contributes to this effort as well as to a larger initiative established between the EADS Innovation Works (Hamburg) in collaboration with the Modelica developers at the Linkoping University in Sweden.

This research project is aimed at integrating SysML and Modelica in order to enable system modelling and simulation respectively. In the early development stages system engineers do not need a deep, detailed physical or mathematical modelling and simulation but rather the capability to express system structure and behaviour. In terms of behaviour state-charts (in different versions) are widely used for time-discrete and reactive behaviour modelling and simulation. SysML introduces the parametric concepts for constrained based behaviour modelling. The constraints can be expressed using mathematical equations and thus facilitate time-continuous system behaviour simulation. However, SysML does not define an execution language for any kind of behaviour and leave this choice and the definition of the detailed execution semantics to the implementers and tool vendors.

This work is a step towards the application of Model-Based Systems Engineering paradigm. It combines the descriptive power of SysML with the simulation power of Modelica and enables creation of executable system models for different purposes and at different levels of abstraction.

The main objective of this thesis is to enable a user-interactive of simulation system behaviour that incorporates time-continues, time-discrete or event-based behaviour.

This challenge includes two research problems:

- How to apply the execution semantics of Modelica to SysML models in order to make them executable?
- When using Modelica the system models are typically simulated from a defined start time to a stop time. It is not possible to interact with the model when the simulation is running. The question is: What are the necessary extensions of Modelica simulation environments in order to enable user interactive simulation?

In particular the latest is the focus of this thesis. The main purpose of such simulation is it to enable the interaction with the system model during system simulation in order to support system-related analysis, communication or training.

1.3. Thesis Structure

The thesis work contains the following chapters:

Chapter 2. “State of the Art” is a short introduction to the Modelica language, Modelica tools and SysML.
Chapter 3. “Demonstration System” describes the used demonstration system and its components.
Chapter 4. “Translation of a SysML model to a Modelica model” discusses a possible approach to map SysML to Modelica and shows a full application of this approach by translating the demonstration model from SysML to executable Modelica code.
Chapter 5. “Interactive Simulation Runtime” discusses implementation details of the interactive simulation runtime based on OpenModelica.
Chapter 6. “Interactive Graphical User Interface” presents a short description of a developed interactive simulation environment to demonstrate the whole application.
Chapter 7. “Conclusions and Future Work” summarizes the thesis work and provides several future work directions.

2. State of the Art

2.1. Modelica – An Overview

Bellow is a short introduction to the Modelica language, its features and some application area examples. In addition two commercial and one open source Modelica modelling and simulation environments will be introduced.

Modelica is an object oriented programming language. It is based on the declarative programming paradigm which expresses the logic of a computation by describing what the application should accomplish without describing its control flow. This minimizes side effects which are absolutely unrequested during a simulation phase.
Models in Modelica are described mathematically using differential, algebraic and discrete equations. Modelica tools will have enough information to solve every particular variable automatically, at assessed the given equations. Therefore the Modelica system and component models are perfectly suited to be simulated by a simulation environment.

By the “Simulation in Europe Basic Research Working Group” the endeavours for the Modelica language started in 1996 within ESPRIT Project. Many well-known object-oriented modelling designers worked together to finish the language specification in 1999.
The Modelica Association was founded for further development and promotion of Modelica which is an open source language.

2.1.1. The Modelica application area

The Modelica language can be used for modelling large, complex and heterogeneous physical systems, for example automotive or aerospace applications involving mechanical, electrical, hydraulic and control subsystems or process oriented applications and generation.

2.1.2. Modelling and Simulation Tools for Modelica

The Modelica language is textual based, so a modelling environment is needed to offer a component based rather than visual component based modelling of systems. Also the simulation part of the models needs a simulation environment.

There are several modelling and simulation environments on the market, which offers a component based modelling and the simulation of the Modelica model (Components from the standard Modelica library or especially constructed components [22]).

Needs for the modelling and simulation environments:

- To conveniently define a Modelica model with a graphical user interface (composition diagram/schematic editor) such that the result of the graphical editing is a (internal) textual description of the model in the Modelica format.
- To translate the defined Modelica model into a form which can be efficiently simulated in an appropriate simulation environment. This requires sophisticated symbolic transformation techniques.
- To simulate the translated model using a standard numerical integration methods to visualize the result.

Here are some popular ones.

2.1.2.1. Dymola

The Dynamic Modeling Laboratory, Dymola [23], is a powerful Modelica modelling and simulation environment. Besides the graphical modelling capabilities it is possible to simulate the dynamic behaviour and complex interactions among systems from many engineering domains. The Dymola environment is completely open so users can easily introduce components or modify existing components to match the user's own unique requirements. Dymola is also compatible with many other tools so existing models from other tools can be used, for example it contains an interface to MATLAB and Simulink, and has CAD file import functionality. It has a powerful Modelica translator which is able to work on models with a huge number of equations (> 100,000), provided the complexity of these equations is reasonable. Dymola is probably the most powerful Tool which uses the Modelica language.

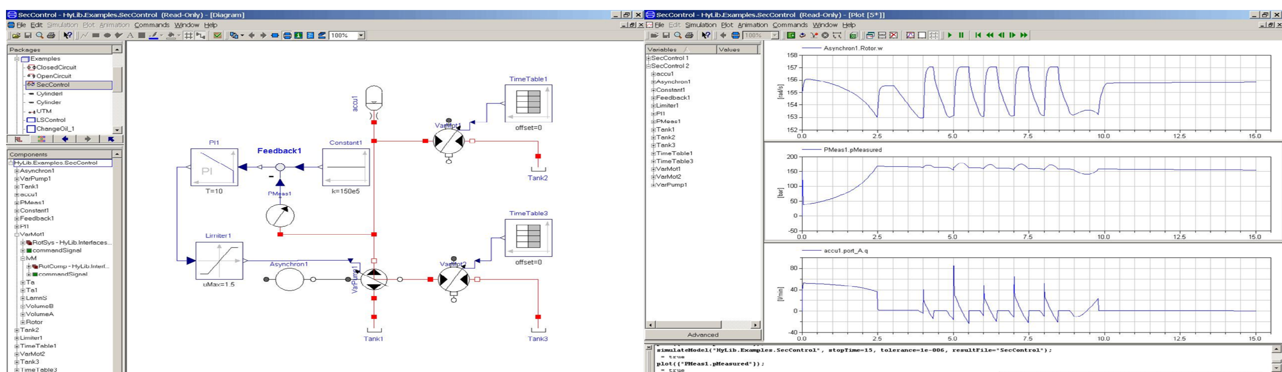

Figure 2-1 Dymola system modelling (left) and plot of simulation results (right)

2.1.2.2. MathModelica

MathModelica [25] is a modelling and simulation environment developed by MathCore Engineering AB. It consists of three major parts – a Modelica Editor, a Notebook and Simulation centre.

There are two major versions of MathModelica: Lite and System Designer (Professional).

- MathModelica Lite is the most basic modelling environment in the MathModelica family and it is free for academic and personal use. Unlike the other editions the Lite version uses the Modelica open source compiler from OpenModelica [24]. It provides a basic graphical modelling environment to conveniently define a Modelica model with a graphical user interface using the standard Modelica library components. The code editor provides a textual representation of the graphical model as Modelica code.
- MathModelica System Designer (Professional) has more modelling elements, including the standard Modelica library components and has the capability to simulate the model and plot its results. A model can be further documented in Mathematica Notebook. The MathModelica Notebook can also be used for simulation scripting and model analysis.

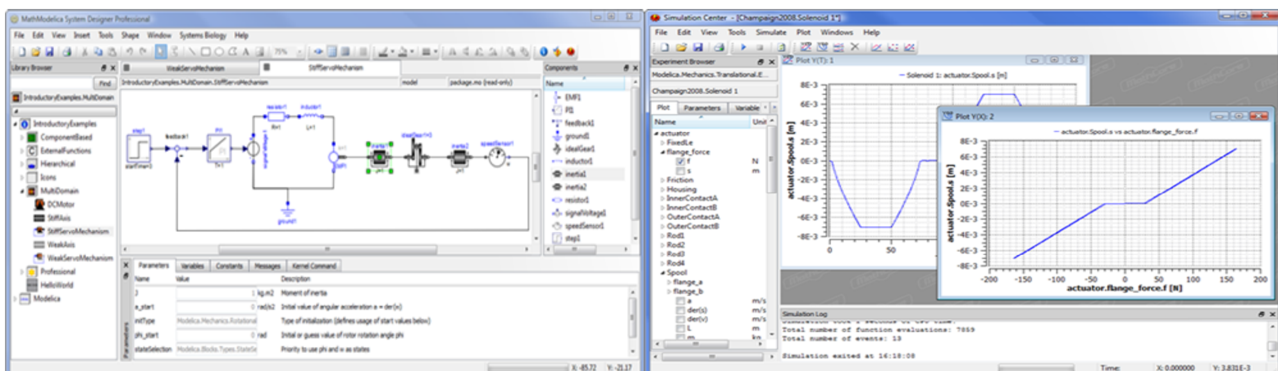

Figure 2-2 MathModelica system modelling (left) and plot of simulation results (right)

2.1.2.3. OpenModelica

OpenModelica is an open source Modelica environment developed and supported by Linköping University [27] and the Open Source Modelica Consortium (OSMC) [26].

The OpenModelica environment consists of several interconnected subsystems. The goal of the project is to create a complete modelling, compilation and simulation environment based on free software distributed in source code and executable form which is intended for use in research, teaching, and industry [17].

The OpenModelica environment is a collection of tools, OpenModelica Tools, to create a complete Modelica model. After instantiating the models can be simulated and the results plotted as a chart. A full tutorial is available based on the Modelica book by Peter Fritzson [2] which introduces the Modelica language, and an Eclipse plug-in (MDT) supports professionals while creating Modelica models. For more information on components please refer to the OpenModelica website [24] or “OpenModelica System Structure” [16].

OpenModelica Tools

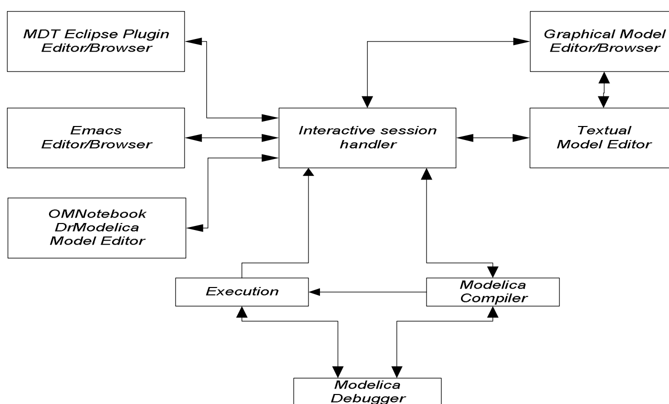

Figure 2-3 OpenModelica (1.4.5) System overview architecture

- Interactive session handler (OMShell) parses and interprets commands. Modelica expressions sent to it by other components for evaluation, simulation, plotting, etc.
- OpenModelica Compiler (OMC) translates Modelica to C code. OMC also builds simulation executables which are linked with selected ODE and DAE solvers.
- An execution and run-time module executes compiled binary code as well as simulation code from equation based models, linked with numerical solvers.
- Emacs textual model editor/browser is a model editor based on Gnu Emacs. Besides an editor, browsing of Modelica file hierarchy is possible.
- Eclipse Plug-in editor/browser provides class and library hierarchy browsing, syntax highlighting and editing capabilities.
- OMNotebook model editor is similar to Mathematica Notebook editor with basic functionality which help document and perform simulation.
- Graphical model editor/browser represents the MathModelica Lite product provided by MathCore without cost for academic usage. It allows graphical model composition, Modelica library browsing, etc.
- Modelica debugger is a conventional full-feature debugger integrated in Eclipse for displaying the source code. Stepping, breakpoint setting/unsetting are supported.

OpenModelica Simulation Runtime

As mentioned above after creating and instantiating a Modelica model it is possible to simulate the model with OpenModelica.

After calling the “simulation(…)” or “buildModel(…)” operation from the interactive session handler, an executable, standalone C/C++ program is generate from the internal simulation runtime code and the generated C/C++ model code by the OMC (in this case model.cpp).

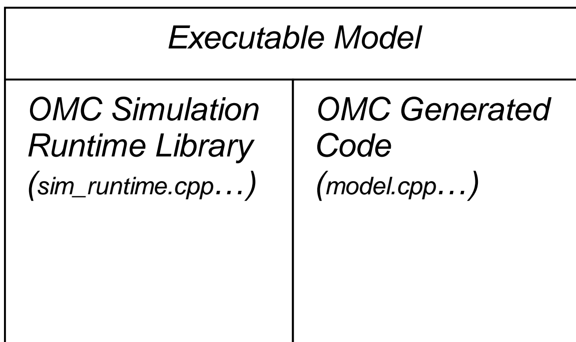

Figure 2-4 OMC generated executable program to simulate a Modelica model

The simulation runtime is insufficiently documented. Based on the C/C++ source code the following main components and their behaviour are identified:

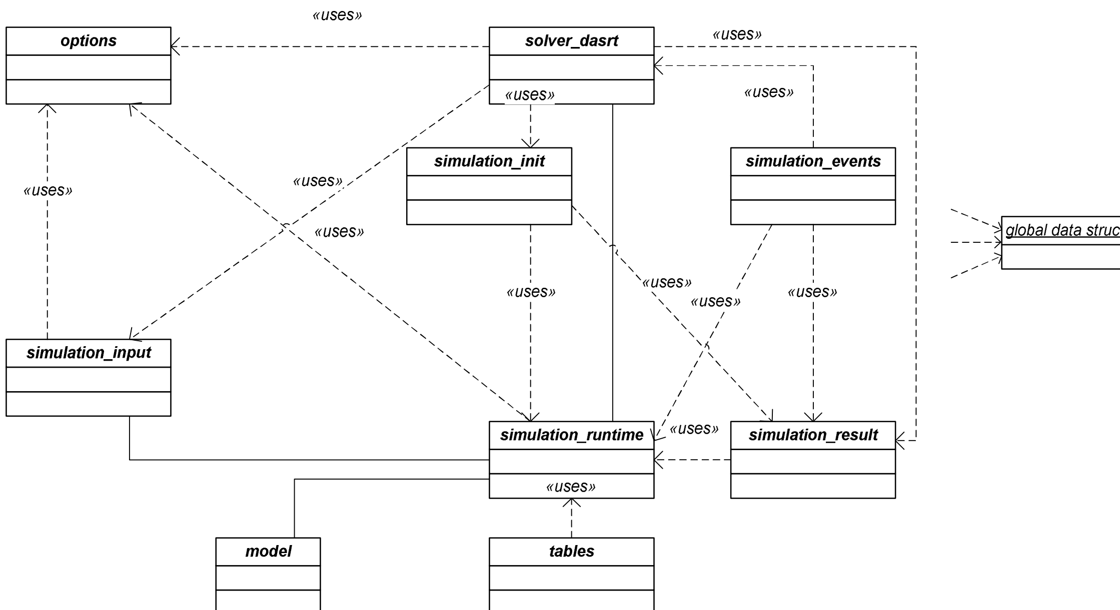

Figure 2-5 OM Simulation Runtime main components and their dependencies

- global data struct: Contains all model information, simulation options and simulation data. It represents the model at any time of the simulation. Nearly all components

use the global data structure to store and read information or data concerning the simulation and the model. This structure is part of the “simulation_runtime”.

- simulation_runtime: Includes the main function and call to the “solver_dasrt”. The “simulation_runtime.cpp” uses the generated “model.cpp” to initialize the global data structure.
- solver_dasrt: Wrapper for a “Differential/Algebraic System Solver” (DASSL). The DASSL is a free-for-use and open source solver [10] [18]. It solves all mathematically equations from the model. The simulation runtime also contains a simple Euler solver but it has not been implemented yet.

The OpenModelica simulation is not in real-time and accordingly not user interactive. It simulates the model between a specified time interval (start and stop time) as fast as the computer power will allow. The simulation runtime stores the simulation results in a “model_res.plt”. Afterwards a standard GUI plots the results into a chart.

For more information please visit the Modelica Association website [12] or
“Principles of Object-Oriented Modeling and Simulation with Modelica 2.1” by
Peter Fritzson [2].

2.2. The Systems Modelling Language

The Systems Modeling Language (SysML) [1] is a general-purpose graphical modeling language for the Systems-Engineering domain. It is used to specifying, analyzing, designing, and verifying complex systems. The language provides graphical representations with a semantic foundation for modeling system requirements, behavior, structure, and parametric, which is used to integrate with other engineering analysis models.

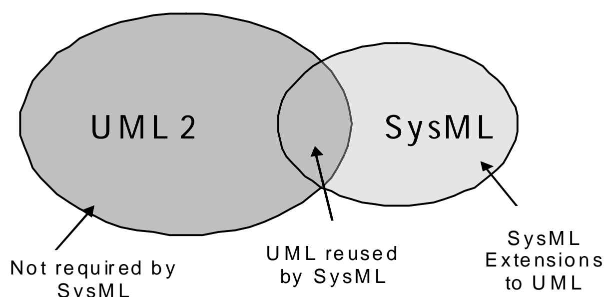

Figure 2-6 Relationship between SysML and UML

SysML represents a subset of UML 2 with extensions needed to satisfy the requirements of the UML for Systems Engineering.

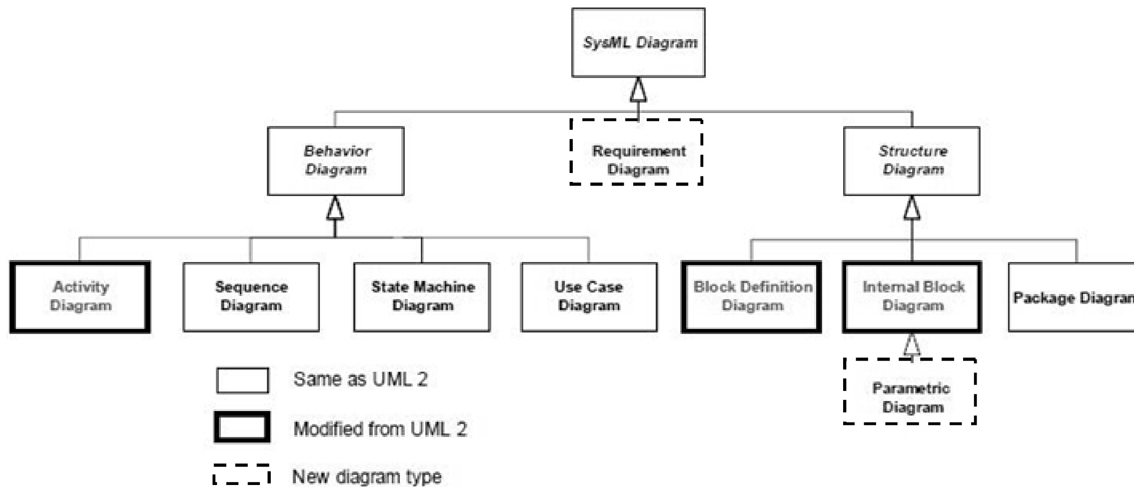

Figure 2-7 SysML Diagram Types

The taxonomy of SysML diagrams is presented in Figure 2-7 SysML Diagram Types.

The following are the major extensions of SysML Diagrams compared to UML Diagrams:

- The *Requirements diagram* supports requirements presentation in tabular or in graphical notation, allows composition of requirements and supports traceability, verification and satisfaction of requirements by other system elements.
- The *Block diagram* extends the Composite Structure diagram of UML 2. This diagram is to capture system components, their parts and connections between parts. Connections are handled by means of ports which may contain data flows.
- T*he Parametric diagram* helps perform engineering analysis such as *performance* analysis. Parametric diagram contains constraint elements, which define mathematical equation, linked to properties of model elements.
- *Activity diagrams* show system behaviour as data and control flows. Activity diagram is similar to Enhanced Functional Flow Block diagram (EFFBDs), which is already widely used by system engineers. Activity decomposition is supported by SysML.

For more information about SysML see the OMG SysML website [20] or
“A Practical Guide to SysML” by Sanford Friedenthal, Alan Moore and Rick Steiner [3].

3. Demonstration System

In the following a system is described that will be used throughout this thesis as an example system.

This example of a system is selected based on the following criteria:

- The example system should not be too complex. It should be understandable by readers without requiring specific technical background.
- The demonstration system should represent a natural physical problem which is not domain specific.
- The example shall address basic concepts of the Modelica and SysML languages (such as object-orientation, component-based approach and time-continuous behaviour modelling).
- The demonstration system will be used as proof of concepts throughout this thesis.

3.1. *The Two Tanks System*

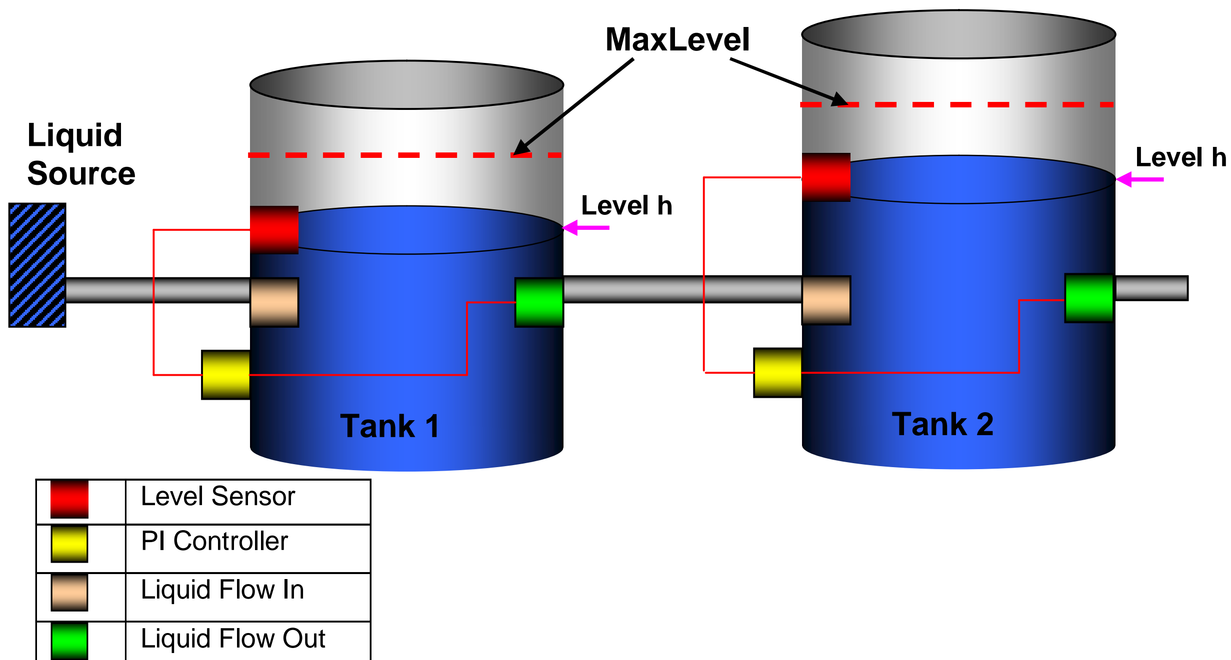

Figure 3-1 Two tanks with proportional–integral continuous controllers connected together

The system depicted in Figure 3-1 is based on a demonstration model which is given in the Modelica book by Peter Fritzson [[2], Page 386]. It represents two tanks connected together, and a liquid source which fills the first tank with liquid. Each tank has a

continuous proportional–integral (PI) controller connected to it, which regulates the level of liquid contained in the tanks to a reference level. While the liquid source fills the first tank with liquid the PI continuous controller regulates the outflow from the tank depending on its actual level. Liquid from the first tank flows into the second tank, which the PI continuous controller also tries to regulate. This is a natural and non domain specific physical problem. The system is called “TanksConnectedPI”.

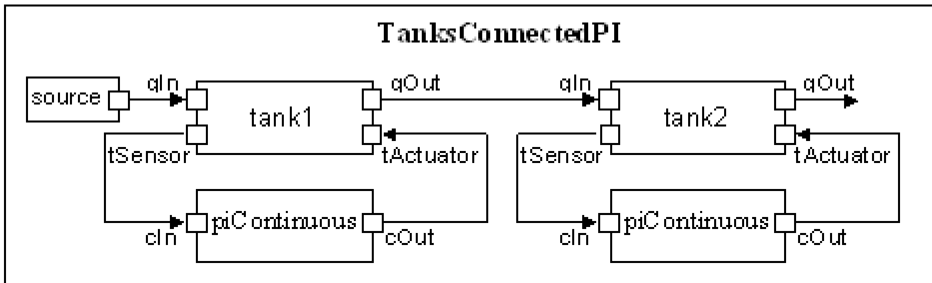

Figure 3-2 TanksConnectedPI structure diagram

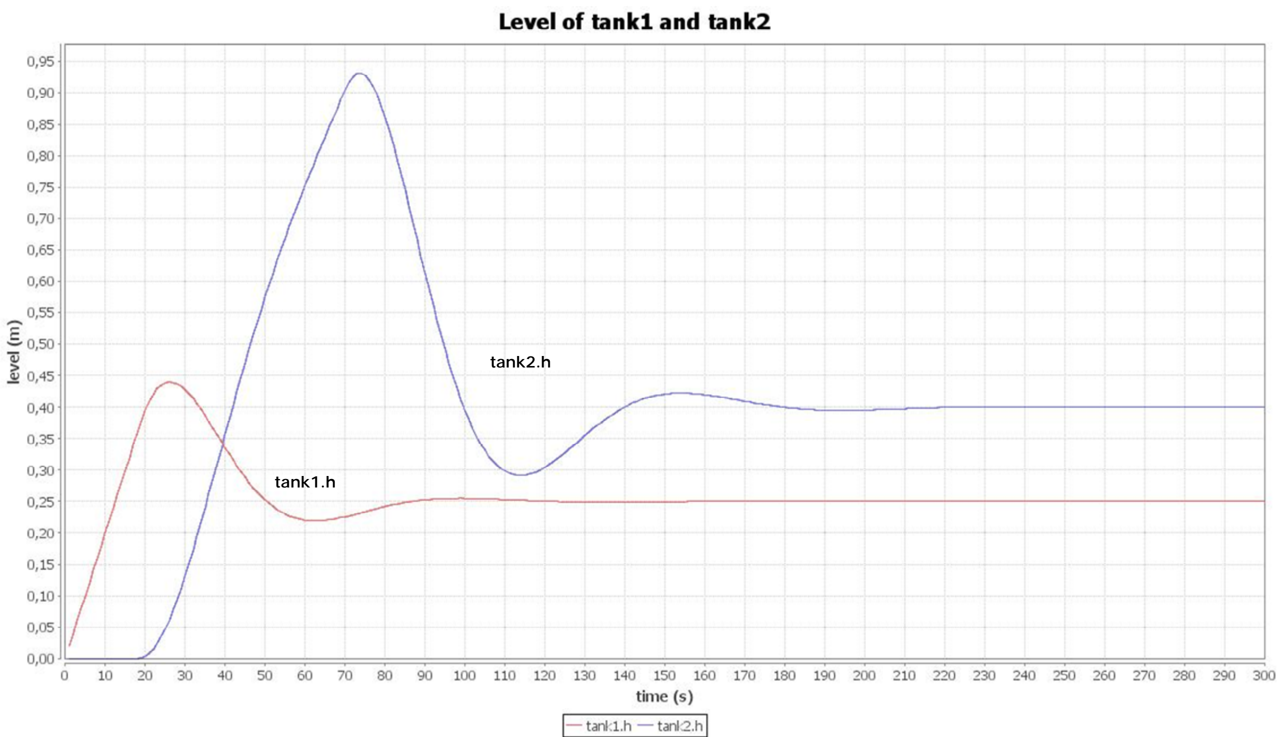

Figure 3-3 Plot of simulation results from the levels of tank1 and tank2

The graphs depicted in Figure 3-3 display the levels of tank1 and tank2 from 0s - 300s. The PI continuous controllers of both tanks try to get the levels to their reference levels (tank1 reference = 0.25, tank2 reference = 0.4). In this example the levels are regulated after 230s.

4. Translation of a SysML model to a Modelica model

As mentioned above the SysML standard does not define any action language, so that by default the created model is not executable. In order to enable the simulation of such models they need to be translated into an executable language, in this case Modelica. In order to do so the concepts from SysML and Modelica needs to be mapped to each other. The following is an approach for mapping and the resultant translation of the "TanksConnectedPI" model elements based on the full available Modelica code [Appendix A].

4.1. Mapping of SysML to Modelica

Modelica and SysML, like UML, follow the object-oriented paradigm. The resulting language structure is similar. For example, the main structural unit in SysML is Block (a sub-type of the UML Class) which corresponds to the Modelica Class in object-oriented sense. However, there are concepts that are different and have no correspondence between the two languages. In order to enable capturing of contents that are not present in SysML its' extension mechanism (profiles) is used. Profiles allow extension of the UML/SysML meta-model by means of stereotypes.
The following sections present the basic mapping between the SysML and Modelica as well as additional stereotypes that are defined in order to enable the capturing of Modelica specific concepts.

4.1.1. Model transformation

Well defined mapping is the most important part of a translation from one language into another. The following tables list a selected subset of SysML elements which are used for modelling the "TanksConnectedPI" demonstration model in SysML. A SysML element and its corresponding Modelica element are depicted in the left and centre columns respectively. The right column contains the commonality between the SysML and Modelica language elements.

SysML (Rhapsody)	Modelica	Commonality
Package	package	The packages partition the model elements into logical groupings.
The package is the basic unit of partitioning. The packages partition the model elements into logical groupings that minimize circular dependencies among them.	Packages in Modelica are used for logical groupings. Packages may contain definitions of constants, functions, and sub packages.	

Table 4-1 SysML Package à Modelica Package

SysML (Rhapsody)	Modelica	Commonality
Block	block	Describes a component. These may include both structural and behavioural features, containing all its parts and properties.
Blocks are modular units of system description. Each block defines a collection of features to describe a system or other element of interest.	In Modelica everything is a class. The basic class concept is “model”. “block” has the same properties as “model” but with some restrictions. The connector instances must have a specified direction.	

Table 4-2 SysML Block à Modelica Block

SysML (Rhapsody)	Modelica	Commonality
Attribute	Variable	Property which contains data and has a specified data type.
Property of a block which contains data. It could be from a pre defined or a user defined data type.	Property of a class which contains data and is from a pre defined data type. It has variability.	

Table 4-3 SysML Attribute à Modelica Variable

SysML (Rhapsody)	Modelica	Commonality
Flow Specification	connector	Used to connect components to each other and describing the flow data.
A flow Specification defines a set of input and or/output flows for a non composite flow port.	A class with restrictions which could be used to connect components to each other.	

Table 4-4 SysML FlowSpecification à Modelica Connector

SysML (Rhapsody)	Modelica	Commonality
Atomic Flow Port Node	Instance of connector	Specifies that a component has an interaction point.
An atomic flow port describes interaction point where an item can flow into or out of a block, or both, as indicates by the direction of the arrow in the Atomic Flow Port Node.	Instance of a connector is part of a component and is used to describe interaction. Its direction has to be defined in the owner class.	

Table 4-5 Atomic Flow Port Node à Instance of connector

SysML (Rhapsody)	Modelica	Commonality
Connector flow(x,y) (between flowPorts)	Connection equation connect(x,y)	Specifies interaction between elements.
A connector is used to bind two parts (or ports) and provides the opportunity for those to interact, although the connector says nothing about the nature of the interaction.	The connection equation specifies an interaction between connectors of different components.	

Table 4-6 SysML Connector à Modelica Connection

SysML (Rhapsody)	Modelica	Commonality
Flow (FlowDirection)	Causality of connector instance	Specifies the flow direction between two connected components. Note: Bidirectional is not possible.
Flows specify the exchange of information between system elements. They allow you to describe the flow of data and commands within a system at an early stage, before committing to a specific design. The direction describes the flow direction.	When using "block" all used connectors must have a flow direction. In Modelica the available directions are "input" or "output".	

Table 4-7 SysML Flow (FlowDirection) à Modelica Causality of connector instance

SysML (Rhapsody)	Modelica	Commonality
Generalisation/Specialisation	Inheritance (extends)	Inherit of structure and behaviour of another block.
A generalization describes a relationship between a general classifier and a specialized classifier. The specialized classifier can inherit structure and behaviour of the general classifier.	A block can inherit structure and behaviour of another block.	

Table 4-8 SysML Inheritance (Gen/Spec) à Modelica extends

SysML (Rhapsody)	Modelica	Commonality
Double	Real	A pre defined data type which represents floating point number values.
A pre defined data type which represents floating point number values.	A pre defined data type which represents floating point number values. A real variable has a set of attributes such as unit of measure, initial value, minimum and maximum value.	

Table 4-9 SysML Datatype Double à Modelica Datatype Real

4.1.2. Additional Stereotypes

SysML Stereotypes can be used to satisfy some semantics of the executable language.

a. There are four variability levels of attributes in Modelica, so a SysML “attribute” needs an additional Stereotype to recognise its variability. Also the optional unit of a value can be presented by a tag of this Stereotype.

<table>
<tr><th colspan="2">SysML (Rhapsody) element</th><th>Needed Modelica addition</th><th colspan="2">Benefit/Effect</th></tr>
<tr><td colspan="2">Attribute</td><td>variability</td><td colspan="2">Depending on its type an attribute will get a type prefix. The type also specifies how the initialisation will be defined.</td></tr>
<tr><th>Stereotype name</th><th>Tags</th><th>Characteristic</th><th colspan="2">Effect</th></tr>
<tr><td rowspan="5"><<variable>></td><td rowspan="4">variability</td><td>parameter</td><td>Prefix “parameter”</td><td>initialValue interpretation “…=x;”</td></tr>
<tr><td>constant</td><td>Prefix “constant”</td><td>initialValue interpretation “…=x;”</td></tr>
<tr><td>discrete-time</td><td>Prefix -non-</td><td>initialValue interpretation “(start = x, …)”</td></tr>
<tr><td>continuous-time</td><td>Prefix -non-</td><td>initialValue interpretation “(start = x, …)”</td></tr>
<tr><td>unit</td><td>String</td><td colspan="2">(…, unit=”…”)</td></tr>
</table>

Table 4-10 SysML Stereotype <<variable>> for Modelica variability and unit

b. A Stereotype is needed to modify the values of an extended class.

<table>
<tr><th colspan="2">SysML (Rhapsody) element</th><th>Needed Modelica addition</th><th>Benefit/Effect</th></tr>
<tr><td colspan="2">Generalisation path</td><td>Modify inherit variable values</td><td>Set default values for inherited attributes.</td></tr>
<tr><th>Stereotype name</th><th>Tags</th><th>Characteristic</th><th>Effect</th></tr>
<tr><td><<extendsRelation>></td><td>typeModification</td><td>String with Dot-Notation</td><td>extends … (…=x, …=x);</td></tr>
</table>

Table 4-11 SysML Stereotype <<extendsRelation>> for Modelica modification of inherit variable values

c. Since Rhapsody 7.2 does not support a SysML abstract block, a Stereotype has to be defined to specify a block as abstract.

SysML (Rhapsody) element		Needed Modelica addition	Benefit/Effect
Block		partial	A block which offers general structure and behaviour for a group of specialised block. This block can not be instantiated as a component.
Stereotype name	Tags	Effect	
		Prefix	
<<abstract>>	non	partial	

Table 4-12 SysML Stereotype <<abstract>> for Modelica partial

d. Modelica provides a method to modify variable values of instances by using the dot notation.

SysML (Rhapsody) element		Needed Modelica addition	Benefit/Effect
Instance		Instance modification	A value of an intern variable from an instance can be modified using the “dot notation”. The variable name and its new value in brackets will be appended to the instance declaration.
Stereotype name	Tags	Characteristic	Effect
<<composite><	instanceModification	String with Dot-Notation	*Instance*… (…=x, …=x);

Table 4-13 SysML Stereotype <<composite>> for Modelica instance modification

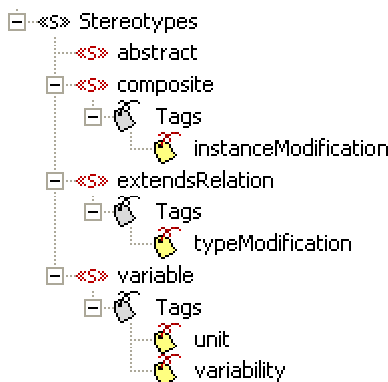

Figure 4-1 The created Stereotypes in Rhapsody

4.1.3. SysML Parametric to Modelica Equation

SysML Parametric diagrams are used to create systems of equations that can constrain properties of blocks. This diagram and a combination of the below described elements are used to generate a Modelica confirm equation.

SysML (Rhapsody) element	Description
Constraint Block Node	A constraint block encapsulates a constraint to enable it to be defined once and then used in different contexts. The block contains Constraints and Constraint parameters which are used in the constraints.
Constraint Property Node	Constraint properties are defined by constraint blocks and used to bind parameters. This enables complex systems of equations to be composed from more primitive equations, and for the parameters of the equations to explicitly constraint properties of blocks.
Constraint Parameter Node	A special kind of property that is used in the constraint expression of a constraint block. Constraint parameters do not have direction.
Value Binding Path	Binding connectors connect constraint parameters to each other and to value properties. They express an equality relationship between their bound elements.
Constraint	Generic mechanism for expressing constraints on a system as text expression that can be applied to any model element. A constraint includes an equation as text expression.

Table 4-14 SysML Parametric elements

The following is an approach to translate a SysML Parametric into a Modelica equation using the above depicted parametric elements:

- An equation which is represented as a constraint has to confirm to the Modelica syntax and semantic for equations.
- A parameter name in the constraint equation expression should be general, this supports the reuse approach of SysML.
- A constraint block contains only a single constraint and all its used constraint parameters. This is easier to understand and translate.
- A constraint property represents this constraint block in a parametric diagram.
- Binding connectors allocate the general constrain parameters to specific block values so that the equation can be translated into Modelica equation code with the required value names.

4.2. TanksConnectedPI System in SysML

The following is the full SysML model of the demonstration system. The model has been created with IBM Rhapsody 7.2 [30].

System structure is depicted in Block Definition Diagrams (BBD), Internal Block Diagrams (IBD) and in Parametric Diagrams (PAR). Every diagram will be translated in consequential Modelica code, which could afterwards merge to a coherent model code.

Since, the modelling tool IBM Rhapsody does not offer a SysML conform diagram frame, the frames will not be depicted in the diagrams.

Note: Translated code is highlighted in green.

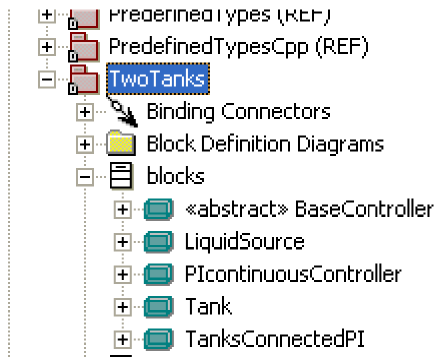

Figure 4-2 TwoTanks Package Structure

Blocks are grouped in packages. For OpenModelica it is important to signal a package membership for a block so that OM can load all classes include in the specified package. For this there is a need for a special “package” class in the project package folder. In addition, all classes need a “within…” declaration in their first code line.

Resultant Modelica code:

<<package>>TwoTanks à package.mo (Modelica)
`package TwoTanks` `end TwoTanks;`

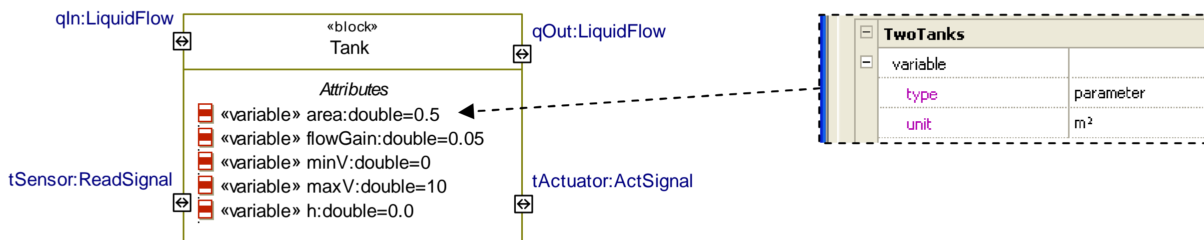

Figure 4-3 Tank Block

The tank is a SysML block with the depicted attributes and instances of flow Properties.

The variability of its attributes is explicitly given in the tag variability of its Stereotype variable. The Tag “unit” presents the unit of the value.

The package membership is given by the package structure in Figure 4-2.

Resultant Modelica code:

<<block>>Tank à Tank.mo (Modelica)

```
within TwoTanks;
block Tank
    ReadSignal tSensor;
    ActSignal tActuator;
    LiquidFlow qIn;
    LiquidFlow qOut;
    parameter Real area (unit = "m2") = 0.5;
    parameter Real flowGain (unit = "m2/s") = 0.05;
    parameter Real minV = 0
    parameter Real maxV = 10;
    Real h (start = 0.0, unit = "m");
end Tank;
```

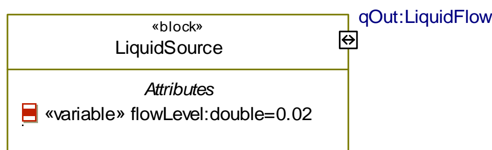

Figure 4-4 LiquidSource Block

Procedure is the same as translate Tank Block.

Resultant Modelica code:

LiquidSource.mo (Modelica)

```
within TwoTanks;
block LiquidSource
    LiquidFlow qOut;
    parameter Real flowLevel = 0.02;
end LiquidSource;
```

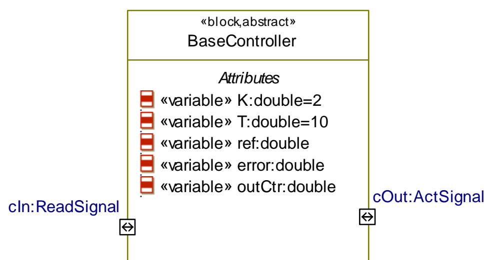

Figure 4-5 BaseController Block

The block in Figure 4-5 has the Stereotype abstract which signals that it is a partial Modelica block.

<u>Resultant Modelica code:</u>

<<block>> BaseController à BaseController.mo (Modelica)

```
within TwoTanks;
partial block BaseController
    ReadSignal cIn;
    ActSignal cOut;
    parameter Real ref;
    parameter Real K = 2;
    parameter Real T (unit = "s") = 10;
    Real error;
    Real outCtr;
end BaseController;
```

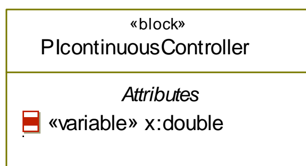

Figure 4-6 PIcontinuousController Block

<<block>> PIcontinuousController à PIcontinuousController.mo (Modelica)

```
within TwoTanks;
block PIcontinuousController;
    Real x;
end PIcontinuousController;
```

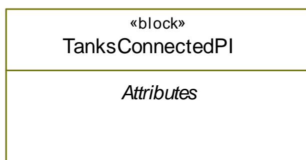

Figure 4-7 TanksConnectedPI Block

<<block>> TanksConnectedPI à TanksConnectedPI.mo (Modelica)

```
within TwoTanks;
block TanksConnectedPI
end TanksConnectedPI;
```

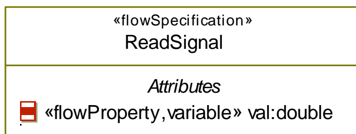

Figure 4-8 ReadSignal FlowSpecification

A flow specification attribute has the Stereotype “flowProperty” which states that each flow property has a data type and a direction (in, out, or inout). To use a flow specification for different instances with different flow directions the attribute direction has to be ignored. Also the Stereotype variable is selected for a flow specification attribute to assign a variability type and a unit for it.

<u>Resultant Modelica code:</u>

<<flowProperty>> ReadSignal à ReadSignal.mo (Modelica)

```
within TwoTanks;
connector ReadSignal
    Real val (unit = "m");
end ReadSignal;
```

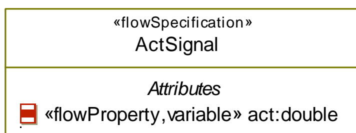

Figure 4-9 ActSignal FlowSpecification

<< flowProperty >> ActSignal à ActSignal.mo (Modelica)

```
within TwoTanks;
connector ActSignal
    Real act;
end ActSignal;
```

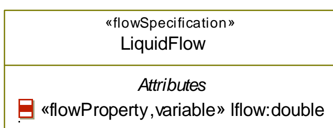

Figure 4-10 LiquidFlow FlowSpecification

<< flowProperty >> LiquidFlow à LiquidFlow.mo (Modelica)

```
within TwoTanks;
connector LiquidFlow
end LiquidFlow;
```

4.2.1. System structure with SysML Block Definition Diagram and SysML Internal Block- Diagram

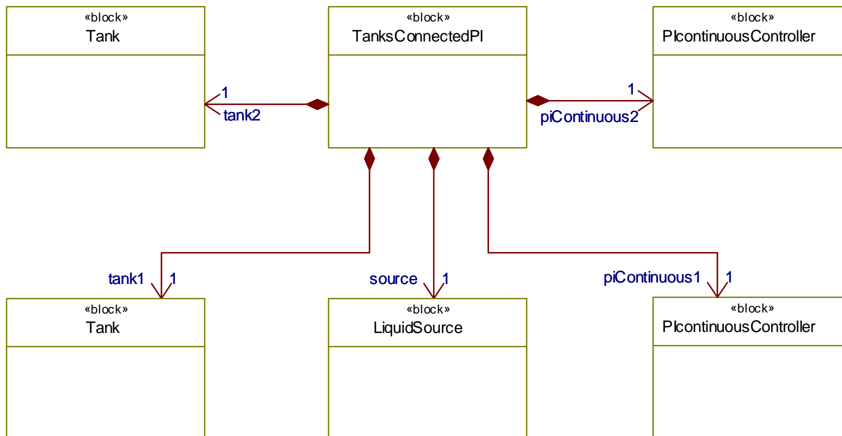

Figure 4-11 BBD TanksConnectedPI

The BDD in Figure 4-11 presents the “TanksConnectedPI” and its relationship to other components, connected with compositions.

<u>Resultant Modelica code:</u>

TanksConnectedPI.mo (Modelica)

```
block TanksConnectedPI
    LiquidSource source;
    Tank tank1;
    Tank tank2;
    PIcontinuousController piContinuous1;
    PIcontinuousController piContinuous2;
end TanksConnectedPI;
```

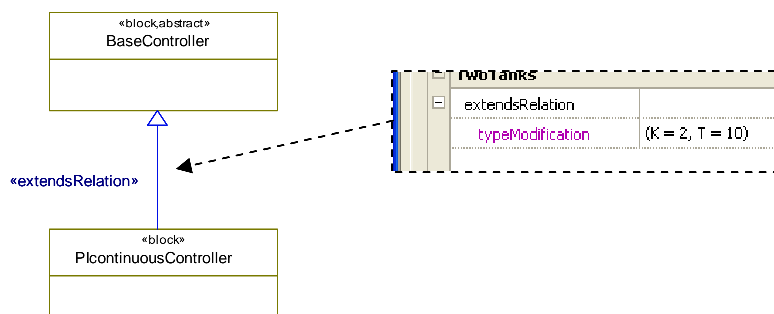

Figure 4-12 Inheritance between BaseController and PIcontinuousController

To cover also a part of the object oriented approach “PIcontinuousController” inherits behaviour from “BaseController”. These components are also SysML blocks. To modify inherited variable values the Stereotype “extendedRelation” gives a modifier string in its tag “typeModification”. In This case “K” and “T” are inherited variables to be modified.

<u>Resultant Modelica code:</u>

<<block>> PIcontinuousController à PIcontinuousController.mo (Modelica)

```
block PIcontinuousController extends BaseController (K = 2, T = 10);
    ...
end PIcontinuousController;
```

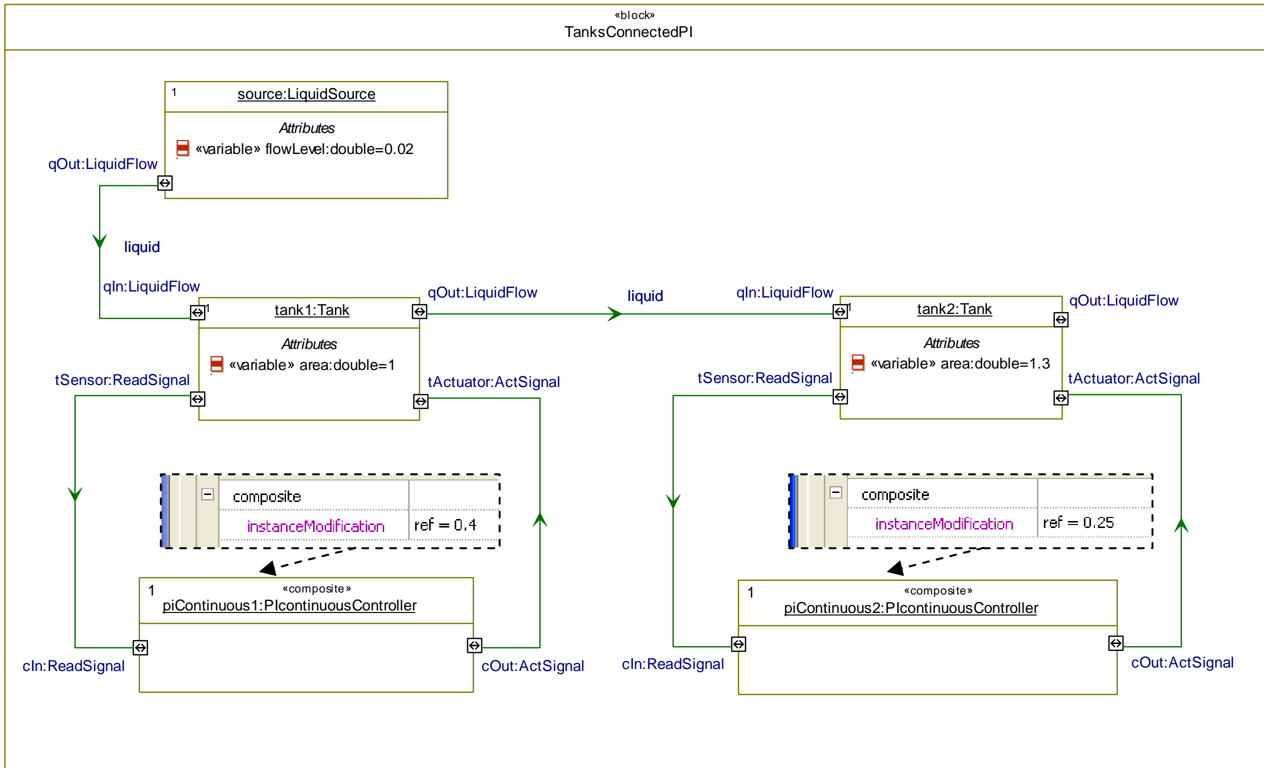

Figure 4-13 IBD TanksConnectedPI

The IBD in Figure 4-13 displays the components of the “TanksConnectedPI” system and their associations to each other. Specialised attributes of the parts (in the first level) can be modified directly in the Tab “Attributes” à “Value”, for example “area” from “Tank” and “flowLevel” from “LiquidSource”. Since Rhapsody does not offer full instance modification the modification of inherit attribute values and nested attribute values must be done manually by the Stereotype “composite” and its tag “instanceModification” as a textual expression, for example “ref” from “PIcontinuousController”. The parts are connected with SysML flow ports.

Resultant Modelica code:

LiquidSource.mo (Modelica)

```
block LiquidSource
    output LiquidFlow qOut;
    ...
end LiquidSource;
```

<<block>>Tank à Tank.mo (Modelica)

```
block Tank
    output ReadSignal tSensor;
    input ActSignal tActuator;
    input LiquidFlow qIn;
    output LiquidFlow qOut;
    ...
end Tank;
```

<<block>> BaseController à BaseController.mo (Modelica)

```
block BaseController
    input ReadSignal cIn;
    output ActSignal cOut;
    ...
end BaseController;
```

TanksConnectedPI.mo (Modelica)

```
block 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(piContinuous1.cOut, tank1.tActuator);
    connect(tank1.tSensor, piContinuous1.cIn);
    connect(tank1.qOut, tank2.qIn);
    connect(piContinuous2.cOut, tank2.tActuator);
    connect(tank2.tSensor, piContinuous2.cIn);
end TanksConnectedPI;
```

4.2.2. Block Definition Diagrams of the constraint blocks

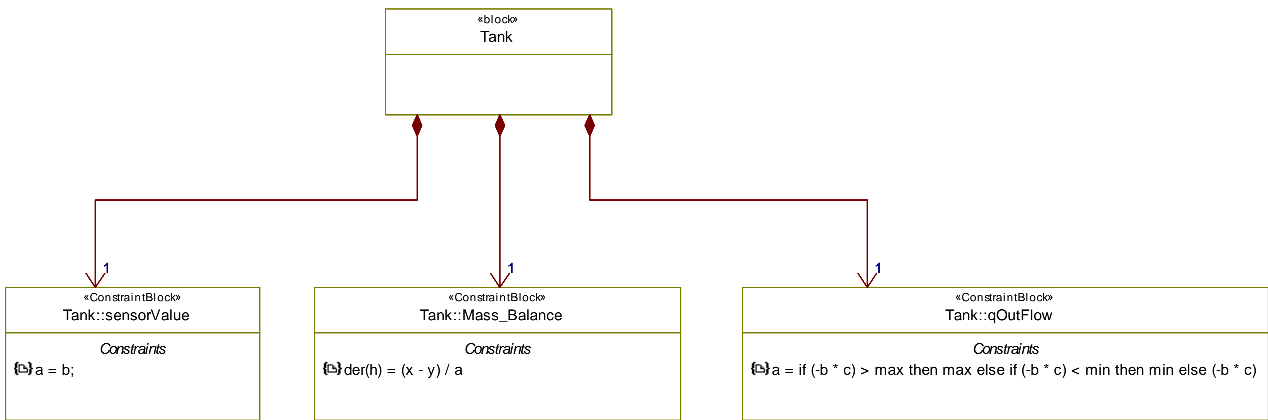

Figure 4-14 BDD Tank Constraints

- The “sensorValue” constraint allocates the tank level “h” to the flow port which is connected to the “PIcontinuousController”. In this case it is a simple allocation but it could also be a complex equation, therefore it is realised as a constraint.
- The “MassBalance” constraint describes how the tank level “h” is identified.
- The “qOutFlow” constraint defines the value of the out flow level depending on some internal attributes and the calculation result of the “PIcontinuousController”.

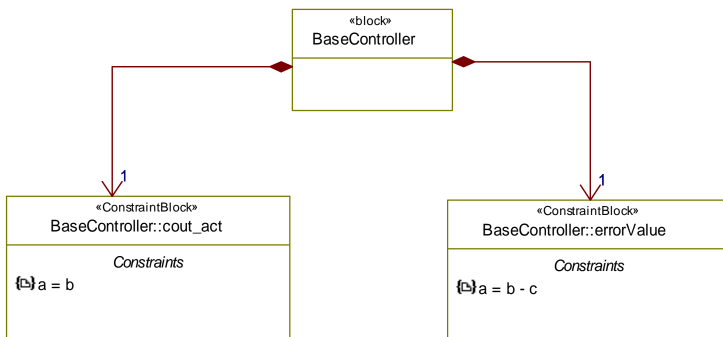

Figure 4-15 BDD BaseController Constraints

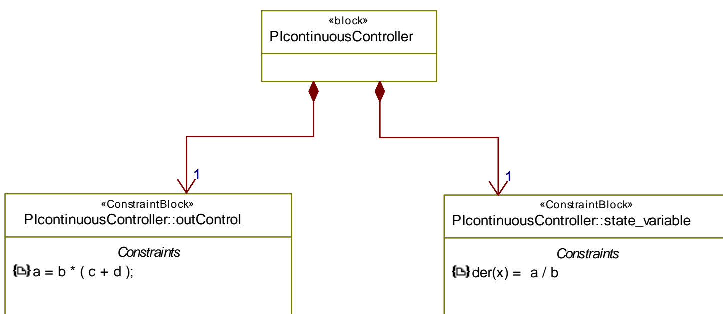

Figure 4-16 BDD PIcontinuousController Constraints

“BaseController” and “PIcontinuousController” provide constraints which calculate the required flow gain to remove excess liquid from the tank depending on the tank level.

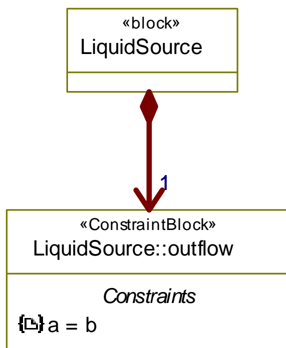

Figure 4-17 BDD LiquidSource Constraints

The liquid source constraint allocates the flow level to the outgoing flow port, which is connected to the first tank. In this case it is a simple allocation but it could also be a complex equation, therefore it is represented as a constraint.

4.2.3. Parametric Diagrams of the parametrics structure

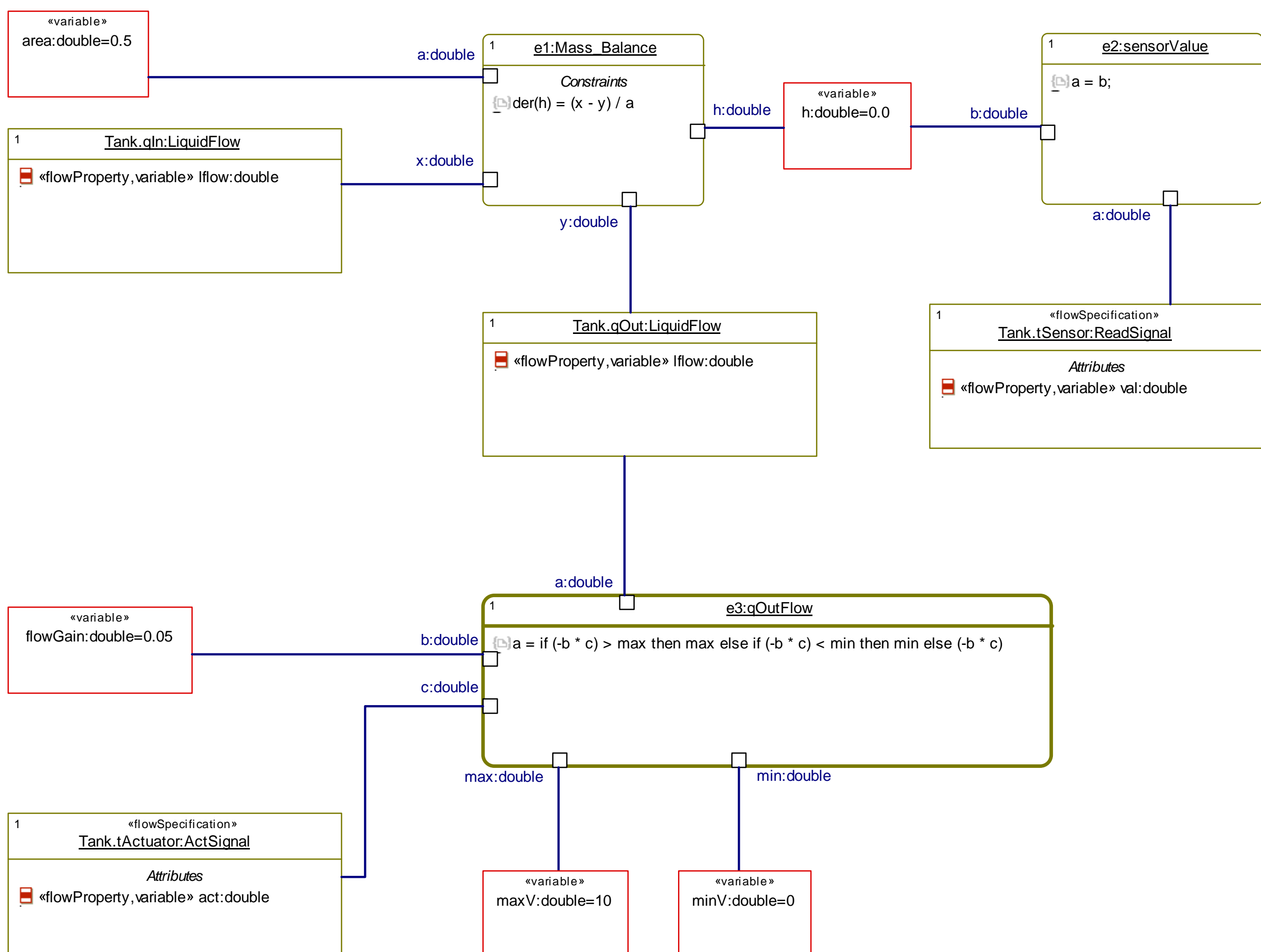

Figure 4-18 PAR Tank

The constraint property “e1:Mass_Balance” describes a special equation type. As depicted in Figure 4-18 the derivative of “h” will be allocated to the model attribute h, but the real values are quite different. This is possible because Modelica integrates the result of

“der(h)” to get the value for “h” automatically, so there is no need for a separate equation for this. The translation will be done as described in chapter 4.1.3 with respect to the Modelica equation syntax and semantic.

Resultant Modelica code:

<<block>>Tank à Tank.mo (Modelica)

```
block Tank
    ...
equation
    der(h) = (qIn.lflow - qOut.lflow) / area;
    qOut.lflow = if (-flowGain * tActuator.act) > maxV then maxV else if
      (-flowGain * tActuator.act) < minV then minV else (-flowGain *
      tActuator.act);
    tSensor.val = h;
end Tank;
```

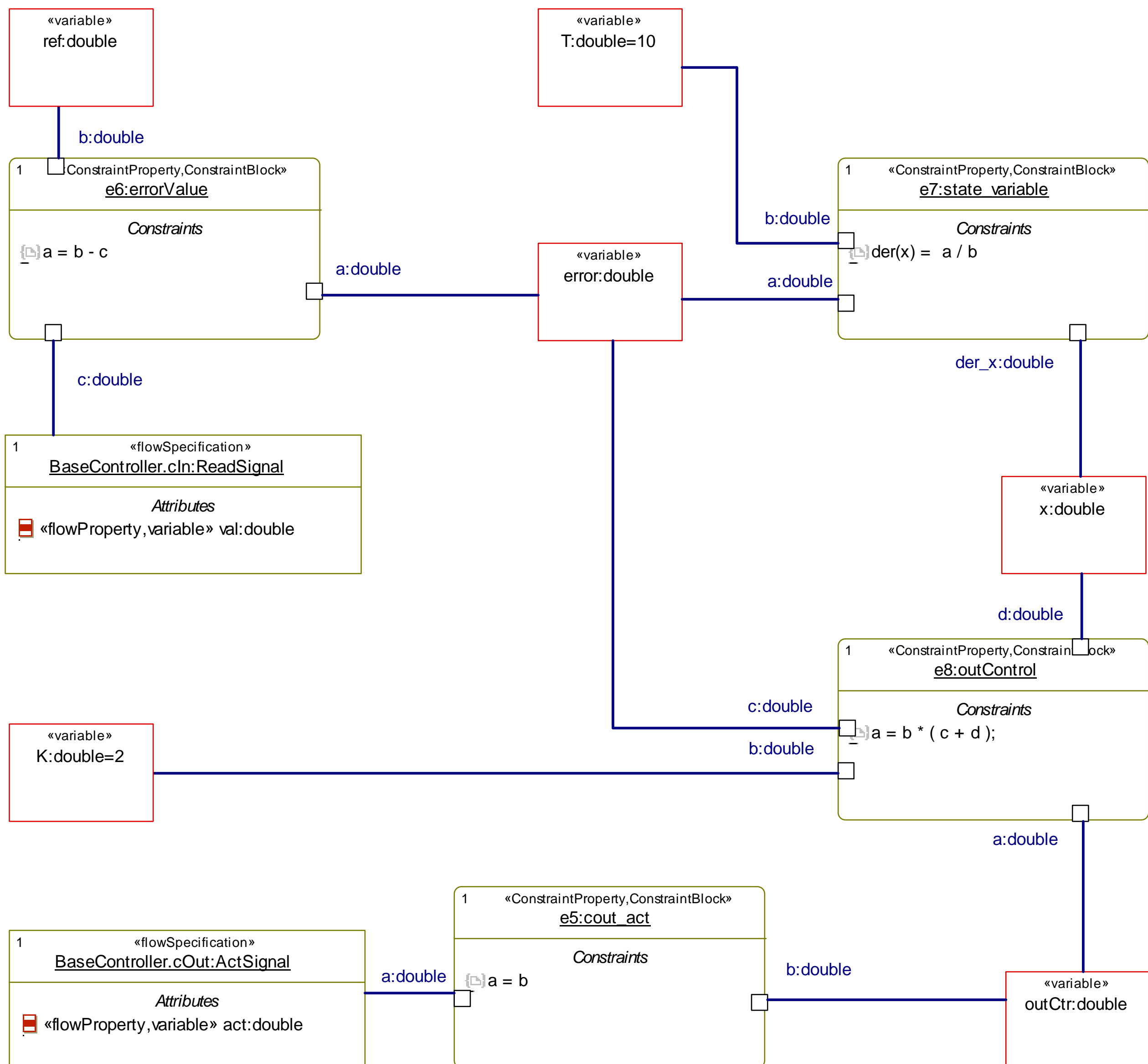

Figure 4-19 PAR BaseController and PIcontinuousController

Same procedure as translation of “PAR Tank”.

Resultant Modelica code:

<<block>> BaseController à BaseController.mo (Modelica)

```
partial block BaseController
    ...
equation
    error = ref - cIn.val;
    cOut.act = outCtr;
end BaseController;
```

<<block>> PIcontinuousController à PIcontinuousController.mo (Modelica)

```
block PIcontinuousController extends BaseController;
    ...
equation
    der(x) = error / T;
    outCtr = K * (error + x);
end PIcontinuousController;
```

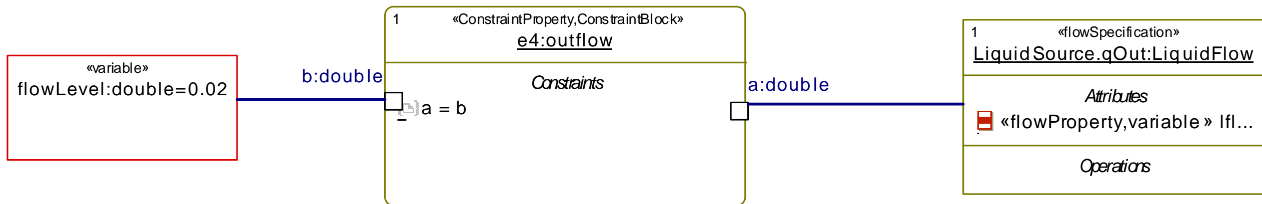

Figure 4-20 PAR Outgoing flow level of the LiquidSource

Same procedure as translation of “PAR Tank”.

Resultant Modelica code:

<<block>> LiquidSource à LiquidSource.mo (Modelica)

```
block LiquidSource
    ...
equation
    qOut.lflow = flowLevel;
end LiquidSource;
```

5. Interactive Simulation Runtime

A simulation runtime is needed to simulate a system. In this case the simulation runtime is combined with the system model which is represented in the executable programming language C/C++. The physical system behaviour is represented as mathematical equations which are time dependent.

The following are some general requirements for an interactive simulation runtime:

- The user shall be able to stimulate the system during a running system simulation and to observe its' reaction immediately.
- Simulation runtime behaviour has to be controllable and adaptable to offer an interaction with a user.
- A user should receive simulation results during a simulation in "real-time" to realise real-time simulation. Since network process time and some other factors like scheduling of processes from the operation system this is not given at any time.
- In order to offer a stable simulation, a runtime has to inform a GUI of errors and consequential simulation aborts.
- Simulation results should not under-run or exceed a tolerance compared to a thoroughly reliable value, for a correct simulation.
- Communication between a simulation runtime and a user GUI should use a well defined interface and be base on a common technology, for example message parsing, CORBA or RMI.
- An interactive simulation runtime should be based on OM, since the OM simulation runtime is the only open source, and as far as is possible, stable Modelica simulation tool [28].

As mentioned above, the OM simulation runtime has no real-time simulation capabilities and does not provide any user interaction while the simulation is running.
The following are some identified modifications and expansions of the existing source code which are needed to fulfil the general requirements:

- Real-time and network communication capabilities expansion: In order to offer a user-interactive and real-time simulation we need, for example, threading, network protocols and synchronization units.

- Management of resources: De-allocation of used memory after a simulation step, release and deletion of all synchronisation units and deletion of all sockets.
- Modification of data storage and in/out operations: Removal of unnecessary in/out operations and other overhead.

Unfortunately the OM system is not subject to any specific or general software architecture and no standard programming style is identifiable. No UML diagrams were created for the OM system and the source code is inadequately documented. The principles of modularisation, information hiding and many other development patterns have not been respected. An attempt to modify and expand the existing modules of OM failed after many attempts because of the above grievances in its documentation and programming style. For example, an attempt to modify the solver system to slow down its calculation frequency or change its variable data failed because of unanticipated behaviour during calculations.

5.1. OpenModelica Interactive

The new simulation runtime will be called “OpenModelica Interactive” (OMI).
OMI is an executable simulation application. The executable file will be generated by the OMC, which contains the full Modelica (SysML) model as C/C++ code with all required equations, conditions and a solver to simulate a whole system or a single system component. However, the best way to expand the existing code with the required capabilities is to separate the OMI system into different subsystems, which will also support the modularisation and information hiding principles. The separation into subsystems is attached to the service-oriented architecture, which has the advantage of replacing, modifying and expanding the single subsystems without changes to the other subsystem components.
The OMI is separated into two subsystems:

- The old modified OM Subsystem
- The new OMI Subsystem

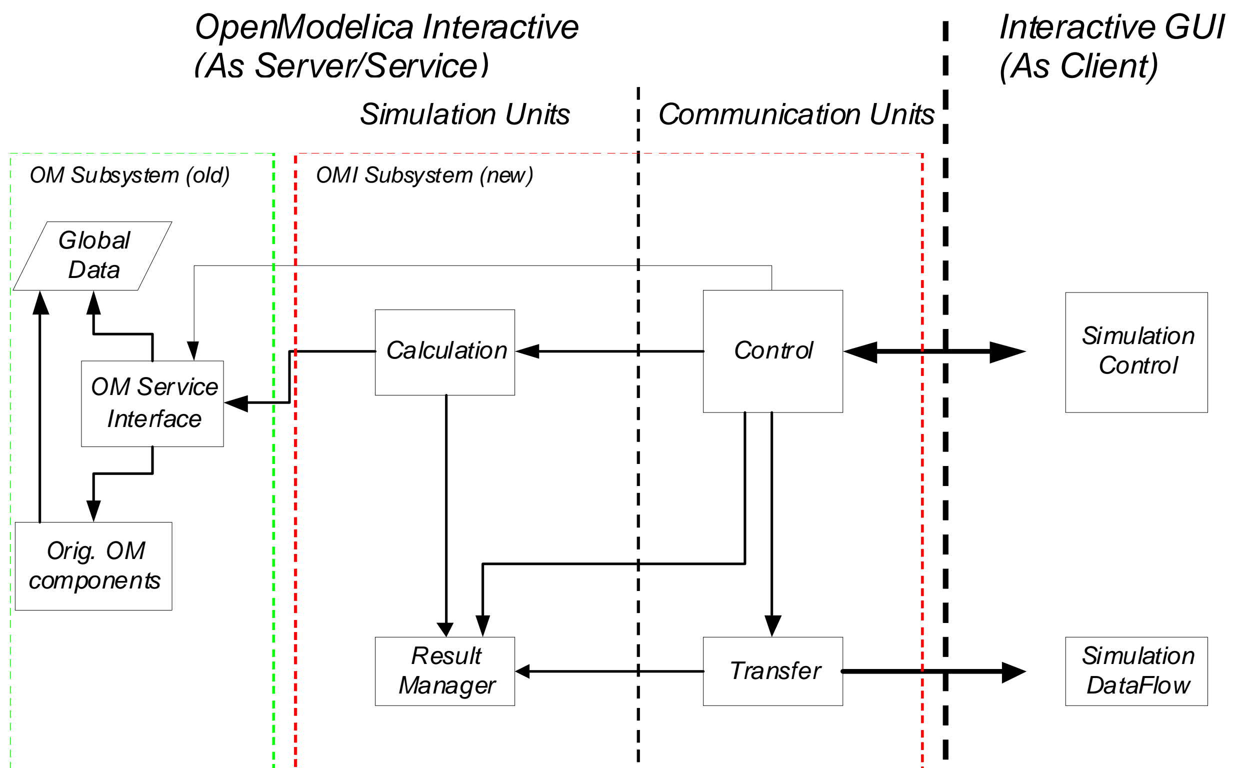

Figure 5-1 OpenModelica Interactive System Architecture Overview

5.1.1. The OpenModelica Subsystem

The OpenModelica subsystem consists of the partially modified “Orig. OM components” and a global data structure, as shown in Figure 2-5.

Modifications:

- Following a calculation stage, the results will be printed into a file in preparation for plotting. OMI does not need this result file. In order to improve the performance this function has to be removed from the “Solver_DASRT”.
- The “Orig. OM components” use many variables which are stored in the global scope. These global scope variables must be reinitialized before running a new solving step, otherwise the solver will not calculate the results correctly.
- Allocated memory must be released after a solving step and also a whole simulation run also. The OM system has to De-allocate this memory after every solving step.

Expansion:

- The new OMI subsystem components need to be called when a simulation begins. This will be done from the main function in “simulation_runtime.cpp”, which starts the “OMI_Control”, which takes over the whole simulation control.

- The access to the simulation data “global_data” needs to be synchronized, therefore a Mutex is implemented, which controls the access between the OM components, such as the solver, and the OMI subsystem components.
- OM Service Interface: A unit which controls all access to the OM subsystem components from other subsystems. All parallel activities on the OM will be synchronized.

5.1.1.1. OpenModelica Subsystem Service Interface

The OM subsystem offers three main services: A Simulation Data-, a Simulation Data-Name and a Solving- Service.

- Simulation Data Service: Model and simulation specific data for example variable names, values and numbers, are stored in a global data structure. Most of this data needs to be changed during the simulation single steps, but some data are static, for example the step time. This service provides data query and manipulation.
- Simulation Data Name Service: Returns the model data names as string for example variable or parameter names, this will be used to generate the filter mask as mentioned in chapter 5.1.2.4.
- Solving Service: This service simulates a Modelica model for a specific time interval by using a DASSL solver and the standard OM components. It sends the result for a stop time to the caller and stores it to the global data structure. Some parameters are needed to use the solving service from the OM Subsystem:
  - Start time
  - Stop time
  - A tolerance for results

5.1.2. The OpenModelica Interactive Subsystem

The OpenModelica Interactive subsystem uses the above mentioned services to simulate a Modelica model without any knowledge of used solvers, equations and conditions. The subsystem is also separated into different modules.

5.1.2.1. OMI::Control

The “Control” module is the interface between OMI and a GUI. It is implemented as a single thread to support parallel tasks and independent reactivity. As the main controlling and communication instance at simulation initialisation phase and while simulation is running it manages simulation properties and also behaviour. A client can permanently

send operations as messages to the “Control” unit, it can react at any time to feedback from the “Calculation” or “Transfer” threads and it also sends messages to a client, for example error or status messages.

The following are its main tasks:

- Waiting for a GUI to connect with, based on the network communication protocols TCP/IP.
- Waiting for a request or an error and abort message from a GUI.
- Handling of a GUI request and replying with the correct execution with a done message.
- Managing all “Calculation” and “Transfer” threads from the OMI subsystem.
- Watching for feedback from a global error handler which handles all occurred errors from “Transfer”, “Calculation” and “Control” threads in the form of an error message.
- Informing a GUI if a fatal error occurs.

5.1.2.2. OMI::ResultManager

While a simulation is running the “Calculation” thread produces simulation results for every time step, and the “Transfer” thread sends the single results to a client. There is a need for synchronization and organisation of simulation results. However, the application cannot store all results because this would cause the system to run out of memory.

This scenario is the typical “producer and consumer problem with restricted buffer”, which is well known in IT science.

The “ResultManager” assumes responsibility for organizing simulation result data and synchronizing access to these data.

Simulation Step Data (SSD)

The main unit of the “ResultManagers” is a collection of simulation step data elements (SimulationStepData) which contain all important result values for each simulation step. The “OM Solver” needs the following data for every single simulation step to solve the equations and to confirm the conditions:

- A time stamp which marks for what time step these data represent.
- All state values and their derivatives.
- All algebraic values.
- All parameter values.

This container is restricted to prevent the system running out of memory.

Simulation Result Data for Forwarding (SRDF)

Main organisation and management tasks while sending data to a GUI:

- Organise which data should be send to a GUI.
- Organise which data are obsolete.
- Manage how to synchronize the access from the different producers and consumers.
- Manage how the producers and consumers should inform about free slot.
- Manage how the producers and consumers should inform about new results.

The “simulation result data for forwarding” (SRDF) is a container which contains references to slots of the SSD array. This container is implemented as an array.
The buffer is restricted to “n” elements. This is important because a “Calculation” thread could be much faster than the “Transfer” thread, which would cause the system to run out of memory. Also “SRDF” is organized as a queue so it based on the principle of First in First out (FIFO). This is the above mentioned typical “producer and consumer problem with a restricted buffer”.

The following is a brief description of the organisation of the data array “SRDF” based on a short example:

t_n: Simulation result for the time n (C++ structure).
arr_srdf[n]: Array buffer with the maximum size “n”, starts at address “1000”.

ptf ●: Pointer to the first element i.e. least t_n appendage the FIFO principles.

- If “ptf” points to a slot with a null, “pop” does not work.

ptd ▲: Pointer to the next free slot, where an element t_n could be inserted.

- If “ptd” points to a busy slot, push does not work.

push: Insert a t_n into “arr_srdf”.

pop: Take and remove a t_n from the “arr_srdf”.
laa = Last array address.

Initialization Phase and example push, pop operations as pseudo code:
- arr_srdf[n] initialized with null
- ptf = arr_srdf;
- ptd = arr_srdf;

- laa = &arr_srdff[n-1] //Last Array Address in this case 1028

1000	1004	1008	1012	1016	1020	1024	1028
null	null	null	null	null	null	null	null
●▲							

1. push t_0 à if *ptd == null, then: (*ptd = t_0, if ptd != laa then: pdt++ else: ptd = arr_srdf)

t_0	null	null	null	null	null	null	null
●	▲						

2. push t_1 à if *ptd == null, then: (*ptd = t_1, if ptd != laa then: pdt++ else: ptd = arr_srdf)

t_0	t_1	null	null	null	null	null	null
●		▲					

3. push t_6 à if *ptd == null, then: (*ptd = t_6, if ptd != laa then: pdt++ else: ptd = arr_srdf)

t_6	t_1	t_2	t_3	t_4	t_5	t_6	null
●						▲	

4. pop à if *ptf != null, then: (get(*ptf), *ptf = null, if ptf != laa then: pft++ else: ptf = arr_srdf))

null	null	t_2	t_3	t_4	t_5	t_6	null
	●					▲	

5. pop à if *ptf != null, then: (get(*ptf), *ptf = null, if ptf != laa then: pft++ else: ptf = arr_srdf))

Case array buffer is full: Cannot perform a push action, until the slot which the ptd is pointing on is not null.

t_8	t_9	t_2	t_3	t_4	t_5	t_6	t_7
		▲●					

Case array buffer is empty: Cannot perform a pop action, until the slot which the ptf is pointing on is not filled with a t_n

null	null	null	null	null	null	null	null
				●▲			

```
push(result tn )
{
      If(*ptd == null)
      {
            *ptd = tn ;
            If(ptd != laa)
                  ptd++;
            else: ptd = arr_srdf;
      }
      else: Can't push tn because there is no free slot
}

pop()
{
      if(i*ptf != null)
      {
            do(*ptf);
            *ptf = null;
            If(ptf != laa)
                  ptf++;
            else: ptf = arr_srdf;
      }
      else: Can't pop an element because the buffer is empty
}
```

Figure 5-2 Pseudo code of push and pull in SRDF

The computer science has a design pattern to solve the "producer and consumer problem with restricted buffer". It will use Semaphores and Mutexes. Involved members are "Calculation" as producer and "Transfer" as consumer.

5.1.2.3. OMI::Calculation

The "Calculation" thread is synonymous to a producer which uses the "OM Solving Service" to get results for a specific time step and to inform the "ResultManager" about the new simulation results. It uses the parameters described in 5.1.1.1. to calculate the interval between single calculation steps ($T_n \rightarrow T_{n+1}$) in a loop, until the simulation is interrupted by the "Control" or because of an occurred error.
If a single solving step is very complex and takes a long time to be solved, it is possible to create more than one producer to start the next simulation step during the data storing time.

5.1.2.4. OMI::Transfer

Similar to a consumer, the "Transfer" thread tries to get simulation results from the "ResultManager" and send them to the GUI immediately after starting a simulation. If the

communication takes longer than a calculation step, it is also possible to create more than one consumer.

The "Transfer" uses a property filter mask containing all property names whoes result values are important for the GUI. The GUI must set this mask using the "setfilter" operation from chapter 5.1.3.2, otherwise the transfer sends only the actual simulation time. This is very useful for increasing the communication speed while sending results to the GUI.

5.1.3. Communication Interface (Architecture)

As depicted in Figure 5-1 the behaviour between the OMI and a GUI is like a server and client behaviour respectively.

5.1.3.1. Communication

There are some possible technologies to realise the communication between the OMI and a GUI. The following are some of these technologies:

- CORBA: The "Common Object Requesting Broker Architecture" is a standard defined by the OMG which enables software components written in multiple computer languages to work together. This specification offers a name service, object management service and some other very useful concepts.
- Message Parsing using a common network communication technology: The principle of message parsing is used when an application does not have shared memory. It is used in combination with a network communication technology when the information exchange can be constructed on a basic structure, for example strings.

For the OMI realisation CORBA is too overloaded. The name service will not be used because there is only one single simulation runtime and only one GUI. There are no objects on the "C++" simulation runtime side. However, message parsing using a common network technology seems to be the most suitable way.

The network communication technology "TCP/IP" will be used to send and receive messages; it has many advantages compared with "UDP/IP" [8]. Each system has its own server and client implementations to receive and send messages respectively.

For an example system application the servers and clients will get static IP addresses.

Name	Description	URL
Control Server	Waits for requests from the GUI	Waits for connection on: 127.0.0.1:10501
Control Client	Replies to the GUI and sends other synchronization messages to it	Tries to connect on: 127.0.0.1:10500
Transfer Client	Sends simulation results to a GUI	Tries to connect on: 127.0.0.1:10502

Table 5-1 OMI server and client components

Name	Description	URL
Control Client	Requests to the OMI Control Server	Tries to connect on: 127.0.0.1:10500
Control Server	Waits for information from the OMI Control Client	Waits for connection on: 127.0.0.1:10500
Transfer Server	Waits for simulation results from the OMI Transfer Client	Waits for connection on: 127.0.0.1:10502

Table 5-2 GUI server and client components

5.1.3.2. Operation Messages

To use messages parsing there is a need to specify a communications protocol.
A string message begins with a specified prefix and ends with a specified suffix.
The prefix describes the request type, for example an operation. Depending on the request type, some additional information and parameters can appended on it. The suffix is to check if the message has been received correctly and if the sender has created it correctly. All parts should be separated with “#”.
The following are all available message strings between a GUI and the OMI system:

Request from GUI to OMI::Control

GUI Request	Description	OMI::Control Reply
start#end	Starts or continues the simulation	done#end
pause#end	Pauses the running simulation	done#end
stop#end	Stops the running simulation and resets all values to the beginning	done#end
shutdown#end	Shuts the simulation down	done#end
setfilter# var1:var2# par1:par2# end	Sets the filter for variables and parameters which should send from OMI to the client GUI	done#end

changetime#Tn#end	Changes the simulation time and goes back to a specific time step	done#end
changevalue#Tn# par1=2.3:par2=33.3# end	Changes the value of the appended parameters and stets the simulation time back to the point where the user clicked in the GUI	done#end
error#TYPE#end	Error handling not implemented yet	Error: *

Table 5-3 Available messages from a GUI to OMI (Request-Reply)

Messages from OMI::Control to GUI

OMI::Control	Description	GUI
Error: MESSAGE	If an error occurs the OMI::Control generates an error messages and sends the this messages with the prefix “Error:” to the GUI	Up to the GUI developers

Table 5-4 Available messages from OMI::Control to GUI

Messages from OMI::Transfer to GUI

OMI::Transfer	Description	GUI
result#ID#Tn# var1=Val:var2=Val# par1=Val:par2=Val# end	Sends the simulation result for a time step Tn to the client GUI. Maybe an ID is important to identify the results which are obsolete (not implemented yet).	none

Table 5-5 Available messages from OMI::Transfer to GUI

5.1.4. OpenModelica Interactive Structure and Behaviour

The OMI structure and behaviour will be represented as UML diagrams. Use cases will be illustrated in UML Sequence diagrams.

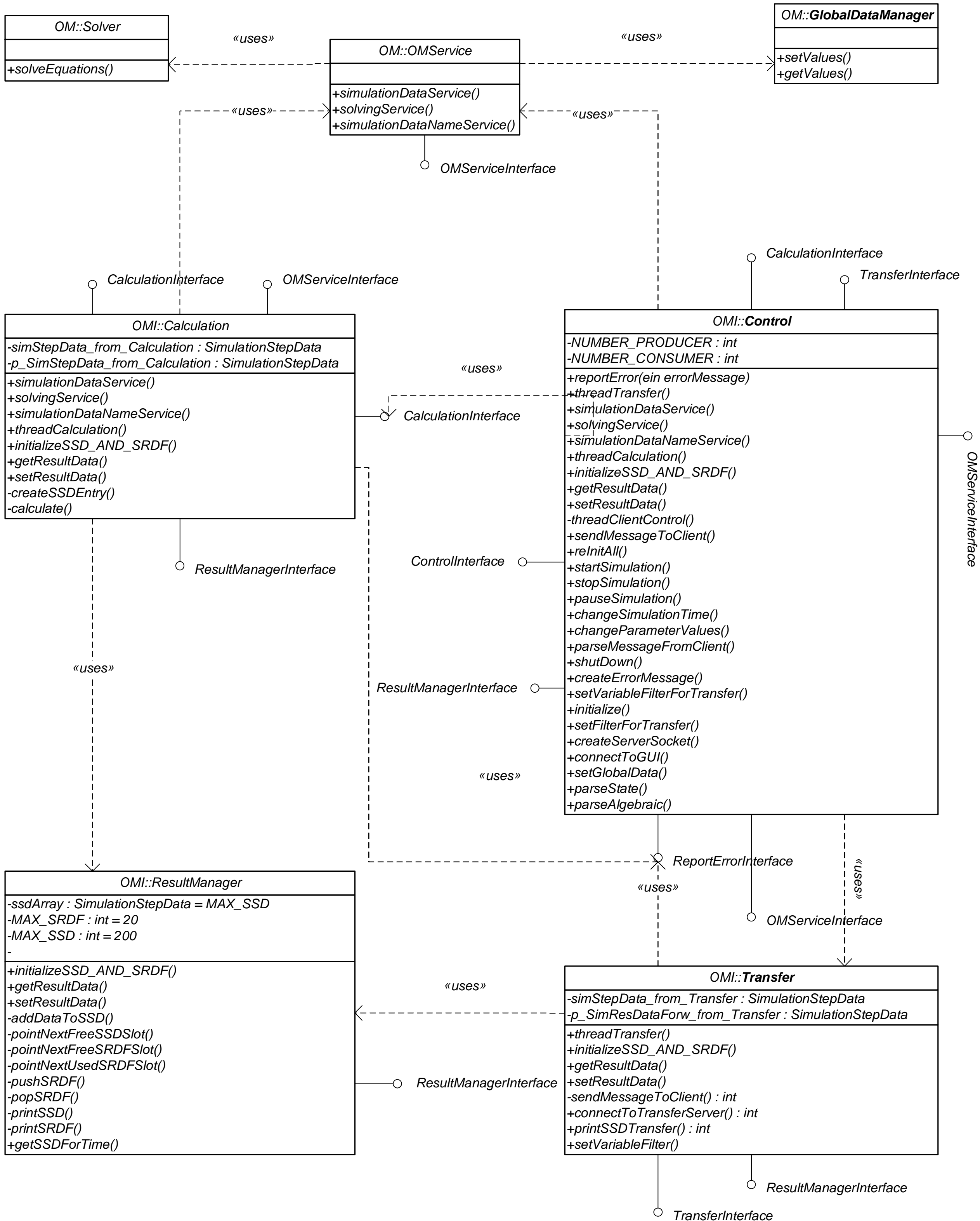

Figure 5-3 UML-Structure OM and OMI with some attributes and methods

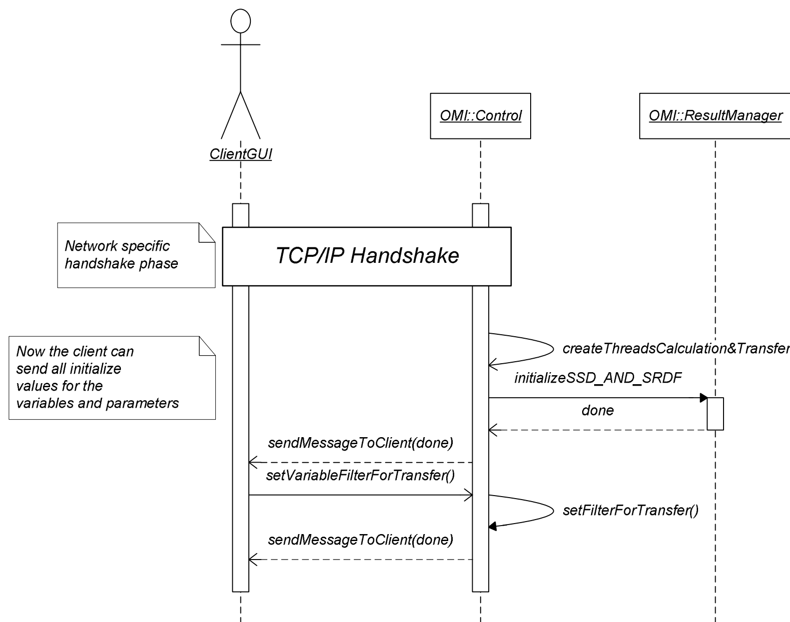

Figure 5-4 UML-Seq Handshake, model initialization and set Transfer filter mask

The UML-Sequence diagram in Figure 5-4 illustrates the network specific handshake phase, the model initialization phase, which includes creation and initialisation of all producers and consumers, and the definition of the filter mask for the consumers (Transfer threads) the filter message is the “setfilter” operation from Table 5-3.

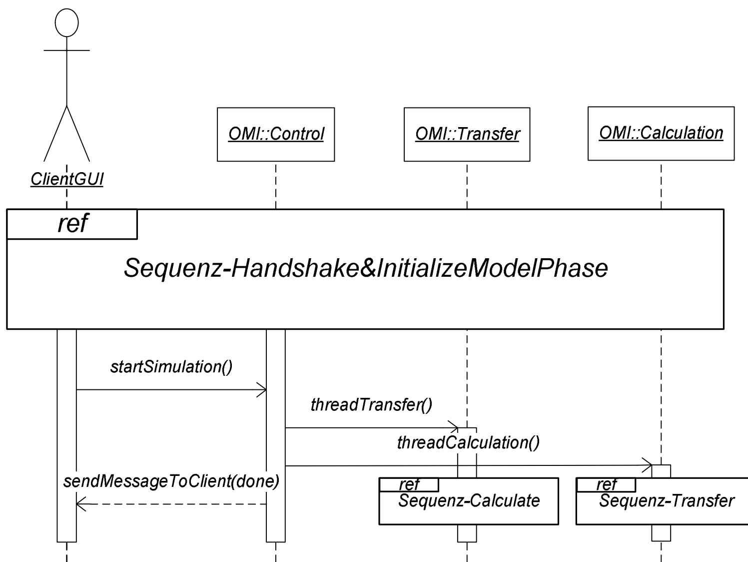

Figure 5-5 UML-Seq Simulation start

After the initialization phase the client can start the simulation with the message “start” from Table 5-3. This will cause the “OMI::Control” to start all producers and consumers so they will calculate and send results respectively.

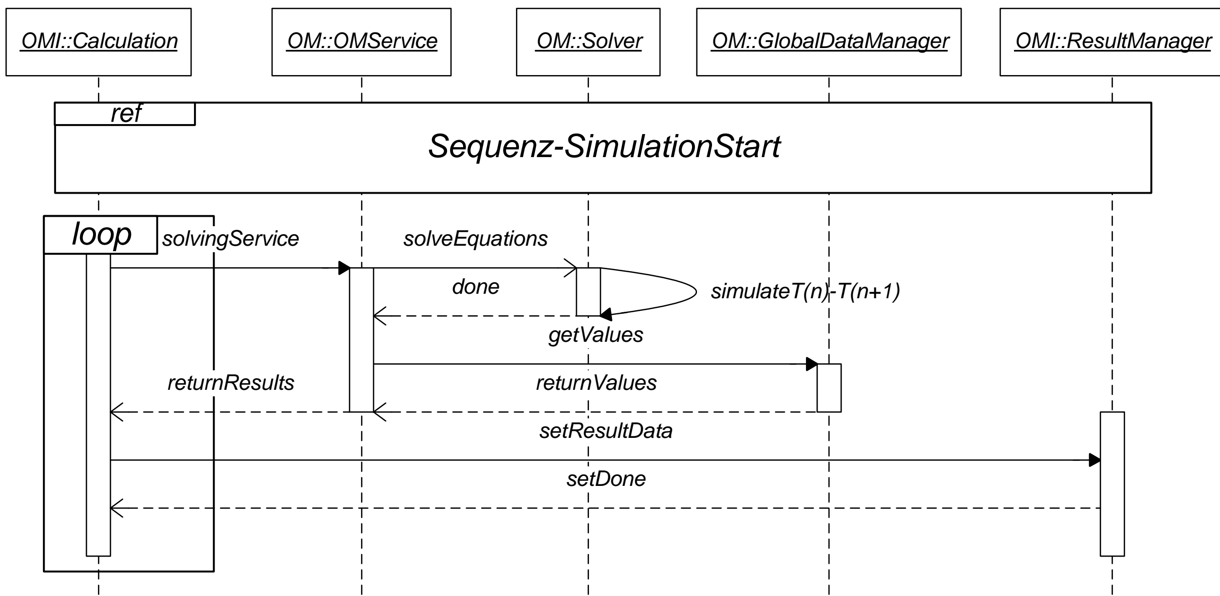

Figure 5-6 UML-Seq Calculation phase

After simulating T(n) to T(n+1) the result must set to the “SimulateStepData” collection. The “setResultData()” method is synchronized and the caller must wait if a mutex or the semaphore is in use.

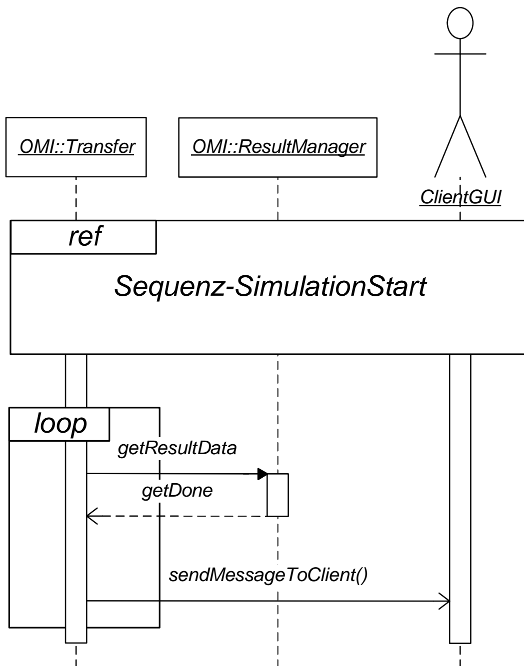

Figure 5-7 UML-Seq Transfer to client phase

The “Transfer” thread calls the “getResultData” method in a loop and waits for new results referenced in the “SimulateStepData” collection to send them to a GUI.

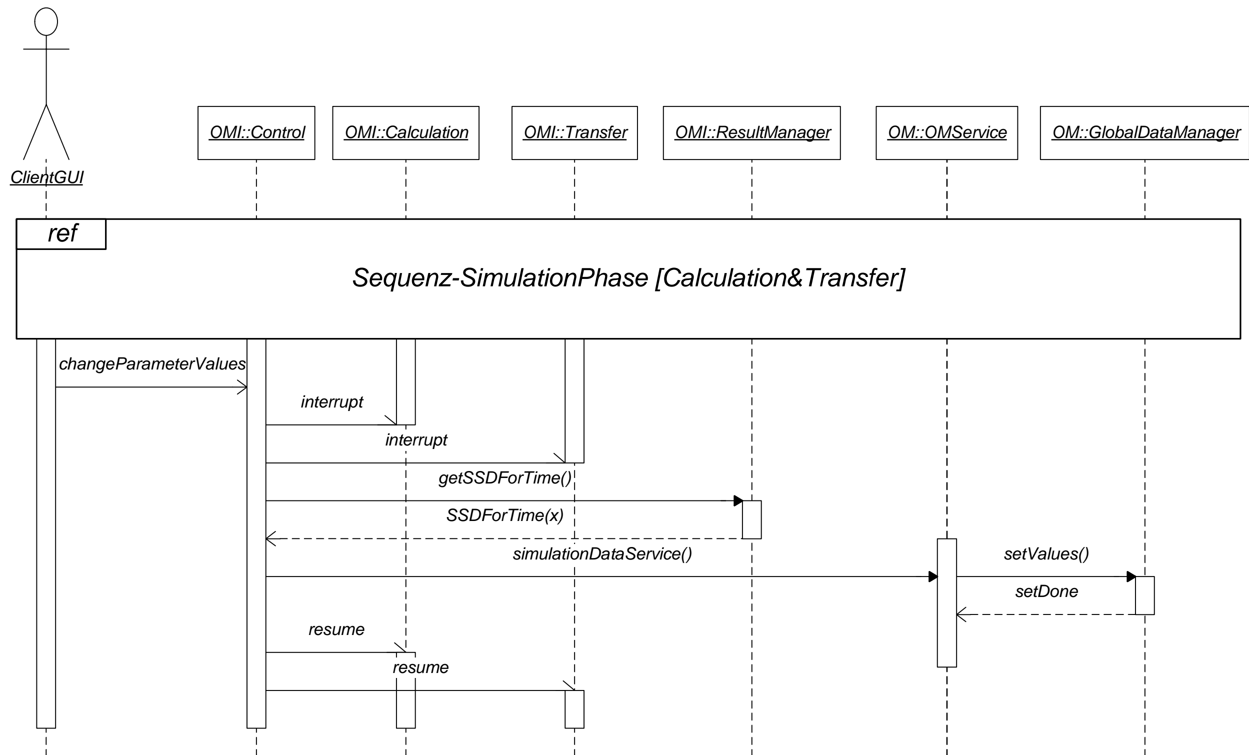

Figure 5-8 UML-Seq Change Value of a parameters

A more complex sequence is changing parameter values. The client sends a "changevalue" message with a time T(n) and the new values. "Control" interrupts all producers and consumers so it can access on the "SSD" and "SRDF" of the "ResultManager". "Control" uses the "OM::Service" to put the new values into the global data structure. After this, it resets the data in to "SSD" by using data from the time step T(n) and resumes all components.

5.1.5. Testing of the OpenModelica Interactive simulation runtime

Since rounding errors occur while storing and recalling result values by the "OMI::ResultManager", the "OM::Solver" will get changed values compared to the non Real-time calculation of OM.

5.1.5.1. Back to Back Tests

Two or more versions of the same application are compared concerning their outputs using the same inputs. In this case one version is the original OM system and the other is the new OMI system. The demonstration model will be used with the standard variable and parameter values [Appendix B]. Only the outgoing flow level of the source will be changed during the simulation time.

Name	Start value	Value after 200s	Value after 400s	Value after 600s
source.flowLevel	0.02	0.04	0.08	0.16

Table 5-6 source.flowLevel values for a Back to Back Test

As depicted in Table 5-6 the outgoing liquid from the source starts at "0.02" and doubles every 200 seconds. The following plot shows the level of liquid in the first tank ("tank1.h") and the gain of the outgoing liquid from the source ("source.qOut.lflow")

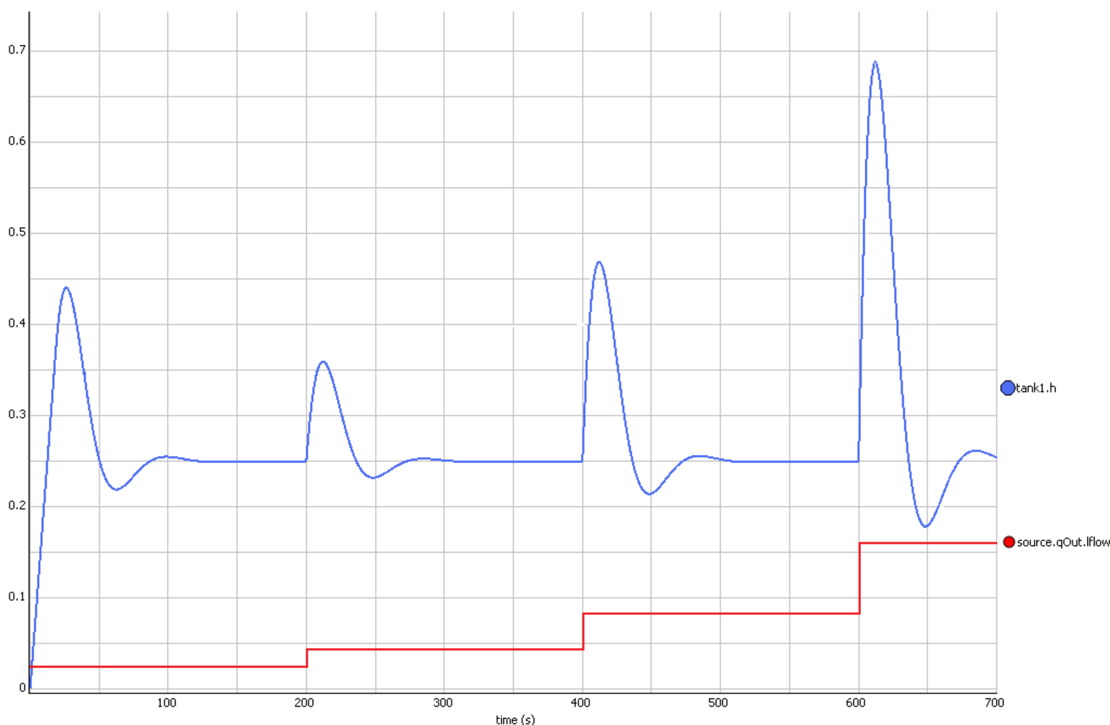

Figure 5-9 Plot of Simulation Results Tank1.h and Source.qOut.lflow

Time (s)	Iflow	OM - tank1.h	OMI - tank1.h	Deviation (absolute)	Deviation (percent)
0.0	0.02	0.000000	0.000000	0.000000	0.00%
1.0	0.02	0.020000	0.020000	0.000000	0.00%
2.0	0.02	0.040000	0.040000	0.000000	0.00%
3.0	0.02	0.060000	0.060000	0.000000	0.00%
4.0	0.02	0.070000	0.070000	0.000000	0.00%
18.0	0.02	0.360000	0.360000	0.000000	0.00%
19.0	0.02	0.376354	0.375674	**0.000680**	**-0.18%**
20.0	0.02	0.376526	0.375149	**0.001377**	**-0.37%**
92.0	0.02	0.250041	0.250041	0.000000	0.00%
131.0	0.02	0.250001	0.250001	0.000000	0.00%
132.0	0.02	0.250000	0.250000	0.000000	0.00%
198.0	0.02	0.249999	0.250000	0.000001	0.00%
199.0	0.02	0.250081	0.250000	0.000081	-0.03%
200.0	0.04	0.262371	0.262512	**0.000141**	**+0.05%**
201.0	0.04	0.266349	0.266330	0.000019	-0.01%
202.0	0.04	0.266702	0.266689	0.000013	0.00%
203.0	0.04	0.265699	0.265612	0.000087	-0.03%
389.0	0.04	0.249999	0.250000	0.000001	0.00%
399.0	0.04	0.250064	0.250000	0.000064	-0.03%
400.0	0.08	0.275022	0.275007	0.000015	-0.01%
401.0	0.08	0.282507	0.28258	0.000073	+0.03%
402.0	0.08	0.283273	0.283346	0.000073	+0.03%
403.0	0.08	0.281430	0.281512	0.000082	+0.03%
589.0	0.08	0.250000	0.250000	0.000000	0.00%
599.0	0.08	0.250430	0.250000	**0.000430**	**-0.17%**
600.0	0.16	0.30002	0.299893	**0.000127**	**-0.04%**
601.0	0.16	0.315029	0.315043	0.000014	0.00%
602.0	0.16	0.316480	0.316591	**0.000111**	**+0.04%**
603.0	0.16	0.312852	0.312944	0.000092	+0.03%

Table 5-7 Results of the Back to Back Test

The time values from 0.0s – 132.0s are selected at random. The time when "Source.qOut.Iflow" is changed and its limits are important for this Back to Back test. "OM - tank1.h" represents the results of the original OM simulation runtime. "OMI - tank1.h" represents the results of the new modified OMI simulation runtime. As depicted in Table 5-7 the deviations between "OM - tank1.h" and ""OMI - tank1.h"" are in the range of ±0.01% and ±0.05%. This is acceptable in view of the fact that the deviation will not be larger. It will be further reduced according to the number of results provided

6. Interactive Graphical User Interface

In order to demonstrate the developed interactive simulation capabilities a Graphical User Interface (GUI) has been developed. The simulation environment is implemented in Java [31]. The GUI is implemented in the Standard Widget Toolkit (SWT) [32], it is an open source widget toolkit for Java designed to provide efficient and mobile GUI development.

6.1. Simulation configuration

While translating a SysML model into Modelica code a XML file containing all model information has to be generated. This file will be needed to configure simulation data, for example, it defines which parameter should be used interactively or which attributes should be displayed on a plot.

The XML file contains the following information:

- Specific model information, for example its name, version or general comments.
- All variables with values, types and comments.
- All parameters with values, types and comments.

A simulation configuration tool reads this initial XML [Appendix C] file and offers a GUI to change values and properties on it.

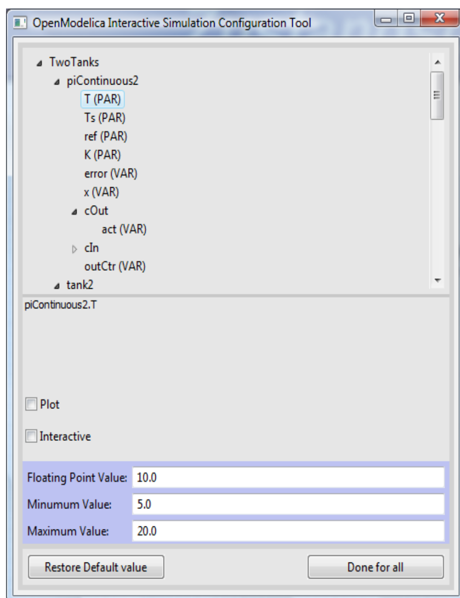

Figure 6-1 Simulation Configuration Tool

The simulation configuration tool displays all components and their attributes as a tree.

A variable is marked with a (VAR) and the user has the following options for it:

- Display the selected variable on a plot (Check the “Plot” option).

A parameter is marked with a (PAR) and the user has the following options for it:

- Display the selected parameter on a plot (Check the “Plot” option).
- Use the selected parameter interactively (Check the “Interactive” option).
- Set another start value as default. This will cause a “changevalue” message from the GUI to OMI with the new values.
- If the type is a float or integer the user can set a minimum and maximum value for it. This can be used to offer sliders in the control center.

The tool completes the information from the initial XML file with new data and generates a new configured XML file [Appendix D].

6.2. Simulation Environment

The interactive GUI depicted in Figure 5-1 communicates with the OMI simulation runtime, which runs concurrently on a computer using operations from Table 5-3. The GUI has also a “GUI::ControlServer”, a “GUI::ControlClient” and a “GUI::TransferServer” to receive result data from the “OMI::TransferClient”. The network configuration is depicted in Table 5-2.

The interactive simulation environment reads the configured XML file and generates all containers and objects containing all variables, parameters and their properties. The GUI builds a connection to the simulation runtime and initialises the runtime displayed in Figure 5-4. After initialization of the runtime a simulation control center will be displayed and offers the interactive simulation of the model using the OMI simulation runtime.

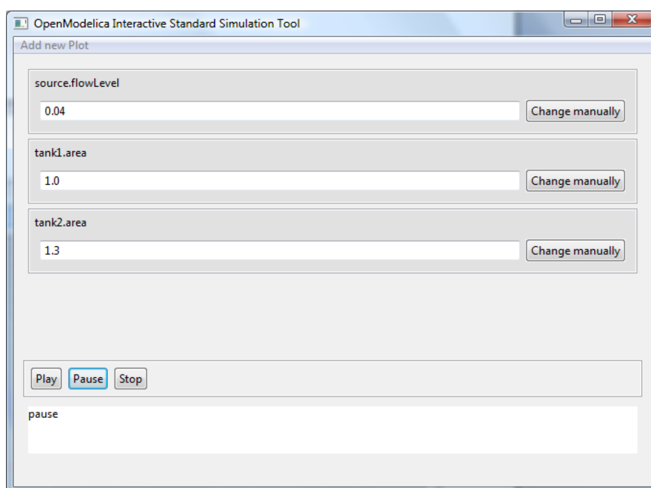

Figure 6-2 Simulation control center

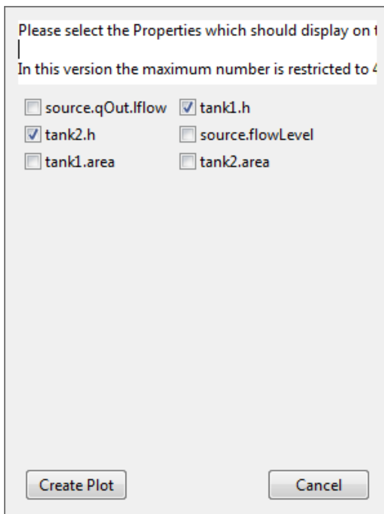

Figure 6-3 Selection of properties to display on plot

The user can display the results as graphical charts which are implemented using "JFreeChart" a freely available Java graph plotting solution [29].

After selecting "Add new Plot" in the control center a new window depicted in Figure 6-3 is shown. In this window the user can select all variables and parameters which are marked as "Plot" in the simulation configuration tool. By selecting "Create Plot" a new and empty chart plot view as displayed in Figure 6-4 will be created.

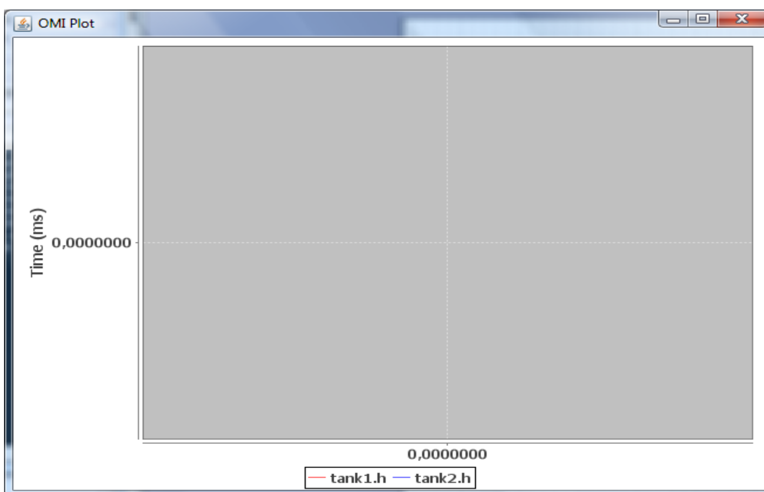

Figure 6-4 New plot to display tank1.h and tank2.h

The chart plot view communicates with the control center using the observer pattern to get new results for its properties.

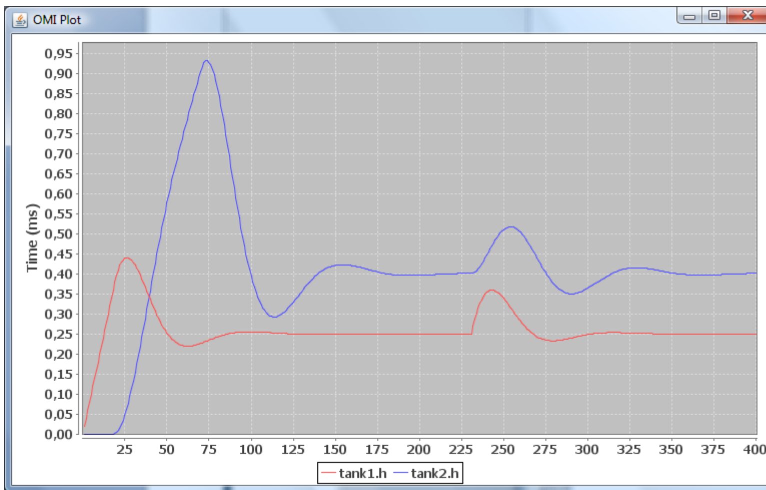

Figure 6-5 Live plot of tank1.h and tank2.h

By clicking the play button in the control center the “start#end” message from Table 5-3 will be sent to the runtime and the simulation begins. Now the user can enter a new value for a parameter and click “Change manually” to stimulate the model interactively.

7. Conclusions and Future Work

7.1. Conclusions

This work presents a system modelling and simulation approach that enables the creation of executable system models which can be simulated interactively. It proved the possibility to make SysML models executable by providing a possible mapping between the SysML and Modelica and rules for translating SysML models into Modelica code. A concrete example of a Two Tanks Systems is used to illustrate this approach.
Moreover, this thesis presents a new OpenModelica Interactive (OMI) simulation environment which enables user- interactive real-time system simulation of system time-continuous and time-discrete behaviour and provides a powerful interface for the visualization of system which is simulated. A simplified graphical user interface of a simulation centre is implemented in order to demonstrate the outcomes.

7.2. Future Work

The future work is mostly connected with mapping of SysML to Modelica, implementation of SysML diagrams and extending the simulation runtime for more simulation functionalities. Full implementation of the tasks listed below may result in a complete and stand-alone system modelling and simulation environment tool.

A general and complete mapping supporting all the concepts of SysML and Modelica is still to be elaborated. The following are some elements to be mapped completely:

- Modelica Equations and Algebraic Equations.
- Integration of the Modelica Standard Library.
- SysML State Chart Diagram.
- SysML Activity Diagram.
- SysML Requirements.
- Efficient translation rules.

An efficient and powerful simulation runtime is the most important part for an interactive real-time simulation. The following are some extensions to the OMI simulation runtime:

- A more powerful and stable solver.
- Extending of the simulation control API with additional operations.
- Extending of the communication interface.

In order to make this approach useable in operational field the following parts need to integrate as a stand-alone modelling and simulation environment tool or an Eclipse Plug-In:

- The SysML modelling environment.
- The SysML to Modelica transformation.
- The OMI simulation runtime.
- The simulation center.

The following are open tanks to integrate the parts:

- Automated Modelica code generation from SysML diagrams.
- Modelica code and diagram synchronization.
- Integration with the OpenModelica compiler (OMC).
- Modelica code or library presentation as SysML.
- Integration with Modelica Development Tooling (MDT).

This is an ongoing research project at the EADS Innovation Works in cooperation with the Linkoping University and the OMG SysML/Modelica Integration Working Group.

IV. References

[1] Sanford Friedenthal, Alan Moore and Rick Steiner, 2008, Practical Guide to SysML: The Systems Modeling Language, Morgan Kaufmann.

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

[3] Andrew S. Tanenbaum and Maarten Van Steen, 2006,Distributed Systems: Principles and Paradigms, Prentice Hall

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman, 2006, Compilers: Principles, Techniques, and Tools, Addison Wesley.

[5] André Willms, 2008, Einstieg in Visual C++ 2008, Galileo Computing.

[6] Jürgen Wolf, 2006, C++ von A bis Z, Galileo Computing.

[7] Ralf Reussner und Wilhelm Hasselbring, 2006, Handbuch der Software-Architektur, dpunkt Verlag.

[8] Andrew S. Tanenbaum, 2003, Computer Networks (4th Edition), Prentice Hall.

[9] Frieder Grupp und Florian Grupp, 2007, Simulink: Grundlagen und Beispiele, Oldenbourg.

[10] K.E. Brenan, S.L. Campbell, and L.R. Petzold, 1996, Numerical Solution of Initial Value Problems in Differential/Algebraic Equations. SIAM, second edition.

[11] Friedenthal, Sanford, Greigo, Regina, and Mark Sampson, INCOSE MBSE Roadmap, in "INCOSE Model Based Systems Engineering (MBSE) Workshop Outbrief" (Presentation Slides), presented at INCOSE International Workshop 2008, Albuquerque, NM, pg. 6, Jan. 26, 2008

[12] Modelica Association, 2005, "Modelica Language Specification Version 3.0", http://www.modelica.org/documents/ModelicaSpec30.pdf, September 5, 2007.

[13] Object Management Group UML, "OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2", http://www.omg.org/docs/formal/07-11-04.pdf, November 2007.

[14] Object Management Group UML, "OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2", http://www.omg.org/docs/formal/07-11-02.pdf, November 2007.

[15] Object Management Group SysML, "OMG Systems Modeling Language (OMG SysML™) Specification", http://www.omg.org/docs/formal/08-11-02.pdf, November 2008.

[16] PELAB, Peter Fritzson, "OpenModelica System Documentation, Version, 2008-01-27 for OpenModelica1.4.5", http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/releases/1.4.5/doc/OpenModelicaSystem.pdf, January 2009.

[17] PELAB, Peter Fritzson, "OpenModelica Users Guide, Version 2009-01-27 for OpenModelica 1.4.5", http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/releases/1.4.5/doc/OpenModelicaUsersGuide.pdf, January 2009.

[18] Computing and Mathematics Research Division Lawrence Livermore National Laboratory, Petzold, Linda R., http://www.netlib.org/ode/ddassl.f, December 12 2006.

[19] The International Council on Systems Engineering (INCOSE), Last Accessed: 2009 http://www.incose.org/

[20] Object Management Group (OMG) Systems Modelling Language, Last Accessed: 2009 http://www.omgsysml.org/

[21] Modelica and the Modelica Association, Last Accessed: 2009 http://www.modelica.org/

[22] Modelica and the Modelica Association, Modelica Libraries, Last Accessed: 2009 http://www.modelica.org/libraries

[23] Dynasim AB, Dymola, Last Accessed: 2009 http://www.dynasim.se/

[24] The OpenModelica Project, Last Accessed: 2009 http://www.ida.liu.se/~pelab/modelica/OpenModelica.html

[25] MathCore Engineering AB, MathModelica, Last Accessed: 2009 http://www.mathcore.com/products/mathmodelica/

[26] Open Source Modelica Consortium, Last Accessed: 2009 http://www.ida.liu.se/labs/pelab/modelica/OpenSourceModelicaConsortium.html

[27] Linköping University, Last Accessed: 2009 http://www.liu.se

[28] OpenModelica source code version 1.4.5 from Subversion repository, http://www.ida.liu.se/labs/pelab/modelica/OpenModelica.html#Download

[29] Object Refinery Limited, JFreeChart, Last Access: 2009 http://www.jfree.org/jfreechart/

[30] IBM Rational Rhapsody, Systems-Engineering Tool Rhapsody 7.2, http://www.telelogic.com/products/rhapsody/index.cfm

[31] Sun Microsystems, Java, http://java.sun.com/

[32] The Standard Widget Toolkit (SWT), Version 3.4.2 (13 February 2009), http://www.eclipse.org/swt/

V. Appendix

Appendix A. The TanksConnectedPI demonstration model Modelica code

package.mo

```
package TwoTanks
end TwoTanks;
```

ReadSignal.mo

```
within TwoTanks;

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

LiquidFlow.mo

```
within TwoTanks;

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

ActSignal.mo

```
within TwoTanks;

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

Tank.mo

```
within TwoTanks;

block Tank
  output ReadSignal  tSensor   "Connector, sensor reading tank level (m)";
  input  ActSignal   tActuator "Connector, actuator controlling input
flow";
  input  LiquidFlow  qIn       "Connector, flow (m3/s) through input
valve";
  output LiquidFlow  qOut      "Connector, flow (m3/s) through output
valve";
  parameter Real area(unit = "m2")       =  0.5;
  parameter Real flowGain(start = 1.99, unit = "m2/s") = 0.5;
  parameter Real minV= 0, maxV = 10;    // Limits for output valve flow
  Real           h(start = 0.0, unit = "m")   "Tank level";
 equation
  der(h)      = (qIn.lflow - qOut.lflow)/area;    // Mass balance equation
  qOut.lflow  = if (-flowGain*tActuator.act) >maxV then maxV
          else if (-flowGain*tActuator.act) <minV then minV
          else (-flowGain*tActuator.act);
  tSensor.val = h;
end Tank;
```

BaseController.mo

```
within TwoTanks;

partial block BaseController
  input  ReadSignal cIn  "Input sensor level, connector";
  output ActSignal  cOut "Control to actuator, connector";
  parameter Real Ts(unit = "s") = 0.1;
  parameter Real K = 2 "Gain";
  parameter Real T(unit = "s") = 10 "Time constant";
  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;
```

PIcontinuousController.mo

```
within TwoTanks;

block 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;
```

LiquidSource.mo

```
within TwoTanks;

block LiquidSource
  output LiquidFlow qOut;
  parameter Real flowLevel = 0.02;
equation
  qOut.lflow = flowLevel;
end LiquidSource;
```

TanksConnectedPI.mo

```
within TwoTanks;

block 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(piContinuous1.cOut, tank1.tActuator);
  connect(tank1.tSensor, piContinuous1.cIn);
  connect(tank1.qOut, tank2.qIn);
  connect(piContinuous2.cOut, tank2.tActuator);
  connect(tank2.tSensor, piContinuous2.cIn);
end TanksConnectedPI;
```

Appendix B. Standard parameter and variable values

Name	Type	Value
tank1.h	Variable	0.0
tank2.h	Variable	0.0
piContinuous1.x	Variable	0.0
piContinuous2.x	Variable	0.0
tank1.tActuator.act	Variable	0.0
tank1.qIn.lflow	Variable	0.0
tank2.tActuator.act	Variable	0.0
tank2.qIn.lflow	Variable	0.0
tank2.qOut.lflow	Variable	0.0
piContinuous1.error	Variable	0.0
piContinuous2.error	Variable	0.0
source.qOut.lflow	Variable	0.0
tank1.tSensor.val	Variable	0.0
tank1.qOut.lflow	Variable	0.0
tank2.tSensor.val	Variable	0.0
piContinuous1.cIn.val	Variable	0.0
piContinuous1.cOut.act	Variable	0.0
piContinuous1.outCtr	Variable	0.0
piContinuous2.cIn.val	Variable	0.0
piContinuous2.cOut.act	Variable	0.0
piContinuous2.outCtr	Variable	0.0
source.flowLevel	Parameter	0.02
tank1.area	Parameter	1.0
tank1.flowGain	Parameter	0.5
tank1.minV	Parameter	0.0
tank1.maxV	Parameter	10.0
tank2.area	Parameter	1.3
tank2.flowGain	Parameter	0.5
tank2.minV	Parameter	0.0
tank2.maxV	Parameter	10.0

piContinuous1.Ts	Parameter	0.1
piContinuous1.K	Parameter	2.0
piContinuous1.T	Parameter	10.0
piContinuous1.ref	Parameter	0.25
piContinuous2.Ts	Parameter	0.1
piContinuous2.K	Parameter	2.0
piContinuous2.T	Parameter	10.0
piContinuous2.ref	Parameter	0.4

Appendix C. Structure example of the „TwoTanks_Init.xml”

```
- <model>
- <general>
  <name>TwoTanks</name> - <!-- Project Name -->
  <version>1.0</version> - <!-- Version of project -->
  <id /> - <!-- A unique identifier for a ModelConfiguration.xml and a corresponding
    ModelSimulation.exe, the Model data initialization tool generates the ID -->
  <n_states>4</n_states>
  <n_algebraics>17</n_algebraics>
  <n_parameters>17</n_parameters>
  <n_string_parameters>0</n_string_parameters>
  <n_string_variables>0</n_string_variables>
  <comment>This TwoTanks model is a demonstration model for the bachelor thesis by
    Parham Vasaiely</comment>
    </general>
- <mainclass name="main"> - <!-- Main Class -->
- <general>
  <type>TanksConnectedPI</type>
  <comment>no comment yet</comment>
    </general>
- <variables>
- <var name="tank1.h">
- <general>
  <type>Real</type> - <!-- Datatype -->
  <comment>no comment yet</comment>
```

```
<plot>false</plot>
  </general>
<value>0.0</value>
  </var>
…
- <parameters>
- <par name="source.flowLevel">
- <general>
<type>Real</type> - <!-- Datatype -->
<comment>no comment yet</comment>
<plot>false</plot>
<interactive>false</interactive>
  </general>
<value>0.02</value>
<minValue />
<maxValue />
  </par>
      …
```


Appendix D. Structure example of the „TwoTanks_SimulationConfig.xml“


```
- <model>
- <general>
<name>TwoTanks</name> - <!-- Project Name -->
<version>1.0</version> - <!-- Version of project -->
<id>4711</id> - <!-- A unique identifier for a ModelConfiguration.xml and a corresponding
  ModelSimulation.exe, the Model data initialization tool generates the ID -->
<n_states>4</n_states>
<n_algebraics>17</n_algebraics>
<n_parameters>17</n_parameters>
<n_string_parameters>0</n_string_parameters>
<n_string_variables>0</n_string_variables>
<comment>This TwoTanks model is a demonstration model for the bachelor thesis of
  Parham Vasaiely</comment>
  </general>
- <mainclass name="main">- <!-- Main Class -->
- <general>
<type>TanksConnectedPI</type>
```

```
<comment>no comment yet</comment>
  </general>
- <variables>
- <var name="tank1.h">
- <general>
<type>Real</type> - <!-- Datatype -->
<comment>no comment yet</comment>
<plot>true</plot>
  </general>
<value>0.0</value>
  </var>
- <parameters>
- <par name="source.flowLevel">
- <general>
<type>Real</type> - <!-- Datatype -->
<comment>no comment yet</comment>
<plot>true</plot>
<interactive>true</interactive>
  </general>
<value>0.02</value>
<minValue>0.01</minValue>
<maxValue>1.0</maxValue>
  </par>
```

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 24.08.2009

Ort, Datum Unterschrift