Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Master Thesis

Parham Vasaiely

Model-Based Design, Verification and Validation of Systems using SysML and Modelica

EADS Innovation Works UK

Faculty of Engineering and Computer Science
Department of Computer Science

Fakultät Techchnik und Informatik
Department Informatik

Parham Vasaiely

Model-Based Design, Verification and Validation of Systems using SysML and Modelica

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Bettina Buth
Zweitgutachter : Prof. Dr. Zhen Ru Dai

Abgegeben am 24.08.2011

Parham Vasaiely

Title of the thesis

Model-Based Design, Verification and Validation of Systems using SysML and Modelica

Keywords

SysML, Modelica, Simulation, System, Model-based Engineering, Systems Engineering, Verification, Validation, Test, Test Framework, TTCN-3, UTP

Abstract

The increasing complexity of modern technical systems is challenging the system engineering domain extremely. Therefore the International Council on Systems Engineering (INCOSE) identified the Model-Based Systems Engineering (MBSE) as the key technology for successful systems engineering in the future. This work is a step towards the MBSE paradigm. The approach presented in this work is to combine the descriptive power of SysML and the simulation capabilities of Modelica, to develop an executable design model, for early prototyping, verification and validation of systems. A test specification and implementation language will be developed in Modelica, based on the standardized test specification language TTCN-3. It can be used as a test model framework. As a second objective, this work is based on well defined standard technologies. The reuse of existing technologies will not only support their global understanding and dissemination, but also reduce time and cost of developing and of finding acceptance by developers in projects.

Parham Vasaiely

Thema der Masterarbeit

Model-basiertes Design, Verifikation und Validierung von Systemen unter Verwendung der SysML und Modelica

Stichworte

SysML, Modelica, Simulation, System, modell-basierte Entwicklung, Systementwicklung, Verifikation, Validierung, Test, Test Framework, TTCN-3, UTP

Kurzzusammenfassung

Die steigende Komplexität von modernen, technischen Systemen stellt eine extreme Herausforderung für die Systementwicklung dar. Deshalb hat das International Council on Systems Engineering (INCOSE) die model-basierte Systementwicklung (MBSE) als die Schlüsseltechnologie für die erfolgreiche Entwicklung von Systemen in der Zukunft identifiziert. Diese Arbeit kombiniert die Modellierungssprache SysML und die Simulationssprache Modelica, um ein ausführbares Design Model zu entwickeln, welches für erste Prototypen und der Verifikation und Validierung von Systemen verwendet werden kann. Eine Test Spezifikations- und Implementierungssprache in Modelica wurde auf der Basis von TTCN-3 Konzepten entwickelt. Diese kann als Test Model Framework verwendet werden. Des Weiteren werden in diesem Ansatz ausschließlich standardisierte Technologien verwendet. Die Verwendung von Standards reduziert nicht nur die Projektzeit und Kosten, sondern hilft deren Verständnis zu erhöhen und steigert damit die Chancen dieses Ansatzes in Projekten akzeptiert zu werden.

Acknowledgments

First of all I must thank my family because of their support and love. Especially my mom, Soudabeh has always believed in me. Her optimism and positivity is my moving spirit. Over all these years, she has been my most important source of support.

I must thank EADS Germany as well as EADS UK, for their very professional support. Joining the EADS was surely the best decision I ever made, since this company is responsible for many important inventions of our modern world and is pioneer in so many aspects.

Finally, I am grateful for the tools and academic software licenses provided by Microsoft, IBM, the Eclipse Foundation and OpenModelica.

Table of Contents

I. List of Figures ... VII

II. List of Tables ... IX

III. Glossary ... XI

1. Introduction ... 1

1.1. Background ... 1

1.2. Objective ... 2

1.3. Thesis Structure ... 4

2. State of the Art ... 5

2.1. Modelica ... 5

2.1.1. The Modelica application area ... 5

2.1.2. OpenModelica ... 6

2.2. SysML ... 6

2.2.1. IBM Rational Rhapsody ... 7

2.3. Verification and Validation ... 8

2.3.1. The Testing and Test Control Notation version 3 (TTCN-3) ... 9

2.3.2. V-Model XT ... 9

2.3.3. Black-Box Testing ... 10

3. Demonstration System ... 12

3.1. The Aircraft Water Tank System ... 12

3.2. Possible System and Liquid Tank States ... 13

3.2.1. System states represented in the SysML State Machine Diagram ... 15

3.3. Use Case: Filling Tanks of the Aircraft Water Tank System ... 16

3.3.1. Use Case Specification as Text ... 16

3.3.2. Use Case Specification as SysML Diagram ... 17

4. System Design with SysML and Modelica ... 18

4.1. Transformation Approach between SysML and Modelica ... 19

4.1.1. SysML and Modelica Mapping of Language Elements ... 20

4.1.2. Additional Modelica Syntax as SysML Stereotypes ... 24

4.1.3. SysML Parametric to Modelica Equation ... 26

4.2. SysML to Modelica Transformation Application using the Example Model ... 28

4.2.1. System structure with SysML Package Diagram ... 28

4.2.2. System structure with SysML Block Definition Diagram ... 29

4.2.3. System structure with SysML Internal Block Diagram ... 36

4.2.4. Block Definition Diagrams of the constraint blocks ... 38

4.2.5. Parametric Diagrams of the parametric structure ... 41

4.3. Automated Modelica Code Generation ... 45

5. Modelica as Test Specification and Implementation Language 46
5.1. Modelica and TTCN-3 46
5.2. TTCN-3 Core Language Definitions and Concepts 47
5.2.1. TTCN-3 Built-In Data Types and Values 47
5.2.2. TTCN-3 Test Structure Definitions 49
5.2.3. TTCN-3 Test Behaviour Definitions 50
5.3. Modelica4Testing: A Test Model Framework for Modelica 51
5.3.1. Modelica4Testing Test Structure Definitions 53
5.3.2. Modelica4Testing Test Behaviour 58
5.3.3. Modeica4Testing Verdict Type and Values 62
5.3.4. The Verdict Mechanism 62
5.4. Modelica4Testing Meta-Model and Scope 65
5.5. The Modelica4Testing Test Model Framework 67
5.5.1. Modelica4Testing_Library 67
5.5.2. Modelica4Testing Verification and Validation Model Skeletons 68
6. Application Example of System Level Testing 71
6.1. OpenModelica as Simulation and Test Execution Tool 72
6.2. Test Case Specification 73
6.2.1. Test Case Specification as Text 73
6.2.2. Test Case implemented in Modelica4Testing 74
6.3. System Simulation and Test Execution 75
6.3.1. System under Test Preparation 75
6.3.2. Test Execution and Evaluation Process 76
7. Conclusions and Future Work 79
7.1. Conclusions 79
7.2. Future Work 80
IV. References XI
V. Appendix XVI
Appendix A. Modelica design model code of Aircraft Water Tank System XVI
Appendix B. Test model code developed in Modelica4Testing XIX
Appendix C. Approach of mapping UTP to Modelica4Testing XXIII
a. UTP to Modelica4Testing Test Structure XXV
b. UTP to Modelica4Testing Test Behaviour XXV
c. UTP to Modelica4Testing Test Data and Time Mechanism XXVII

I. List of Figures

Figure 1-1 High Level Approach of the Model-Based Design, Verification & Validation Process...... 3
Figure 2-1 OMG Illustration of the Relationship between SysML 1.2 and UML2 6
Figure 2-2 OMG Illustration of the SysML Diagram Types 7
Figure 2-3 V-Model XT.. 10
Figure 2-4 Black-Box Testing Environment and Components.. 11
Figure 3-1 Aircraft water tanks with continuous controllers connected together............................. 13
Figure 3-2 Possible states of tanks during the filling process.. 13
Figure 3-3 System states represented as a SysML state machine 15
Figure 3-4 SysML Use Case Diagram: Filling up Tanks with Fresh Water from a Tanker 17
Figure 4-1 Level of Abstraction in Systems- Engineering using the V-Modell XT............................ 18
Figure 4-2 SysML and Modelica transformation approach 19
Figure 4-3 List of Stereotypes in Rhapsody 26
Figure 4-4 Aircraft Water Tank System Package Structure using SysML Package Diagrams........ 28
Figure 4-5 ReadSignal FlowSpecification 29
Figure 4-6 ActuatorSignal FlowSpecification 30
Figure 4-7 LiquidFlow FlowSpecification... 30
Figure 4-8 ModeSignal FlowSpecification.. 30
Figure 4-9 StatusSignal FlowSpecification... 31
Figure 4-10 FlowLevelSignal FlowSpecification.. 31
Figure 4-11 BaseController Block... 31
Figure 4-12 PIDcontinuousController Block 32
Figure 4-13 PIcontinuousController Block.. 32
Figure 4-14 LiquidTank Block... 34
Figure 4-15 ControlUnit Block... 34
Figure 4-16 AircraftWaterTankSystem Block 35
Figure 4-17 BBD Aircraft Water Tank System with PID continuous controllers 35
Figure 4-18 IBD Aircraft Water Tank System using PID 36
Figure 4-19 BDD Constraint Blocks of Control Unit 38
Figure 4-20 BDD Constraint Blocks of Liquid Tank... 39
Figure 4-21 BDD Constraint Blocks of BaseController 40
Figure 4-22 BDD Constraint Blocks of PID Continuous Controller.. 40
Figure 4-23 PAR Liquid Tank 41
Figure 4-24 PAR Control Unit... 43
Figure 4-25 PAR BaseControler and PIDcontinuousController 44
Figure 4-26 Automated Code Generation as a Key Feature for the Application in Practice............ 45

Figure 5-1 JUnit and TTCN-3 in relation to the V-Model XT....................46
Figure 5-2 Modelica and Modelica4Testing Modelling Layers....................51
Figure 5-3 MathModelica Simple Modelica Meta-Model....................52
Figure 5-4 Modelica4Testing Overview of SUT and its Components....................55
Figure 5-5 Modelica4Testing Overview of Test System Configuration....................57
Figure 5-6 Modelica4Testing Overview of Test Case and its Stimulation and Observation Parts...59
Figure 5-7 Modelica4Testing Verdict Mechanism to assign a local test case verdict....................63
Figure 5-8 Modelica4Testing Assigning Overall Verdict: Single Test Case....................64
Figure 5-9 Modelica4Testing Assigning Overall Verdict: Test Suite with independent Test Cases 65
Figure 5-10 Modelica4Testing Assigning Overall Verdict: Test Suite with dependent Test Cases .65
Figure 5-11 Modelica4Testing Meta-Model: Test Model Framework for Modelica....................66
Figure 5-12 Modelica4Testing Hierarchical Scope....................67
Figure 6-1 Black-Box view of the Aircraft Water Tank System as the SUT....................71
Figure 6-2 Executing the Test Model and the SUT using the Modelica Tool OpenModelica....................72
Figure 6-3 Eclipse based Simulation and Test Environment for OM: Management View....................72
Figure 6-4 Eclipse based Simulation and Test Environment for OM: Result View....................73
Figure 6-5 Copy of AWTS State Machine described in section 3.2.1....................73
Figure 6-6 Simulation Results for a correct Model, Test Case is Pass....................77
Figure 6-7 Simulation Results for a incorrect Model, the Test Case is Fail....................77
Figure 6-8 Simulation Results for the first test session in detail....................78
Figure 6-9 Simulation Results for the second test session in detail....................78
Figure 7-1 Illustration of the test model used in the M4T approach....................79
Figure 7-2 Including the UML2 Testing Profile as Future Work....................80
Figure V-1 OMG illustreation of the UTP Meta-Model (Test Architecture and Behavior)....................XXIV

II. List of Tables

Table 1 SysML Package → Modelica Package 20
Table 2 SysML Block → Modelica Block 21
Table 3 SysML Attribute → Modelica Variable 21
Table 4 SysML Part → Modelica Variable (Part) 21
Table 5 SysML Association (Part) → Modelica Syntax Element 21
Table 6 SysML FlowSpecification → Modelica Connector 22
Table 7 SysML Flow Port Node → Modelica Instance of Connector 22
Table 8 SysML Atomic Flow Port Node → Modelica Instance of Connector 22
Table 9 SysML Connector → Modelica Connection 22
Table 10 SysML Flow (FlowDirection) → Modelica Causality of connector instance 23
Table 11 SysML Inheritance (Gen/Spec) → Modelica extends 23
Table 12 SysML Datatype Double → Modelica Datatype Real 23
Table 13 SysML Description → Modelica Comment 23
Table 14 SysML Stereotype <<abstract>> for Modelica partial 24
Table 15 SysML Stereotype <<variable>> for Modelica variability and unit 25
Table 16 SysML Stereotype <<extendsRelation>> Modelica modification of inherit variable value 25
Table 17 SysML Stereotype <<composite>> for Modelica instance modification 26
Table 18 SysML Parametric elements 27
Table 19 TTCN-3 Basic Type Integer 47
Table 20 TTCN-3 Basic Type Float 47
Table 21 TTCN-3 Basic Type Boolean 48
Table 22 TTCN-3 Basic Type Verdicttype 48
Table 23 TTCN-3 Structured Type Record 48
Table 24 TTCN-3 Structure Definition: Module 49
Table 25 TTCN-3 Structure Definition: System under Test (SUT) 49
Table 26 TTCN-3 Structure Definition: Main Test Component 49
Table 27 TTCN-3 Structure Definition: Parallel Test Component 49
Table 28 TTCN-3 Structure Definition: Test System Interface 49
Table 29 TTCN-3 Structure Definition: Test Configuration 49
Table 30 TTCN-3 Structure Definition: Part Mappings 50
Table 31 TTCN-3 Structure Definition: Port Connections 50
Table 32 TTCN-3 Behaviour Definition: Test Case 50
Table 33 TTCN-3 Behaviour Definition: Module Control Part 50
Table 34 TTCN-3 Behaviour Definition: Test Execution 50
Table 35 TTCN-3 Behaviour Definition: Test Verdict (local and overall) 51

Table 36 Modelica4Testing Test Model 53
Table 37 Modelica4Testing Test Context...53
Table 38 Modelica4Testing Design Model 54
Table 39 Modelica4Testing Test System Interface 54
Table 40 Modelica4Testing SUT 55
Table 41 Modelica4Testing Test System 55
Table 42 Modelica4Testing Test Component 56
Table 43 Modelica4Testing MTC...56
Table 44 Modelica4Testing PTC 57
Table 45 Modelica4Testing Test Configuration...57
Table 46 Modelica4Testing Port Connection 58
Table 47 Modelica4Testing Test Case..58
Table 48 Modelica4Testing Test Case Stimulator 59
Table 49 Modelica4Testing Test Case Stimulator Function 60
Table 50 Modelica4Testing Test Case Evaluator 60
Table 51 Modelica4Testing Test Case Evaluator Function 61
Table 52 Modelica4Testing Test Case Evaluator Model 62
Table 53 Modelica4Testing Verdicttype 62
Table 54 Modelica4Testing Verdicttype possible Values...62
Table 55 Modelica4Testing Test Case Verdict Precondition 63
Table 56 Modelica4Testing Test Case Verdict Post-Condition 63
Table 57 Modelica4Testing Test Case Verdict SUT Behaviour..63
Table 58 Modelica4Testing Overwriting Rules for the Verdict..64
Table 59 Modelica4Testing Verdict Mechanism element order and implementation in Modelica ...64
Table 60 Test Case implementation in Modelica4Testing, Stimulator Part...................................74
Table 61 Test Case implementation in Modelica4Testing, Evaluator Part....................................75
Table 62 Liquid Tank with correct implemented tank mode equation 76
Table 63 Liquid Tank with incorrect implemented tank mode equation..76
Table 64 Attributes as graphs in the simulation result plot 76
Table 65 UTP to Modelica4Testing Test Structure XXV
Table 66 UTP to Modelica4Testing Test Behaviour XXVII
Table 67 UTP to Modelica4Testing Test Data and Time Mechanism XXVII

III. Glossary

AWTS	Aircraft Water Tank System
BBD	Block Definition Diagram
DASSL	Differential/Algebraic System Solver
EADS	European Aeronautic Defence and Space Company
EMF	Eclipse Modeling Framework
ETSI	European Telecommunications Standards Institute
FSM	Finite State Machine
GUI	Graphical User Interface
IBD	Internal Block Diagram
IEEE	The Institute of Electrical and Electronics Engineers
INCOSE	International Council on Systems Engineering
IS	Interactive Simulation
M4T	Modelica4Testing
MBSE	Model-Based Systems Engineering
MTC	Main Test Component
OM	OpenModelica
OMC	OpenModelica Compiler
OMG	Object Management Group
OMI	OpenModelica Interactive
OSMC	Open Source Modelica Consortium
PAR	Parametric Diagram
PID	proportional–integral–derivative
PoC	Point of Control
PoO	Point of Observation
PTC	Parallel Test Component
SM	State Machine
SRS	System Requirements Specification
SUT	System under Test
SysML	Systems Modelling Language
TC	Test Case
TSI	Test System Interface
TTCN-3	The Testing and Test Control Notation version 3
TTCN-3 CL	TTCN-3 Core Language
UC	Use Case
UI	User Interface
UP	Unified Software Development Process
UTP	UML2 Testing Profile
VVT	Verification, Validation and Test
XML	Extensible Markup Language

1. Introduction

1.1. Background

The increasing complexity of modern technical systems, for example used in the aerospace or defence industry, and the fact that these systems are designed and manufactured in distributed, collaborative engineering teams, issue a challenge to the systems engineering discipline. The higher level of complexity is coupled to the increasing number of involved technologies, the increasing degree of automation, and the fact that software and hardware components are strongly coupled. For example the Airbus A380-800, as one of the largest passenger aircrafts in the world, has about 4.5 million components and parts distributed over 73 m of length. Not only is the aircraft built in a distributive manner in France, Germany, Spain, and the United Kingdom, it also has many suppliers involved in the development process.

The International Council on Systems Engineering (INCOSE) [26] identified the Model-Based Systems Engineering (MBSE) [7] as the key technology for successful system engineering in the future. But model-based engineering was already in use, as a standard method of engineering, in many areas. However, each domain does have its own representation of domain specific data. For example, the control system and electrical engineering disciplines are using block diagrams, representing special elements like resistors and conductors and their relations [33], or the software engineering discipline which is using the Unified Modelling Language (UML) [28] to describe software components and their operations graphically. All these models are very efficient and useful when communicating a problem within the same domain, but when trying to communicate with engineers from other disciplines, these representations are getting useless to communicate a problem. So there was the need for a standardized notation. This notation should describe system requirements and system design at any level of abstraction. In 2007, the Object Management Group [27], as the leading consortium aimed at setting standards for model-based engineering, announced the availability of such a language. The Systems Modelling Language (SysML) [29] was developed in order to support communication among the domains involved in the same engineering process by defining a standardized graphical modelling language.

Another main benefit of the MBSE approach is the simulation of systems. The simulation of system models in term of early prototyping or verification and validation, is one of its most powerful features [2]. Since SysML does not describe an action language it is up to a SysML modelling tool to provide the translation to an executable programming language, in order to make SysML models executable. For example, the IBM SysML tool Rational Rhapsody provides code generation from SysML models and enables simple interactive simulation of models. But default programming language are not dedicated to provide continues simulation of complex equation systems. In turn, Modelica [23] is a well-defined object oriented modelling language which is dedicated to the simulation of physical systems. Putting together SysML and Modelica gives a powerful combination for modelling and simulation of complex systems at any stage of system development.

The application of this combination in term of early prototyping using simulation has been proved in the work [8]. The work presents an approach of translating a SysML model into Modelica, to be simulated non-interactively as well as interactively. However a specification of using SysML and Modelica in the term of verification and validation is missing. Many technologies and methods to be used in the term of verification and validation of systems do already exist and have been standardized, so there is no need to develop new ones as long as the existing methods have not been used exhaustively. These standards are defined by IT standardization organizations like the Institute of Electrical and Electronic Engineers (IEEE) [30] or the Object Management Group (OMG) [27]. Since these standards are designated for re-use, they are specified for a more general term rather than a domain specific problem.

1.2. Objective

This work will support the application of the model-based systems engineering (MBSE) paradigm in the subareas of model-based design as well as model-based verification and validation. It combines the descriptive power of OMG SysML with the simulation power of Modelica. The approach developed in this work enables the creation of executable system design models, in order to confirm a system model and a prototype against the system specification requirements. The verification and validation process is done by test simulation. Based on various definitions available in the literature, we define test simulation as the process of designing and creating a computerised model of a system for the purpose of conducting various tests in order to evaluate the behaviour of the corresponding real system under a given set of conditions. These tests will be executed using a specially developed Modelica based test specification and implementation language, which has adopted TTCN-3 Core Language concepts for testing. The application of this approach will focus on technical systems, used in the aerospace or defence industry. The verification and validation approach aims the functional system design level of a system [9].

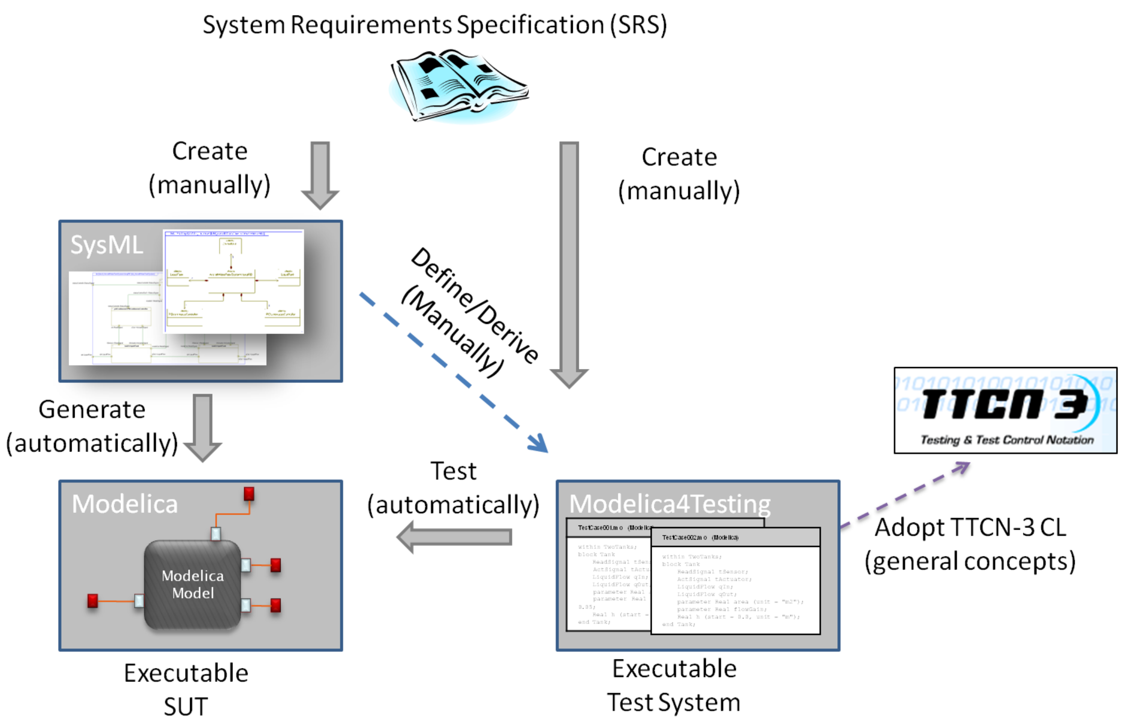


Figure 1-1 High Level Approach of the Model-Based Design, Verification & Validation Process

The Figure 1-1 illustrates the whole approach to make the concepts more visual. The developed Model-Based Design, Verification & Validation process, presented in Figure 1-1, can be divided into the following tasks:

1. Create a system design in SysML using the system requirements specification (SRS).
2. Generate Modelica code in order to have an executable design model.
3. Define or derive a system test model in the Modelica4Testing Language using the SRS and the Requirements and Behavior diagrams of the SysML design model, in parallel to the first task. This test model can be used as an executable test model.
4. Execute the test model in order to confirm the implemented prototype against the SRS using functional test methods like black-box testing.

1.3. Thesis Structure

This work can be divided into four logical parts containing different chapters and a set of appendices as described below:

Part I: Introduction and Project Overview

Chapter 1. Introduction starts by describing the system engineering discipline and its challenges. It represents the purpose, the content and the structure of the paper.

Chapter 2. "State of the Art" is a short introduction to the technologies and languages used in the context of model-based design, verification and validation of technical systems.

Chapter 3. "Demonstration System" represents the used demonstration system and its components. Also testable, respectively simulative requirements and use cases of the example system will be presented in SysML.

Part II: Model-Based System Design

Chapter 4. "System Design with SysML and Modelica" represents the system design in SysML and includes the translation from SysML into Modelica, in order make the design model executable. This approach will be applied by translating the demonstration model.

Part III: System Verification and Validation

Chapter 5. "Modelica as Test Specification and Implementation Language" presents an approach to use Modelica as a test execution language. In addition a test model framework will be presented. This model framework adopts standardized TTCN-3 Core Language concepts to Modelica, in order to make the test specification more acceptable.

Part IV: Application Example and Conclusion

Chapter 6. "Application Example of System Level Testing" shall represent the developed approaches to be applied in the verification and validation of the example system. This includes dynamic testing of the functional requirements with black-box tests.

Chapter 7. "Conclusions and Future Work" summarizes the paper and provides future work directions.

Appendix A: "Modelica design model code of Aircraft Water Tank System" includes the entire Aircraft Water Tank System as a design model, represented in Modelica code.

Appendix B: "Test model code developed in Modelica4Testing" includes the entire Aircraft Water Tank System test model, represented in Modelica4Testing.

Appendix C: "Approach of mapping UTP to Modelica4Testing" represents a mapping approach to translate UML2 Testing Profile structure and architecture elements into Modelica4Testing elements. This approach can be used to derive a test model from UTP.

2. State of the Art

The purpose of this chapter is to list the state of the art applications, technologies, methods and languages which are used in the context of model-based design, verification and validation in the systems engineering discipline. This is done by presenting the methods, techniques and technologies and afterward setting them in context.

2.1. Modelica

The following shall give a short introduction to the Modelica language, its features and some application area examples. In addition, an open source Modelica simulation environment will be introduced. This environment offers non-interactive as well as interactive real-time simulation.

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 [2].
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 assigned 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. The final language specification was submitted in 1999. The Modelica Association was founded for further development and promotion of Modelica which is an open source language [32]

In addition to programming Modelica code, graphical modelling capabilities are given using the Modelica Standard Library [33]. To do so components from the standard Modelica library can be used or especially constructed domain specific components.

2.1.1. **The Modelica application area**

Modelica 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. **OpenModelica**

Since this work uses Modelica as a textual based programming language there is a need for a compiler and a simulation runtime to execute the code.
There are several modelling and simulation environments on the market, which offer a code and component based modelling as well as the simulation of a created model. The most popular candidates are Dymola [34], MathModelica [35] and OpenModelica latter is the only open source and non-commercial tool on the market.

OpenModelica (OM) is developed and supported by the Linköping University [38] and the Open Source Modelica Consortium (OSMC) [37]. 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. The OpenModelica environment is a collection of tools, OpenModelica Tools. 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 [10] or “OpenModelica System Structure” [11].

2.2. SysML

The Systems Modeling Language (SysML) [29] is a general-purpose graphical modelling language for the Systems-Engineering domain. It is used for specifying, analyzing, designing, and verifying complex systems. The language provides graphical representations with a semantic foundation for modelling system requirements, behaviour, structure, and parametric, which is used to integrate with other engineering analysis models.

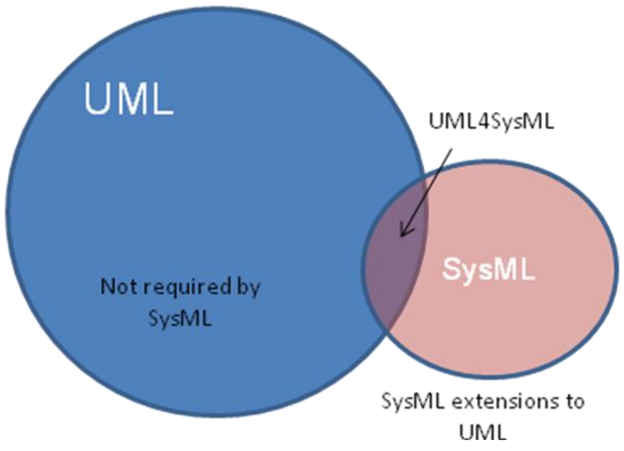


Figure 2-1 OMG Illustration of the Relationship between SysML 1.2 and UML2

The SysML is a profile of the Unified Modeling Language 2 (UML) [1] and represents a subset of extensions needed to satisfy the requirements of the UML for Systems Engineering. The Figure

2-1 presents a subset of UML which is not used by SysML, a subset of UML which is used without extensions (UML4SysML) and a set of elements which are only available in the SysML 1.2 profile.

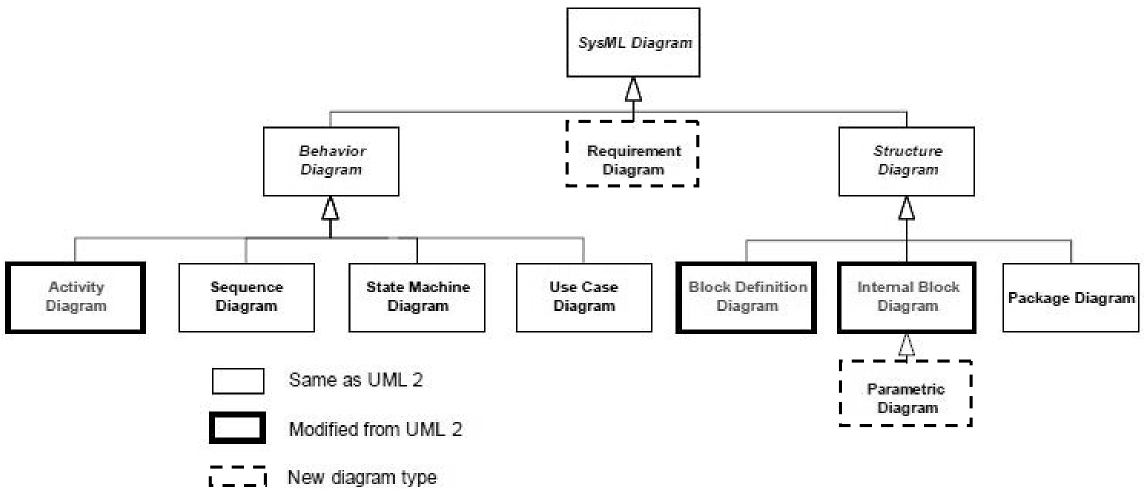


Figure 2-2 OMG Illustration of the SysML Diagram Types

The taxonomy of SysML diagrams is presented in Figure 2-2 OMG Illustration of the SysML Diagram Types.

The following are the major extensions of SysML Diagrams compared to UML Diagrams [1]:

- 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.
- The 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 [29] or “A Practical Guide to SysML” by Sanford Friedenthal, Alan Moore and Rick Steiner [1].

2.2.1. **IBM Rational Rhapsody**

IBM Rational Rhapsody is part of the IBM Rational tools family, IBMs successful effort to provide collaborative design and development for systems engineers and software developers [39]. Rational Rhapsody supports users to create real-time or embedded systems and software using

modelling and simulation techniques. Rational Rhapsody 7.6 is using industry standard languages for example UML, SysML, AUTOSAR [40] and DoDAF [41]. It can be used to validate functionality early in development. Rational Rhapsody does have its own SysML Profile implementation which can differ in some particular ways from the OMG SysML 1.2 specification [12].
In the following the "IBM Rational Rhapsody 7.6" will be designated as "Rhapsody".

2.3. Verification and Validation

The following section should introduce the general verification and validation (V&V) process and its definition in the systems engineering process, as well as the test process. In addition a test specification languages and a system development model will be introduced.

The different fields of engineering do have different definitions for Verification, Validation and Test. For example the Institute of Electrical and Electronic Engineers (IEEE) [30], the world's largest professional association for the advancement of technology, defines verification, validation and test (VVT) for hardware and software systems as follows (IEEE – 610 [13]):

- **Verification** is the process of evaluating a system or component, to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
- **Validation** is the process of evaluating a system or component during or at the end of the development process, to determine whether it satisfies specified requirements.
- **Testing** is an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.

In the context of modelling there are also variations of definitions. Balci, a noted researcher in the Modelling and Simulation field, extended the Department of Defence definition for V&V and Test as follows [24]:

- **Model verification** is substantiating that the model is transformed from one form into another, as intended, with sufficient accuracy. Model verification deals with building the model correctly. The accuracy of transforming a problem formulation into a model specification or the accuracy of converting a model representation from a micro flowchart form into an executable computer program is evaluated in model verification.
- **Model validation** substantiates that the model, within its domain of applicability, behaves with satisfactory accuracy, consistent with the M & S objectives. Model validation deals with building an accurate model. An activity of accuracy assignment can be labelled as verification or validation based on an answer to the following question: In assigning the

accuracy, “Does the model’s behaviour compare well to the corresponding system behaviour?” Even if the answer to the question of accuracy is “yes” that does not answer the question of whether the model is the right one.

- **Model testing** is determining whether inaccuracies or errors exist in the model. In model testing, the model is subjected to test data or test cases to determine if it functions properly. Test failure implies the failure of the model, not the test. A test is devised, and testing is conducted to perform either validation or verification or both. Some tests are designed to evaluate the behavioural accuracy or validity of the model, and some other tests are intended to determine the accuracy of model transformation from one domain into another (verification).

In this work some additional definitions for testing are used from the IEEE Standards 829 [14] and 1012 [15], as follow:

- **Test Run** is the execution of a model between a defined interval (t_0 – t_n) using start and stop or in real time.
- **Test Simulation** is defined as the process of designing and creating a computerised model of a system for the purpose of conducting various tests in order to evaluate the behaviour of the corresponding real system under a given set of conditions.

2.3.1. **The Testing and Test Control Notation version 3 (TTCN-3)**

The Testing and Test Control Notation version 3 (TTCN-3) is an international standardized language, having its roots in the area of testing hardware and software components of IT and telecommunications systems [16]. The international standard has been developed by the European Telecommunications Standards Institute (ETSI). TTCN-3 is a test specification and implementation language to define test procedures for black-box testing of distributed systems.

2.3.2. **V-Model XT**

The V-Modell is a process model for planning and executing systems engineering projects [9]. The V-Modell improves project transparency, project management and the probability of success by defining concrete practices with associated results and responsible Roles. The V-Modell XT is a further development of the V-Modell 97. In the following the "V-Modell XT" will be designated as "V-Modell". The V-Modell is designed as guidance for planning and executing development projects, taking into account the entire system life cycle. It defines the results to be achieved in a project and describes the actual approaches for developing these results. In addition the V-Modell specifies the responsibilities of each participant. In addition the V-Model gives guidelines for the system verification and validation process, by dividing the system in different levels of abstraction. Each level contains a development task and a parallel test task.

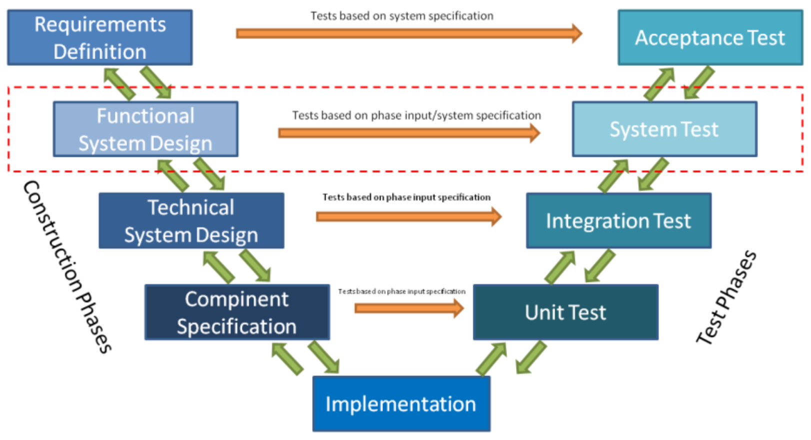


Figure 2-3 V-Model XT

V stands for "Verification and Validation". The left side of the "V" represents the decomposition of requirements and creation of system specifications. The right side of the "V" represents the corresponding test levels for integration of parts and their verification and validation [9].

The following test levels are described by the right side of the V-Model:

Acceptance testing

(A) Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system. (B) Formal testing conducted to enable a user, customer, or other authorised entity to determine whether to accept a system or component.

System testing

Testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. The developed methods in this work will mainly range in this level.

Integration testing

Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them.

Component testing

Testing of individual hardware or software components, or groups of related components.

2.3.3. **Black-Box Testing**

The IEEE is defining Black-Box and Black-Box Testing using the following terms (IEEE 610, [13]):

- **Black-Box:** A system or component whose inputs, outputs, and general function are known but whose contents or implementation are unknown or irrelevant.
- **Black-Box Testing:** Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.

Black-Box testing is testing using the functional requirements of a SUT, without knowledge of the internal structure. It uses the SUT inputs as the point of control (PoC) and its outputs as the point of observation (PoO). The fundamental difference between Black- and White-Box testing is the fact that tests do not deal with how a given output is produced, only whether it is the desired and expected output. The verification, validation and test (VVT) engineer, therefore, focuses only on the outputs generated in response to selected inputs and execution conditions and ignores the internal mechanism of the system. Therefore, the VVT engineer does not required any specific knowledge of the underlying system, and the testing is carried out at the system or individual subsystem level where the partitioning criteria is based on the system functional specifications [3].

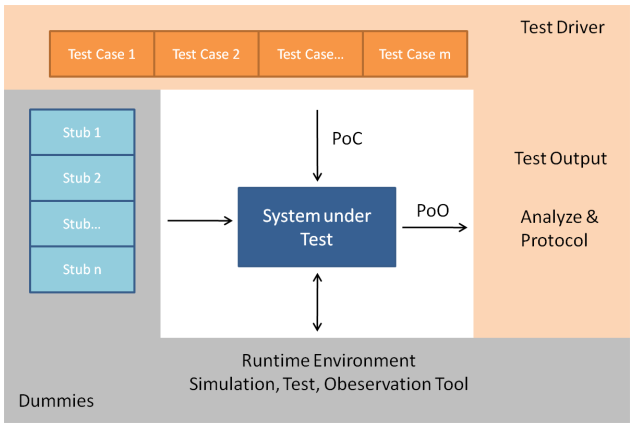


Figure 2-4 Black-Box Testing Environment and Components

Figure 2-4 illustrates the general view of the Black-Box testing over the system under test and the position of the used artefacts. In the Black-Box view the PoC, which will be used to stimulate the SUT, as well as the point of observation PoO, which is used to observe the results, are outside of the SUT.

Finite State Machine Testing

Finite State Machine testing is a possible application of the Black-Box testing approach.

The IEEE is defining Finite State Machines using the following term (IEEE 610, [13]):

- A computational model consisting of a finite number of states and transitions between those states, possibly with accompanying actions.

The purpose of Finite State Machine (FSM) testing method is mostly to evaluate systems for proper execution of control functions [3]. FSM modelling is based on automata theory, which involves the concepts of system states, events, transitions and activities. Engineered systems that embody FSM philosophy are characterized by a behaviour pattern where, under each state or mode, the system behaves (e.g., performs activities and generates outputs) in a specified and unique manner. The system remains in that state until a specific external input or internal event occurs. When that occurs, and certain conditions are fulfilled, the system transitions into another state, under which it may perform an entirely different and unique, set of tasks.

3. Demonstration System

A technical system will be used through this work as an example system, since this work focuses mainly on this kind of systems. In the following this system is described based on its system requirements specification (SRS) [14], usually created by a customer or system user. In most cases this specification is written in an informal natural language, which is not computable. In chapter 0 of this section, parts of the SRS will be modelled in a formal way using SysML State Machine Diagrams.

The 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.
- The functional system requirements are suitable to be verified and validated by tests.

3.1. The Aircraft Water Tank System

A passenger aircraft has portable water tanks to provide fresh potable water for washing rooms and toilets on board. This system will be called aircraft water tank system (AWTS) [42]. The system contains two tanks which are connected together. The output of the first tank is connected with the input of the second tank. A liquid source fills the first tank with liquid. Each tank has a continuous control system, connected to it, which controls the level of liquid contained in the tank to a given reference level. A proportional–integral- differential (PID) control system [43] will be used by default. While a liquid source fills liquid into the first tank its continuous controller regulates the outflow depending on its actual tank liquid level. The same applies to the second tank. In addition a control unit observes and controls the entire system to prevent fatal failures. The whole process will be initiated over a user interface. This interface regulates the liquid source flow and the overall system status.

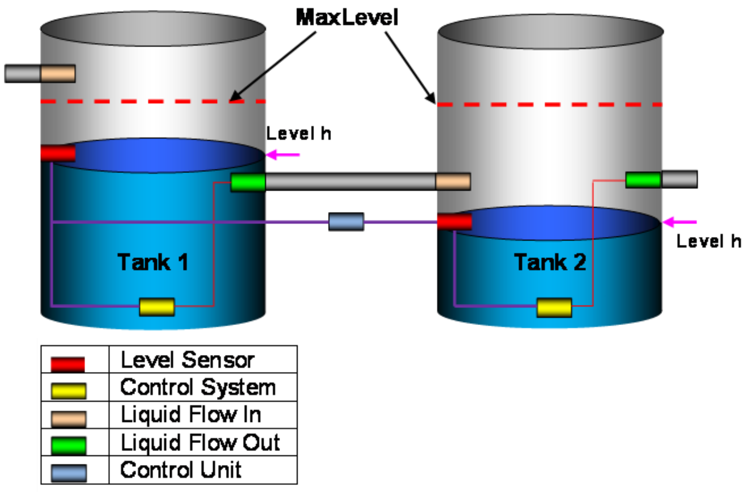


Figure 3-1 Aircraft water tanks with continuous controllers connected together

The coupled tank system depicted in Figure 3-1 is the core problem of the demonstration system. It is based on a frequently used demonstration model, which targets applied physics as well as control system engineering, among other disciplines [43]. It is also given in the Modelica book by Peter Fritzson [[2], Page 386]. The simple example model has been enhanced with the components specified above, to offer more observable, testable and controllable elements, to be used during the different parts of this work.

3.2. Possible System and Liquid Tank States

A typical technical system, for example from the fields aerospace or defence, has various states during its life cycle. A system that embodies the state based philosophy behaves under each state or mode, in a specified and unique manner [1]. The system remains in that state until a specified event occurs. When that event occurs and expected conditions are fulfilled, the system transitions into another state.

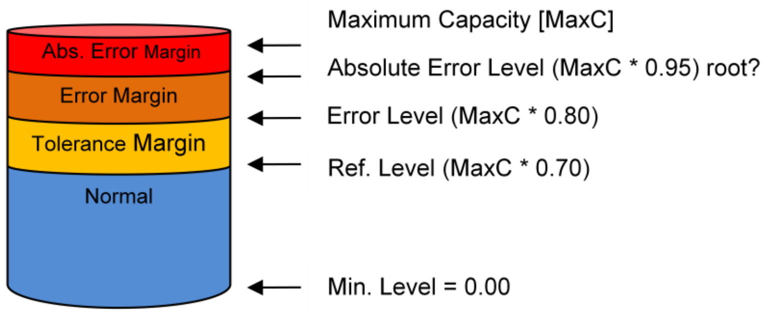


Figure 3-2 Possible states of tanks during the filling process

The following list presents the different states of a tank and their impact over the system and the liquid source if occurred:

- Offline: System is offline and is waiting to be connected with a liquid source.
- Normal: The system tank is in normal condition.
    - Liquid Source: The valve is open, so water can flow into the first tank.
    - Tank: The level is under the reference level of 70%. In this case only the output from the tank should be regulated to control this level and to keep it stable.
- Tolerance Margin: Since a control system can’t regulate the level of liquid immediately after passing the reference level, there is a need for a tolerance margin which can be reached while the controller is trying to regulate [43]. The tolerance margin is from 70% - 80%.
    - In this case the behaviour is the same as the normal state.
- Error Margin: If the liquid level of a tank reaches this margin some error handling tasks must be initiated.
    - Liquid Source: The valve will be closed immediately, so no water can flow into the tank system anymore until the level is in tolerance or normal again.
    - Tank: The level has reached 80% - 95% of the maximum tank level. In this case the input flow should be closed immediately using a signal to a control unit. But the system is still online so the tank and its controller can still try to regulate the level.
- Absolute Error Margin: If the liquid level reaches this margin, all emergency tries has been failed and the system must go offline immediately.
- Liquid Source: The valve will be closed immediately, so no water can flow into the tank system anymore.
    - Tank: No more regulation of the tank level, since the control unit is shutting down the system.
    - System Status: The shut down process will be initialized, so the system can go to the offline mode. An attendant must regulate the tank levels manually using the output valve.
    - Liquid Source: The valve will be closed immediately, so no water can flow into the tank system anymore.

3.2.1. **System states represented in the SysML State Machine Diagram**

A state machine contains all available system states, transitions and events and their relation to each other. It is a way to represent the dynamic behaviour of complex systems. In addition a finite state machine has a finite number of states and transitions between those states. A state machine diagram represents this information graphically.

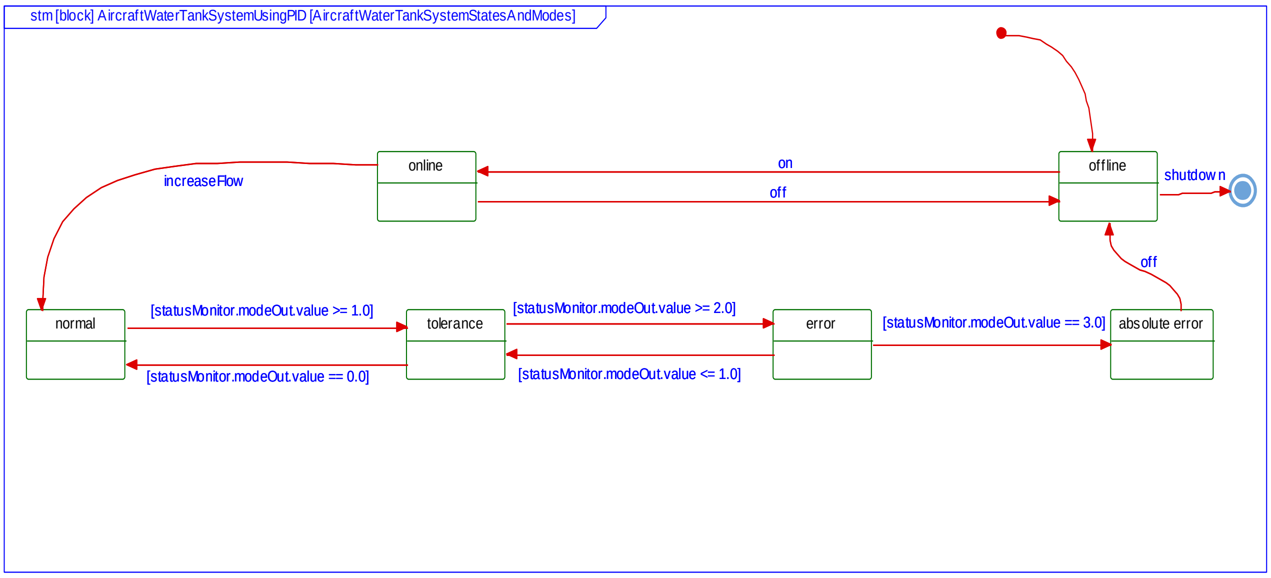

stm [block] AircraftWaterTankSystemUsingPID [AircraftWaterTankSystemStatesAndModes]
online
offline
on
off
shutdown
increaseFlow
off
normal
tolerance
error
absolute error
[statusMonitor.modeOut.value >= 1.0]
[statusMonitor.modeOut.value == 0.0]
[statusMonitor.modeOut.value >= 2.0]
[statusMonitor.modeOut.value <= 1.0]
[statusMonitor.modeOut.value == 3.0]

Figure 3-3 System states represented as a SysML state machine

A SysML State Machine Diagram, presented in Figure 3-3, is used to specify the runtime behaviour of a SysML Block, in our case the Aircraft Water Tank System, in terms of its possible states [1]. As the name implies, the main element is a state. A state declares a condition during the runtime of a system. In addition states have specialisations as initial or final stats. To change between states a transition is used. Transitions will be collected as paths through the different states. As the last main element of this diagram, an event dictates which messages are passed on these transitions. Events can also be implemented as guards, which react if a special condition is reached. A guard condition is represented in square bracket, for example the condition “[StatusMonitor.modeOut.value >= 3]” from Figure 3-3. This state chart diagram will be used to specify a test case later in chapter 6.

The implemented control unit indentifies the different states as the following:

States: Online, Offline

Modes: Normal = 0, Tolerance = 1, Error = 2, Absolute Error = 3

3.3. Use Case: Filling Tanks of the Aircraft Water Tank System

A use case represents expected functionalities of a system, in order to use this system to perform a business case. This is done from the respective actor's user point of view. A use case should have a predefined goal, in general a business process. Preconditions and post-conditions will define the conditions to start a business case and the expected results respectively.
This section will represent one main use case, first as natural text and later as a SysML use case diagram. This use case diagram will be used to specify a test case later in chapter 6.

3.3.1. Use Case Specification as Text

Title: Filling Tanks

Actor: Liquid Tank, Airport Staff

Goal: Filling up tanks with fresh water from a tanker

Trigger: Aircraft enters rendezvous point at the airport area

Preconditions: Aircraft is connected to a docking station. One of the two tanks is not filled with liquid.

Post conditions: Both water tanks are filled with fresh potable water.

Main Path:

1. One of the two tanks is not filled with fresh water, so the airport staff will call the fresh water service to send a truck with fresh water.
2. A water tanker will arrive to a rendezvous point near the aircraft.
3. The airport staff connects the water tanker valve to the aircraft water tank system.
4. The airport staff starts the system using a user interface, implemented at the aircraft fuselage.
5. The first tank will be filled with water. A controller (PI- or PID- controller) is responsible to control the water level in relation to a reference level. After reaching a specified level water will also flow into the second tank.
6. The liquid source will be closed if the user set the input flow to zero manually.
7. The system signals to close the valve which is connected to the first tank automatically.
8. The airport staff disconnects the liquid tanker valve from the first tank.
9. The truck driver will drive away from the rendezvous point.

Alternative Path 01:

7. a) The maximum tank level of one tank has been reached.

Alternative Path 02:

7. a) The liquid tanker goes empty.
8. b) The system does not close the valve automatically the airport worker has to close it manually.

3.3.2. **Use Case Specification as SysML Diagram**

This section introduces use case diagrams, which realize a behavioural aspect of the model [1]. The behavioural view has an emphasis on functionality, rather than the control and logical timing of the system. The use case diagram represents the highest level of abstraction of a view that is available in the SysML and it is used, primarily, to model requirements and contexts of a system. Not all steps listed in 3.3.1 will be depicted in a use case diagram, since the purpose of such a diagram is a more abstract view of the business case.

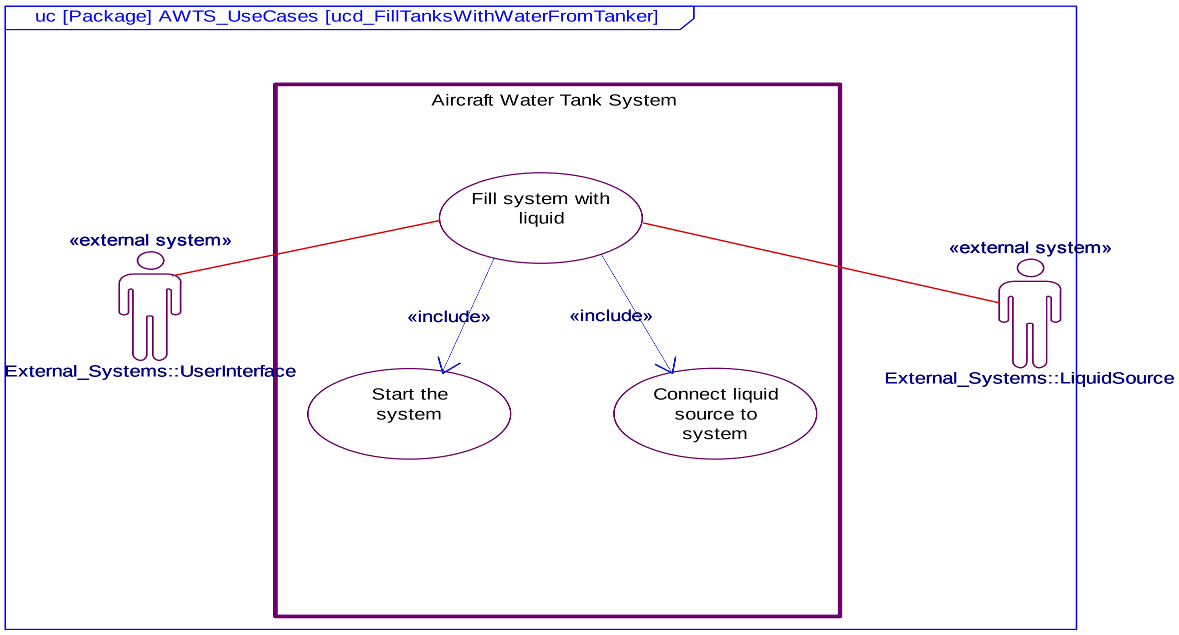

uc [Package] AWTS_UseCases [ucd_FillTanksWithWaterFromTanker]
Aircraft Water Tank System
Fill system with liquid
«external system»
External_Systems::UserInterface
«external system»
External_Systems::LiquidSource
«include»
«include»
Start the system
Connect liquid source to system

Figure 3-4 SysML Use Case Diagram: Filling up Tanks with Fresh Water from a Tanker

4. System Design with SysML and Modelica

As part of the model-based system engineering approach, a system design will be developed as a model in the OMG SysML. In the area of systems development modelling is a widely accepted part of engineering. It is used in order to shift complexity and to focus only on particular aspects of a problem. This level of abstraction makes it easier to solve problems for this aspect. An additional benefit is that, a model can be used for early prototyping, verification and validation of the system design [3]. This can be done by ignoring some real world problems, such as cost and time aspects for developing physical elements or safety critical tasks. Furthermore the fact that things are more easily un-done, since a model will be created as software, is essential. For example in the development process of the aircraft water tank system, two different control systems were available, whereas one controller system was more suitable as the other. Using modelling and simulation, the incompatible controller could easily be identified and replaced. However, ignoring real world problems also have its disadvantages, as for instance complex physical laws which can't be represented by models and will affect after developing the system in the real world.
When looking at the V-Model XT, the system design used in this work can be assigned to the level of Functional System Design [9]. In this level more functional aspects of the system requirements are covered, rather than technical or implementation details. This allows a higher level of abstraction. Figure 4-1 presents the left hand side of the V-Model and the level of abstraction.

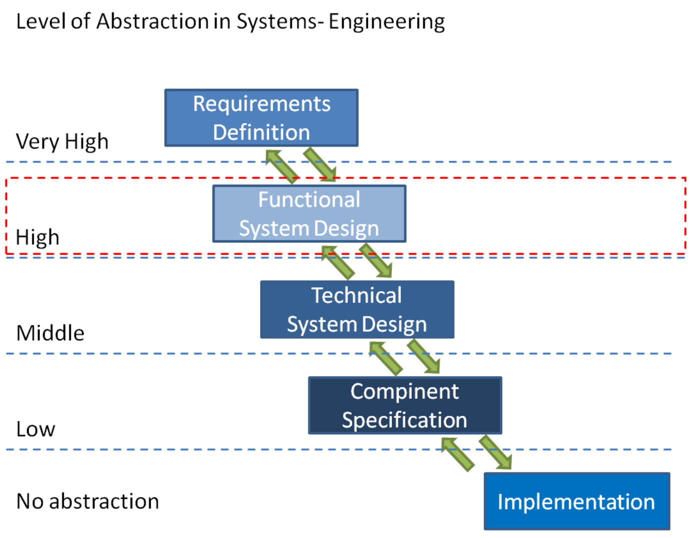


Figure 4-1 Level of Abstraction in Systems- Engineering using the V-Modell XT

A modelling environment is needed to offer a component based rather than visual component based modelling of systems. IBM Rational Rhapsody is a widely used engineering tool and available in the version 7.6 [39]. The SysML models in this work are created using Rhapsody. Since SysML is just a language specification, a tool will implement its own SysML meta-model. Therefore the used SysML elements are part of the Rhapsody-SysML meta-model. This is important to know in case of discrepancies with the SysML 1.2 specification.

4.1. Transformation Approach between SysML and Modelica

As mentioned above the SysML standard does not define any action language 2.2, so that by default the created model is not executable. However, this work will not just present the system design as static SysML diagrams, it will also present the design in Modelica, in order to make it executable for simulation.
Modelica and SysML, like UML, follow the object-oriented paradigm. Since both languages are well suited and adapted for the modelling of physical systems, the resulting structure is similar [1]. 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. But since the focus of this work is to use standardized concepts, the amount of used stereotypes is minimal.

The transformation between the SysML and Modelica languages is based on mapping, which is one possible formal and systematic approach [17]. The transformation approach focuses on both languages using their standardized semantic or meta-model, without enhancing or modifying them. This allows a bi-directional mapping [17], and as a result bi-directional transformation of models, for example used in reversed engineering [45]. Especially defining a new meta-model respectively UML profile will be avoided. For this must be mentioned by not focusing on one language to be prioritises translated, both languages will lose special features. A similar but much simpler approach was developed in the paper [8], so the actual approach can be seen as a completion and extension of this work, in order to cover the new SysML 1.2 and Modelica 3.2 specification, as well as a more generic approach of creating a system design with SysML and Modelica.

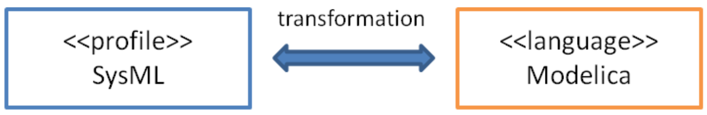


Figure 4-2 SysML and Modelica transformation approach

In parallel to this work an OMG working group is developing a further approach of integrating SysML and Modelica, which can be seen as a related work [44]. The SysML-Modelica Integration Group is developing an approach of modelling Modelica models with SysML. This means that the prior translated language is Modelica. Since SysML does not support all Modelica concepts, this will result in a new SysML profile called “SysML4Modelica”.
The SysML4Modelica approach does have its advantages, as well as disadvantages compared to the approach of using pure SysML as developed in this work:

- SysML4Modelica is domain specific to physical systems engineering, pure SysML is domain independent.

- Using SysML4Modelica a designer or architect must already have knowledge of Modelica. Using pure SysML it is possible to shift this problem to the tester or developer level where Modelica equations must be implemented as constraints.
- SysML4Modelica will result in a new profile, this is a risk when thinking of user acceptance in projects and missing tool support.
- A positive aspect of the SysML4Modelica approach is that many features of modelling with Modelica are covered, but by trying to bring pure SysML and Modelica together without any enhancements, some features may get lost, because there is no correspondent element in SysML and vice versa.

As a summery, SysML4Modelica is very similar to the approach in this work. It is a wider approach, but not applicable yet.

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. **SysML and Modelica Mapping of Language Elements**

As part of the translation process, mapping is a key task. Mapping two languages means to find commonalities in both. This process requires a full understanding of the involved languages and their elements. Therefore a well defined mapping is very important.

The tables below list a selected subset of SysML 1.2 elements, which are used for modelling the "Aircraft Water Tank System" demonstration model. The mapping table between SysML 1.2 and Modelica 3.2 consist of four sections:

1. ID of a mapping rule to be referenced at its application in chapter 4.2.2 and 4.2.3. The ID for mapped elements does have the form "1.x".
2. SysML (Rhapsody) element which should be mapped.
3. Modelica element which matches at most with the SysML element.
4. The commonality between these elements to confirm the correspondence.

Rule	SysML (Rhapsody)	Modelica	Commonality
ID	Package	package	Packages partition model elements into logical groupings.
1.1	A package is the basic unit of partitioning. SysML packages partition model elements into logical groupings to minimize circular dependencies and support modularisation among them. In the following mentioned sub packages are also packages.	Packages in Modelica are used for logical groupings. Packages may contain classes such as model and block, and sub packages. In the following mentioned sub packages are also packages	

Table 1 SysML Package → Modelica Package

Rule	SysML (Rhapsody)	Modelica	Commonality
ID	Block	block	Component representation as model element. This may include both structural and behavioural features as well as containing all its parts and properties.
1.2	SysML Blocks defines composite system entities in SysML. Blocks are modular units. Each block defines a collection of features to describe a component, sub component or system of interest.	In Modelica an element such as a component is a class. The basic class concept is “model”. A Modelica “block” has the same properties as a “model” but with some restrictions. A block connector instance must have a specified direction (input, output)	

Table 2 SysML Block → Modelica Block

Rule	SysML (Rhapsody)	Modelica	Commonality
ID	Attribute	Variable	Property which contains data and has a specified data type.
1.3	Property of a block which contains data. Since SysML is an executable language independent of the attribute type it can be a pre defined basic or a user defined data type.	Property of a class which contains data and is from a pre defined data type. The variability can be defined as constant, parameter, continuous or discrete.	

Table 3 SysML Attribute → Modelica Variable

Rule	SysML (Rhapsody)	Modelica	Commonality
ID	Part	Variable (Part)	Property which represents a sub component or sub system.
1.4	A SysML Block may contain a sub component or sub system which is also a Block. This sub system is a part of the higher level Block.	Property of a class which represents a sub component or sub system.	

Table 4 SysML Part → Modelica Variable (Part)

Rule	SysML (Rhapsody)	Modelica	Commonality
ID	Association (Part)		Association to represent the relationship between components.
1.5	A connection between a SysML Block and its internal part. In addition the multiplicity and type of association is also given. This type of association can also be called cardinality.	No corresponding element, but part of the Modelica syntax to specify a sub component relationship.	

Table 5 SysML Association (Part) → Modelica Syntax Element

Rule	SysML (Rhapsody)	Modelica	Commonality
ID	Flow Specification	connector	Definition of ports and the data flow. This port can be used to connect components to each other.
1.6	A flow Specification defines a set of input and/or output flows for a non composite flow port.	A class with restrictions, which defines a port. This port can be used to connect components to each other.	

Table 6 SysML FlowSpecification → Modelica Connector

Rule	SysML (Rhapsody)	Modelica	Commonality
ID	Flow Port Node	Instance of connector	Instance of a port definition, which is part of a component. It specifies an interaction point for a component.
1.7	A flow port node is the actual instance of a flow specification. It presents an interaction point where items can flow into or out of a block. The direction is indicated by the direction of the arrow in the Flow Port Node.	The actual port is an instance of a connector. It is part of a component and is used to describe interaction points. Its direction has to be defined in the owner class as input or output.	

Table 7 SysML Flow Port Node → Modelica Instance of Connector

Rule	SysML (Rhapsody)	Modelica	Commonality
ID	Atomic Flow Port Node	Input, Output	Definition of a port and instantiating it without defining a special type. This element can be used if there is only one data to flow and the type of the needed interaction point is only used once in the model.
1.8	An atomic flow port defines a port and is also used as its instance. This element can be used if there is only one data to flow and the type of the needed interaction point is only used once in the model. The direction is indicated by the direction of the arrow in the Atomic Flow Port Node.	In Modelica one can define a variable using a basic type and specify if it shall be used as a port by defining the prefix input or output. But this is not a recommended solution.	

Table 8 SysML Atomic Flow Port Node → Modelica Instance of Connector

Rule	SysML (Rhapsody)	Modelica	Commonality
ID	Connector flow(x,y) (between flowPorts)	Connection equation connect(x,y)	Specifies interaction between elements, by connecting their ports.
1.8	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 is a special equation which specifies an interaction between connectors or values of different components.	

Table 9 SysML Connector → Modelica Connection

Rule	SysML (Rhapsody)	Modelica	Commonality
ID	Flow (FlowDirection)	Causality of connector instance	Specifies the flow direction between two connected components or ports. Note: Bidirectional flow is not possible.
1.9	A Flow specifies the direction and type of exchanged information between system elements. It allows you to specify the flow of data and commands. The direction describes the flow direction.	In Modelica the available directions are “input” or “output”. A port used in a “block” must have causality.	

Table 10 SysML Flow (FlowDirection) → Modelica Causality of connector instance

Rule	SysML (Rhapsody)	Modelica	Commonality
ID	Generalisation/Specialisation	Inheritance (extends)	To support reuse of an in general specified component. The definition can be enhanced to specify a new component. Generalisation/Specialisation may have the same impact as inheritance.
1.10	To support reuse of an in general specified component. The definition can be enhanced to specify a new component. To do so generalisation describes a relationship between a general classifier and a specialized classifier. The specialised classifier inherits structure and behaviour of the general classifier.	To support reuse the general specification of a component can be enhanced to specify a new component. A block can inherit structure and behaviour of another block.	

Table 11 SysML Inheritance (Gen/Spec) → Modelica extends

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

Table 12 SysML Datatype Double → Modelica Datatype Real

Rule	SysML (Rhapsody)	Modelica	Commonality
ID	Description	Comment	Describes an element further more in an informal way.
1.12	Describes a SysML element further more in an informal way.	Describes a Modelica element further more in an informal way.	

Table 13 SysML Description → Modelica Comment

4.1.2. **Additional Modelica Syntax as SysML Stereotypes**

As mentioned above SysML Stereotypes can be used to adapt and satisfy some semantics of the executable language.

A stereotypes table may consist of several sections:

1. Rule: ID of a stereotype to be referenced at its application in chapter 4.2.2. The ID for stereotype elements does have the form “2.x”
2. Applicable to SysML (Rhapsody) element: Since Stereotypes must been applied to specified language elements this column gives the corresponding SysML element
3. Needed Modelica addition: The reason why this additional stereotype will be needed
4. Benefit/Effect: Describes the benefit of using this additional information and the effect on a SysML or a Modelica model element.
5. Stereotype name: Represents the name of the created stereotype
6. Tags: A stereotype may contain tags as additional information for example the unit of a value.
7. Effect: Real impact on an element. This may result as code for example as a prefix.

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

Rule	Applicable to SysML (Rhapsody) element	Needed Modelica addition	Benefit/Effect
ID	Class (SysML block)	partial	A block which offers general structure and behaviour for a group of specialised block. This block can’t be instantiated as a component.
2.1			

Stereotype name	Tags	Effect
		Prefix
<<abstract>>	non	partial

Table 14 SysML Stereotype <<abstract>> for Modelica partial

b. There are four variability levels of attributes in Modelica, so a SysML "attribute" needs an additional Stereotype to recognise its variability. This variability indicator is necessary for parameter and constant but not for continuous and discrete attributes since Modelica can identify this automatically. In addition the optional unit of a value will also be adopted and presented by a tag of this Stereotype. In rhapsody there is also a need for a DataType called "variability_type" which supports enumeration of the different identifiers.

Rule	Applicable to SysML (Rhapsody) element	Needed Modelica addition	Benefit/Effect
ID 2.2	Attribute	variability	Depending on its type an attribute will get a type prefix. The type also specifies how the initialisation will be defined.

Stereotype name	Tags	Characteristic	Effect	
<<variable>>	variability (variability_type)	parameter	Prefix "parameter"	initialValue interpretation "…=x;"
		constant	Prefix "constant"	initialValue interpretation "…=x;"
		discrete-time (usage is optional)	Prefix -non-	initialValue interpretation "(start = x, …)"
		continuous-time (usage is optional)	Prefix -non-	initialValue interpretation "(start = x, …)"
	unit	String	(…, unit="…")	

Table 15 SysML Stereotype <<variable>> for Modelica variability and unit

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

Rule	Applicable to SysML (Rhapsody) element	Needed Modelica addition	Benefit/Effect
ID 2.3	Generalisation	Modify inherit variable values	Set default values for inherited attributes.

Stereotype name	Tags	Characteristic	Effect
<<extendsRelation>>	typeModification	String with Dot-Notation	extends…(…=x, …=x);

Table 16 SysML Stereotype <<extendsRelation>> Modelica modification of inherit variable value

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

Rule	Applicable to SysML (Rhapsody) element	Needed Modelica addition	Benefit/Effect
ID	Object (SysML Part)	Instance modification	The value of inherited variables can be modified for instances. The variable name and its new value in brackets will be appended to the instance declaration.
2.4			
Stereotype name	**Tags**	**Characteristic**	**Effect**
<<composite>>	instanceModification	String with Dot-Notation	*Instance*… (…=x, …=x);

Table 17 SysML Stereotype <<composite>> for Modelica instance modification

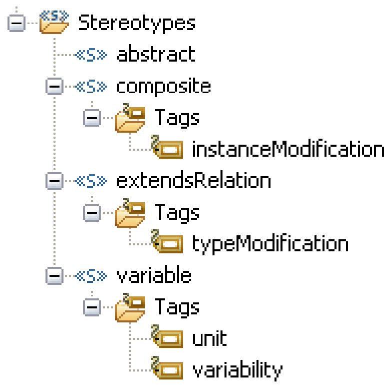

```
Stereotypes
  «S» abstract
  «S» composite
    Tags
      instanceModification
  «S» extendsRelation
    Tags
      typeModification
  «S» variable
    Tags
      unit
      variability
```

Figure 4-3 List of 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.

The tables below list parametric elements from SysML 1.2, which are used to represent mathematical constraint for modelling the “Aircraft Water Tank System” demonstration model. The table consists of three sections:

1. ID of a constraint element to be referenced at its application in chapter 0. The ID does have the form “3.x”.
2. SysML (Rhapsody) constraint element.
3. Description of the element.

Rule ID	SysML (Rhapsody) element	Description
3.1	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.
3.2	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.
3.3	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.
3.4	Value Binding Path	Binding connectors connect constraint parameters to each other and to value properties. They express an equality relationship between their bound elements.
3.5	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 18 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. SysML to Modelica Transformation Application using the Example Model

The following is a full SysML 1.2 model of the demonstration system. The model has been created using IBM Rational Rhapsody 7.6 [39] As mentioned above Rational Rhapsody has its own SysML Profile implementation which can differ in some particular ways from the OMG SysML 1.2. However, while creating the model such variations of the OMG SysML 1.2 specification have been avoided.

System structure is depicted in Package Diagrams (Pkg), Block Definition Diagrams (BBD), Internal Block Diagrams (IBD) and in Parametric Diagrams (PAR). Every SysML element may represent in a diagram and will be translated into corresponding Modelica code. The coherent Modelica model code can be found in the Appendix A.

Note: Since the translation will be demonstrated step by step the resultant code for each step will be highlighted in green.

4.2.1. **System structure with SysML Package Diagram**

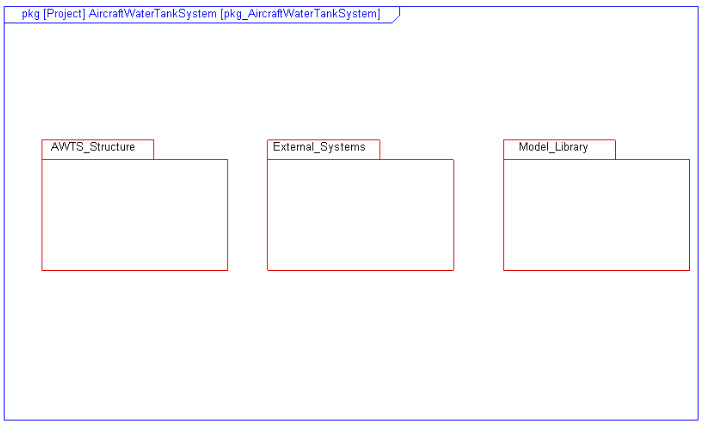

pkg [Project] AircraftWaterTankSystem [pkg_AircraftWaterTankSystem]
AWTS_Structure
External_Systems
Model_Library

Figure 4-4 Aircraft Water Tank System Package Structure using SysML Package Diagrams

SysML elements such as blocks should be grouped in packages. For the OpenModelica (OM) it is important to signal a package membership for each element, so that OM can load all components included in a specified package and its sub packages. In order to do so there is a need for a high level class, Modelica type “package”, in the project folder. Furthermore a Modelica class element needs a “within…” declaration in its first code line including a full qualified package name to signal its membership. A Rhapsody project package will result into a high level Modelica package. Each SysML package and sub package will result into sub packages of the high level Modelica package.

Resultant Modelica code using rule 1.1:

High level Modelica package.

<<package>> AircraftWaterTankSystem → package.mo (Modelica)

`package AircraftWaterTankSystem`
`end AircraftWaterTankSystem;`

Sub packages of the high level Modelica package.

<<package>> AircraftWaterTankSystem → package.mo (Modelica)

`package AWTS_Structure`
`end AWTS_Structure;`

<<package>> Model_Library → package.mo (Modelica)

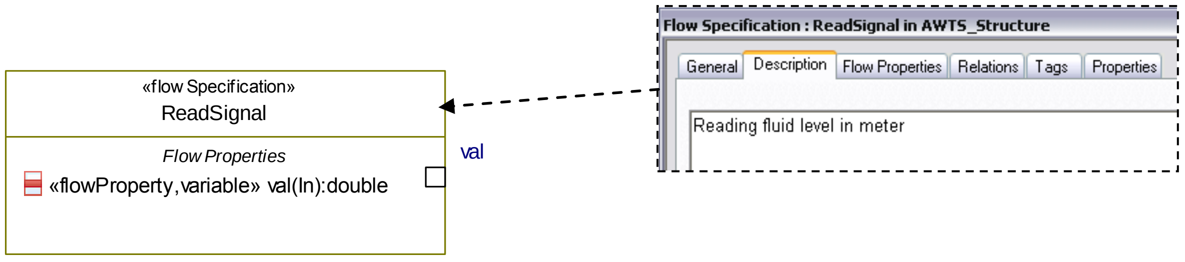
`package Model_Library`
`end Model_Library;`

<<package>> AircraftWaterTankSystem → package.mo (Modelica)

`package External_Systems`
`end External_Systems;`

4.2.2. **System structure with SysML Block Definition Diagram**

A SysML block defines a composite system entity which represents a real physical component, sub system or system. The SysML Block Definition Diagram can be used to define relationships between blocks such as generalizations, associations with types and multiplicity, and dependencies. Since Rhapsody allows creating elements like blocks and flow specifications manually using the project browser, the BDD will also be used to display a block and its properties.



Reading fluid level in meter

Figure 4-5 ReadSignal FlowSpecification

A flow specification may have attributes which has the Stereotype “flowProperty”. Each flow property has a data type and a direction (in, out, or inout). Since a flow specification can be reused by different instances with different flow directions the attribute direction should not be fix at declaration time. However SYSMOD recommends defining an initial direction which can be redirected when this value is used in a parametric diagram. As an addition one can also select the Stereotype “variable” for a flow property to assign a variability type and a unit for it. Component descriptions from Rhapsody will also be respected.

Resultant Modelica code using rule 1.3, 1.6, 1.11, 1.12, 2.1:

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

```
within AircraftWaterTankSystem.AWTS_Structure;

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

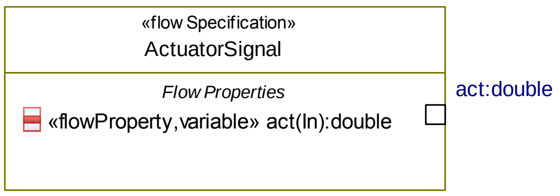


Figure 4-6 ActuatorSignal FlowSpecification

Resultant Modelica code using rule 1.3, 1.6, 1.11, 1.12, 2.1:

<< flowProperty >> ActuatorSignal → ActuatorSignal.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

connector ActuatorSignal // Signal to an actuator for setting valve position
  Real act;
end ActuatorSignal;
```

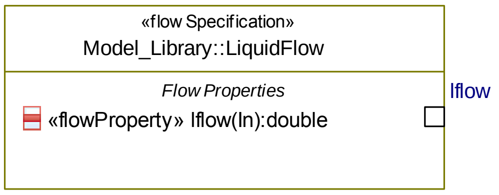


Figure 4-7 LiquidFlow FlowSpecification

To support reuse model elements such as this flow specification can be stored in the "Model_Library" package.

Resultant Modelica code using rule 1.3, 1.6, 1.11, 1.12, 2.1:

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

```
within AircraftWaterTankSystem.Model_Library;

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

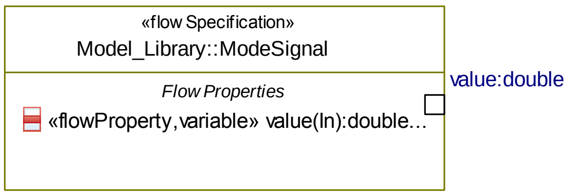

«flow Specification»
Model_Library::ModeSignal
Flow Properties
«flowProperty,variable» value(In):double...
value:double

Figure 4-8 ModeSignal FlowSpecification

Resultant Modelica code using rule 1.3, 1.6, 1.11, 1.12, 2.1:

<< flowProperty >> ModeSignal → ModeSignal.mo (Modelica)

```
within AircraftWaterTankSystem.Model_Library;

connector ModeSignal // Signal to represent the system or component mode
  Real value; // 0=normal, 1=tolerance, 2=error, 3=abs. error
end ModeSignal;
```

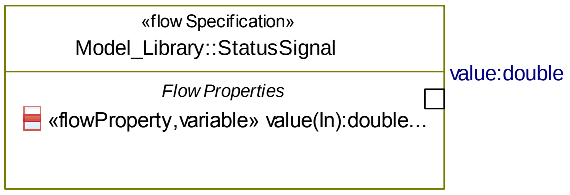

«flow Specification»
Model_Library::StatusSignal
Flow Properties
«flowProperty,variable» value(In):double...
value:double

Figure 4-9 StatusSignal FlowSpecification

Resultant Modelica code using rule 1.3, 1.6, 1.11, 1.12, 2.1:

<< flowProperty >> StatusSignal → StatusSignal.mo (Modelica)

```
within AircraftWaterTankSystem.Model_Library;

connector StatusSignal // Signal to represent the system or component status
  Real value; // 0=offline, 1=online
end StatusSignal;
```

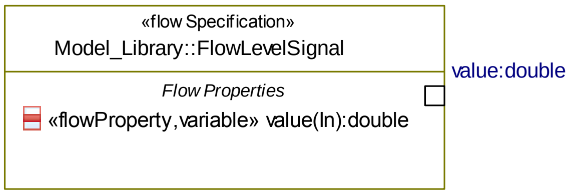

«flow Specification»
Model_Library::FlowLevelSignal
Flow Properties
«flowProperty,variable» value(In):double
value:double

Figure 4-10 FlowLevelSignal FlowSpecification

Resultant Modelica code using rule 1.3, 1.6, 1.11, 1.12, 2.1:

<< flowProperty >> FlowLevelSignal → FlowLevelSignal.mo (Modelica)

```
within AircraftWaterTankSystem.Model_Library;

connector FlowLevelSignal // Liquid flow level as measurement
  Real value (unit = "m3/s");
end FlowLevelSignal;
```

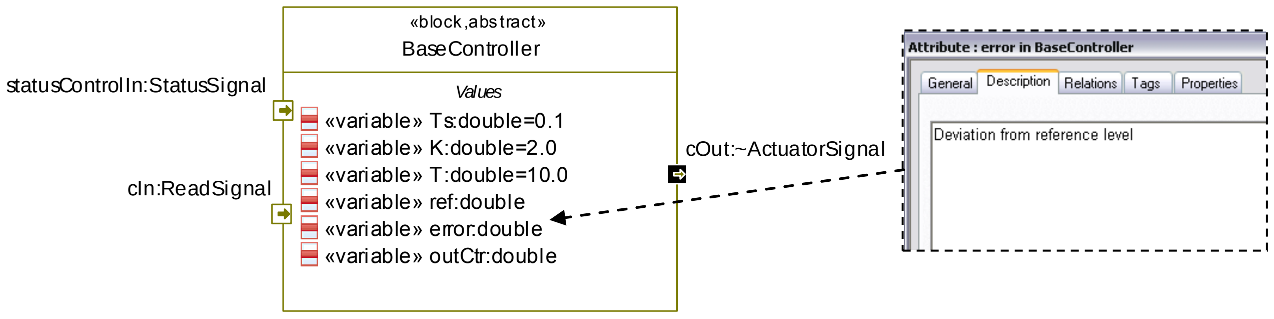


Figure 4-11 BaseController Block

The BaseController is a SysML block and has the Stereotype abstract which signals that it is a partial Modelica block. The block does have the depicted attributes in the Values section. The package membership is given by the package structure in Figure 4-4.

Resultant Modelica code using rule 1.2, 1.3, 1.4, 1.11, 1.12, 2.1, 2.3:

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

```
within AircraftWaterTankSystem.AWTS_Structure;

partial block BaseController
  AircraftWaterTankSystem.Model_Library.StatusSignal statusControlIn;
  AircraftWaterTankSystem.AWTS_Structure.ReadSignal cIn;
  AircraftWaterTankSystem.AWTS_Structure.ActuatorSignal cOut;

  Real Ts(unit = "s") = 0.1;
  Real K = 2; // Gain
  Real T(unit = "s") = 10; // Time constant
  Real ref; // Reference level
  Real error; // Deviation from reference level
  Real outCtr; // Output control signal
equation
end BaseController;
```

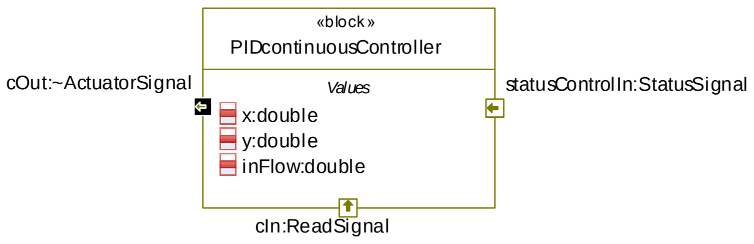


Figure 4-12 PIDcontinuousController Block

Resultant Modelica code using rule 1.2, 1.3, 1.4, 1.11, 1.12, 2.1:

<<block>> PIDcontinuousController → PIDcontinuousController.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

block PIDcontinuousController
  Real x; // State variable of continuous PID controller
  Real y; // State variable of continuous PID controller
equation
end PIDcontinuousController;
```

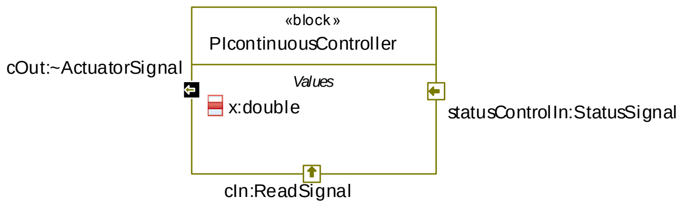


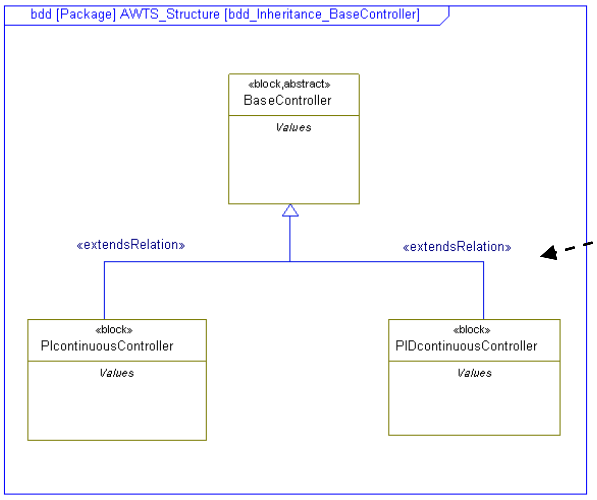
Figure 4-13 PIcontinuousController Block

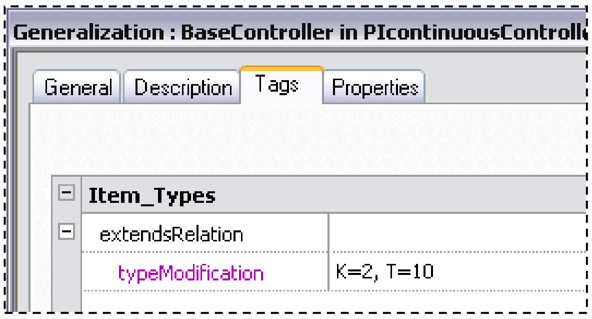
Resultant Modelica code using rule 1.2, 1.3, 1.4, 1.11, 1.12, 2.1:

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

```
within AircraftWaterTankSystem.AWTS_Structure;

block PIcontinuousController
  Real x; // State variable of continuous PID controller
equation
end PIcontinuousController;
```




Generalization : BaseController in PIcontinuousControll

Item_Types	
extendsRelation	
typeModification	K=2, T=10

To cover also a part of the object oriented approach the control systems “PIcontinuousController” and “PIDcontinuousController” are specializations of “BaseController”. To modify attribute values from a partial block the Stereotype “extendedRelation” offers a tag “typeModification”. In This case “K” and “T” are inherited variables to be modified.

Resultant Modelica code using rule 1.10, 2.2:

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

```
within AircraftWaterTankSystem.AWTS_Structure;

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

<<block>> PIDcontinuousController → PIDcontinuousController.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

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

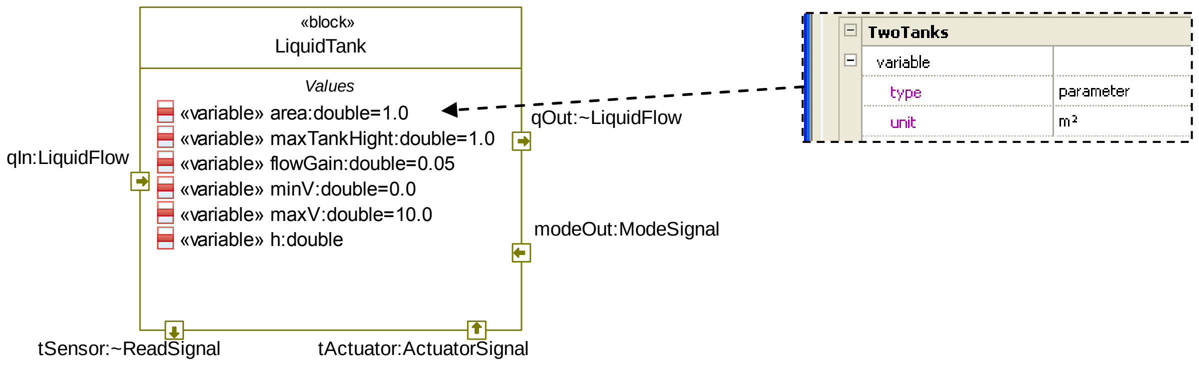


Figure 4-14 LiquidTank Block

Resultant Modelica code using rule 1.2, 1.3, 1.4, 1.11, 1.12, 2.1:

<<block>> LiquidTank → LiquidTank.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

block LiquidTank
  AircraftWaterTankSystem.Model_Library.ModeSignal modeOut;
  AircraftWaterTankSystem.AWTS_Structure.ReadSignal tSensor;
  AircraftWaterTankSystem.AWTS_Structure.ActuatorSignal tActuator;

  AircraftWaterTankSystem.Model_Library.LiquidFlow qIn;
  AircraftWaterTankSystem.Model_Library.LiquidFlow qOut;

  Real area(unit = "m2") = 1; //Will be given as a parameter
  Real maxTankHight (unit = "m") = 1.0; //Will be given as a parameter
  Real flowGain (start = 1.99, unit = "m2/s") = 0.05;
  Real minV = 0; // Minimum for output valve flow
  Real maxV = 10; // Limit for output valve flow
  Real h(start = 0.0, unit = "m"); //Tank level
equation
end LiquidTank;
```

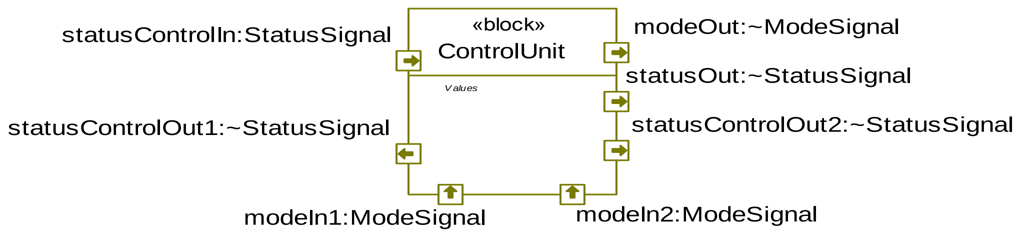


Figure 4-15 ControlUnit Block

Resultant Modelica code using rule 1.2, 1.4:

<<block>> ControlUnit → ControlUnit.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

block ControlUnit
  AircraftWaterTankSystem.Model_Library.ModeSignal modeIn1;
  AircraftWaterTankSystem.Model_Library.ModeSignal modeIn2;
  AircraftWaterTankSystem.Model_Library.StatusSignal statusControlIn;
  AircraftWaterTankSystem.Model_Library.ModeSignal modeOut;
  AircraftWaterTankSystem.Model_Library.StatusSignal statusControlOut1;
  AircraftWaterTankSystem.Model_Library.StatusSignal statusControlOut2;
  AircraftWaterTankSystem.Model_Library.StatusSignal statusOut;
  equation
end ControlUnit;
```

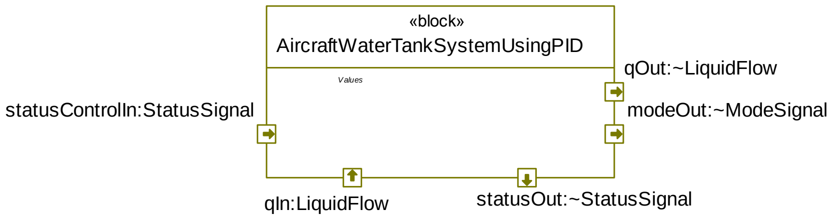


Figure 4-16 AircraftWaterTankSystem Block

Resultant Modelica code using rule 1.2, 1.4:

<<block>> AircraftWaterTankSystemUsingPID → AircraftWaterTankSystemUsingPID.mo (M.)

```
within AircraftWaterTankSystem.AWTS_Structure;

block AircraftWaterTankSystemUsingPID
  AircraftWaterTankSystem.Model_Library.LiquidFlow qIn;
  AircraftWaterTankSystem.Model_Library.LiquidFlow qOut;
  AircraftWaterTankSystem.Model_Library.StatusSignal statusControlIn;
  AircraftWaterTankSystem.Model_Library.StatusSignal statusOut;
  AircraftWaterTankSystem.Model_Library.ModeSignal modeOut;
equation
end AircraftWaterTankSystemUsingPID;
```

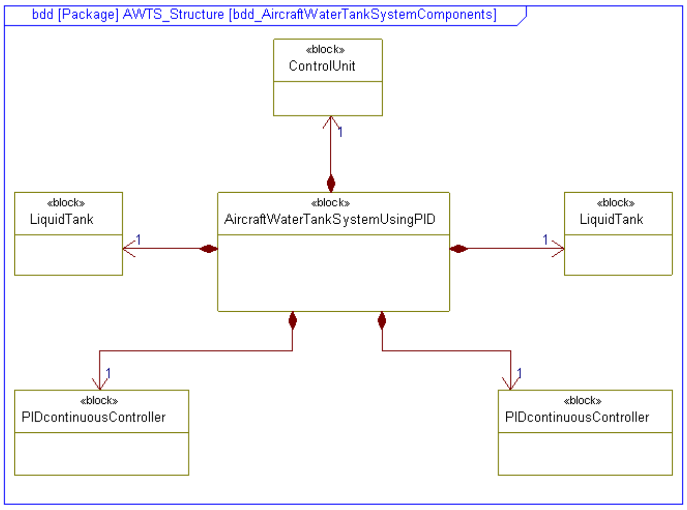


Figure 4-17 BBD Aircraft Water Tank System with PID continuous controllers

The BDD in Figure 4-17 presents the “Aircraft Water Tank System” and its relationship to other system components which are connected using the association type composition.

Resultant Modelica code using Rule 1.4, 1.5:

AircraftWaterTankSystemUsingPID.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

block AircraftWaterTankSystemUsingPID
  AircraftWaterTankSystem.AWTS_Structure.ControlUnit statusMonitor;
  AircraftWaterTankSystem.AWTS_Structure.LiquidTank tank1;
  AircraftWaterTankSystem.AWTS_Structure.LiquidTank tank2;
  AircraftWaterTankSystem.AWTS_Structure.PIDcontinuousController pidContinuous1;
  AircraftWaterTankSystem.AWTS_Structure.PIDcontinuousController pidContinuous2;
...
end AircraftWaterTankSystemUsingPID;
```

4.2.3. **System structure with SysML Internal Block Diagram**

To get a more detailed view on a component and its subcomponents (parts) a new diagram has been implemented in SysML. The Internal Block Diagram (IBD) captures the internal structure of a block in terms of parts, properties and connectors between properties. The primary purpose of internal block diagrams, in conjunction with block definition diagrams, is to express system structural decomposition and interconnection of its parts.

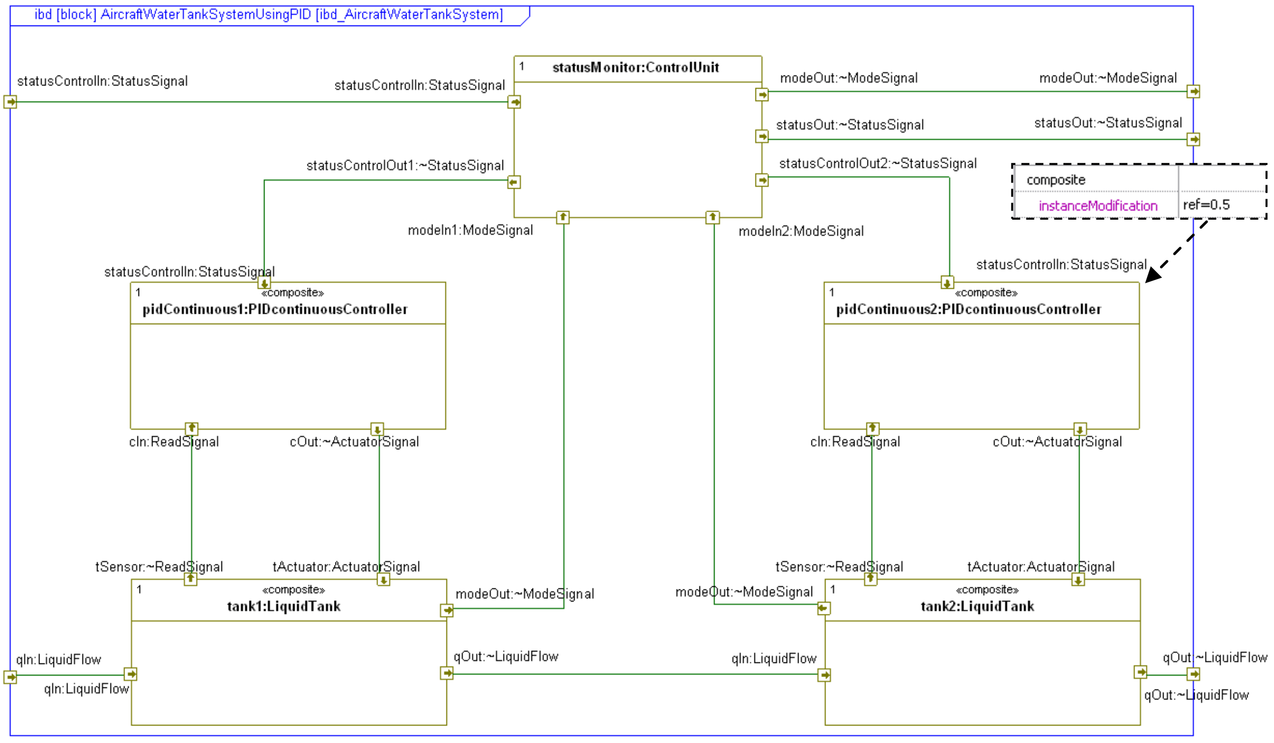

ibd [block] AircraftWaterTankSystemUsingPID [ibd_AircraftWaterTankSystem]
statusMonitor:ControlUnit
statusControlIn:StatusSignal
modeOut:~ModeSignal
statusOut:~StatusSignal
statusControlOut1:~StatusSignal
statusControlOut2:~StatusSignal
modeIn1:ModeSignal
modeIn2:ModeSignal
composite
instanceModification
ref=0.5
«composite»
pidContinuous1:PIDcontinuousController
pidContinuous2:PIDcontinuousController
cIn:ReadSignal
cOut:~ActuatorSignal
tSensor:~ReadSignal
tActuator:ActuatorSignal
tank1:LiquidTank
tank2:LiquidTank
qIn:LiquidFlow
qOut:~LiquidFlow

Figure 4-18 IBD Aircraft Water Tank System using PID

The IBD in Figure 4-18 displays the parts of the AWTS and their associations to each other. The value of first level attributes of instantiated parts can be modified in Rhapsody directly in the Tab “Attributes” → “Value”, for example “area” from “tank1”. However to set a value of an inherited attribute or a nested attribute the Stereotype “composite” must be used. A textual expression containing the attribute name and value finds place in the tag “instanceModification”, for example “ref” or “K” from PIDcontiuousControler. The parts are connected with SysML flow ports. As an addition this diagram finally defines the flow direction.

Resultant Modelica code using rule 1.5, 1.9, 2.4:

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

```
within AircraftWaterTankSystem.AWTS_Structure;

partial block BaseController
  input AircraftWaterTankSystem.Model_Library.StatusSignal statusControlIn;
  input AircraftWaterTankSystem.AWTS_Structure.ReadSignal cIn;
  output AircraftWaterTankSystem.AWTS_Structure.ActuatorSignal cOut;
...
end BaseController;
```

<<block>> LiquidTank → LiquidTank.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

block LiquidTank
  output AircraftWaterTankSystem.Model_Library.ModeSignal modeOut;
  output AircraftWaterTankSystem.AWTS_Structure.ReadSignal tSensor;
  input AircraftWaterTankSystem.AWTS_Structure.ActuatorSignal tActuator;

  input AircraftWaterTankSystem.Model_Library.LiquidFlow qIn;
  output AircraftWaterTankSystem.Model_Library.LiquidFlow qOut;
...
end LiquidTank;
```

<<block>> ControlUnit → ControlUnit.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

block ControlUnit
  input AircraftWaterTankSystem.Model_Library.ModeSignal modeIn1;
  input AircraftWaterTankSystem.Model_Library.ModeSignal modeIn2;
  input AircraftWaterTankSystem.Model_Library.StatusSignal statusControlIn;
  output AircraftWaterTankSystem.Model_Library.ModeSignal modeOut;
  output AircraftWaterTankSystem.Model_Library.StatusSignal statusControlOut1;
  output AircraftWaterTankSystem.Model_Library.StatusSignal statusControlOut2;
  output AircraftWaterTankSystem.Model_Library.StatusSignal statusOut;
  equation
end ControlUnit;
```

AircraftWaterTankSystemUsingPID.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

block AircraftWaterTankSystemUsingPID
...
  ... tank1 (area = 0.5, maxTankHight = 1);
  ... tank2 (area = 1, maxTankHight = 1);
  ... pidContinuous1 (ref = 0.5);
  ... pidContinuous2 (ref = 0.5);
  input AircraftWaterTankSystem.Model_Library.LiquidFlow qIn;
  output AircraftWaterTankSystem.Model_Library.LiquidFlow qOut;
  input AircraftWaterTankSystem.Model_Library.StatusSignal statusControlIn;
  output AircraftWaterTankSystem.Model_Library.StatusSignal statusOut;
  output AircraftWaterTankSystem.Model_Library.ModeSignal modeOut;
equation
  connect(statusMonitor.statusControlOut1, pidContinuous1.statusControlIn);
  connect(statusMonitor.statusControlOut2, pidContinuous2.statusControlIn);
  connect(tank1.modeOut, statusMonitor.modeIn1);
  connect(tank2.modeOut, statusMonitor.modeIn2);
  connect(tank1.qOut, tank2.qIn);
  connect(tank2.qOut, qOut);
  connect(tank1.tSensor, pidContinuous1.cIn);
  connect(pidContinuous1.cOut, tank1.tActuator);
  connect(tank2.tSensor, pidContinuous2.cIn);
  connect(pidContinuous2.cOut, tank2.tActuator);
  connect(statusMonitor.modeOut, modeOut);
  connect(statusControlIn, statusMonitor.statusControlIn);
  connect(statusMonitor.statusOut, statusOut);
  connect(qIn, tank1.qIn);
end AircraftWaterTankSystemUsingPID;
```

4.2.4. **Block Definition Diagrams of the constraint blocks**

As mentioned above, in Modelica behaviour is expressed in a mathematical form using equations. Expressing pure mathematical equations in diagrams are not part of the UML or SysML language. In UML, as well as in SysML, one can use an “opaque expressions” to write a textual statement which is uninterpreted. The expression can be associated to a model by using variable identifiers from the model context, this can be evaluated (Chapter 7.3.36, [18]). However, using opaque expressions is a very informal way and basically a workaround. SysML offers a powerful concept which provides expressing generic or basic mathematical equations as constraints. A constraint block encapsulates a constraint to support its reuse. The block contains the actual constraint and constraint parameters. Constraint parameters are used in the constraint as model independent variables, to be linked with real model elements using SysML binding connectors. However, a particular constraint is also specified in an informal language, but a more formal language such as OCL or MathML could also be used (Chapter 10, [19]).

In Rhapsody a constraint block will be defined once for a package and can be instantiated as a constraint property. A block definition diagram will be used to represent the relationship between a block and a constraint block. But the actual translation into Modelica equations is not possible by just using this information. Only when the real model properties are linked to a constraint property, the complete equation can be translated from SysML to Modelica.
The following diagrams show the relationship, between constraint blocks and blocks.

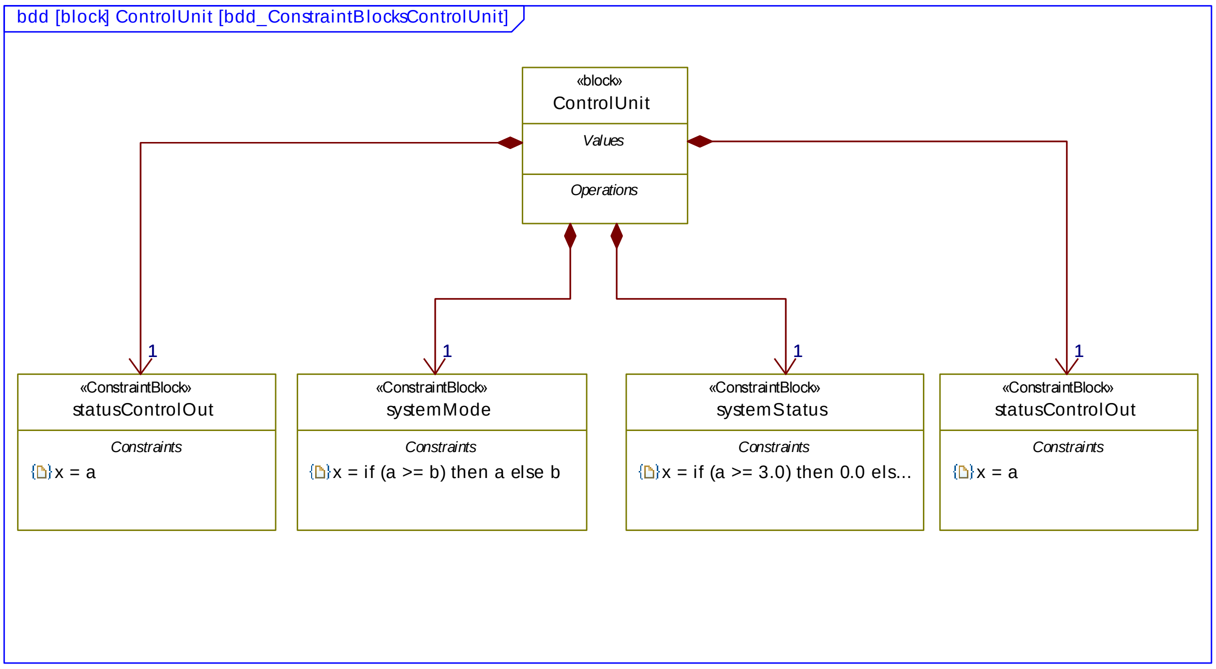


Figure 4-19 BDD Constraint Blocks of Control Unit

Since the control unit represents the overall system status its constraints are mainly condition based. As you can see in Figure 4-19 the constraint block “statusControlOut” is used twice, since this supports the reuse.

- The “statusControlOut” constraint is used a simple equation, since it is associating two variables, in order to allocate the value from “a” to “x”.
- The “systemMode” constraint is assigning the overall mode using the modes received from all tanks.
- The “systemStatus” constraint assigns if the system goes online or offline by using the results from “systemModel”. As mentioned in chapter 0 the system will shut down if the mode is in the absolute error margin.

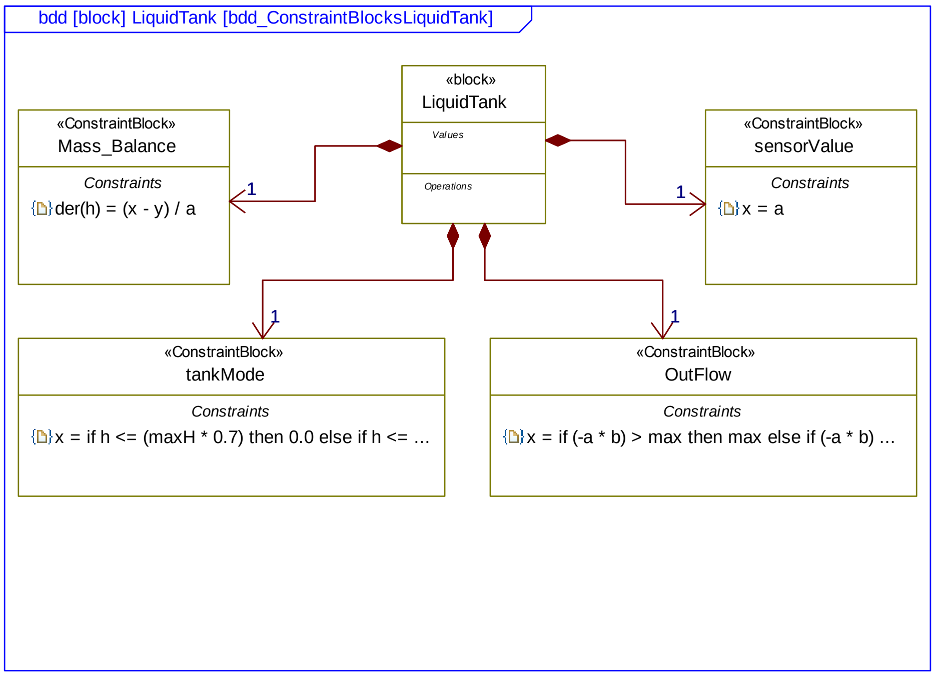


Figure 4-20 BDD Constraint Blocks of Liquid Tank

- The “sensorValue” constraint passes the tank level “h” value to the flow port which is connected to the “PIDcontinuousController”.
- The “MassBalance” constraint describes how the tank level “h” is calculated. This is a special equation, since the “der(x)” operator is special in Modelica.
- The “tankMode” constraint will assign the model of a tank in relation on its level h.
- The “OutFlow” constraint defines the value of the out flow level depending on the result of the controller. The result will be checked to be in a value range of minimum and maximum.

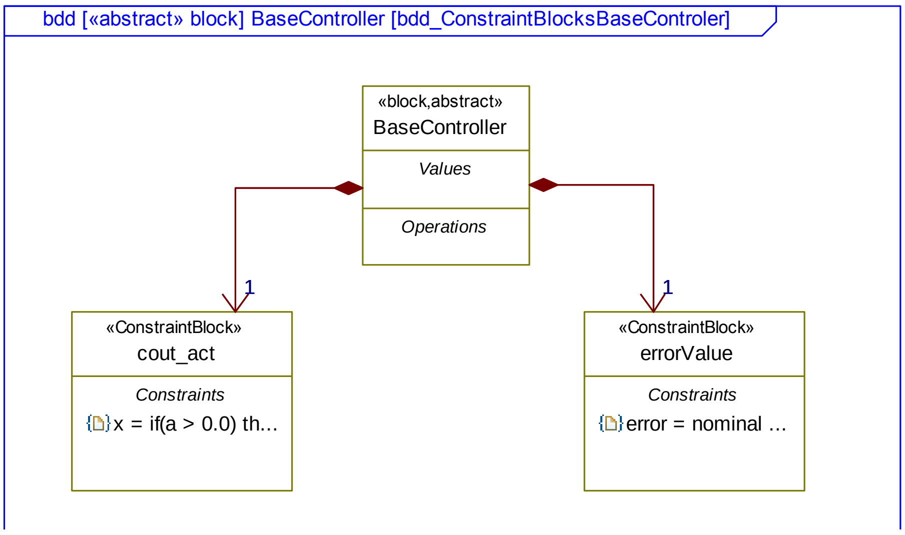


Figure 4-21 BDD Constraint Blocks of BaseController

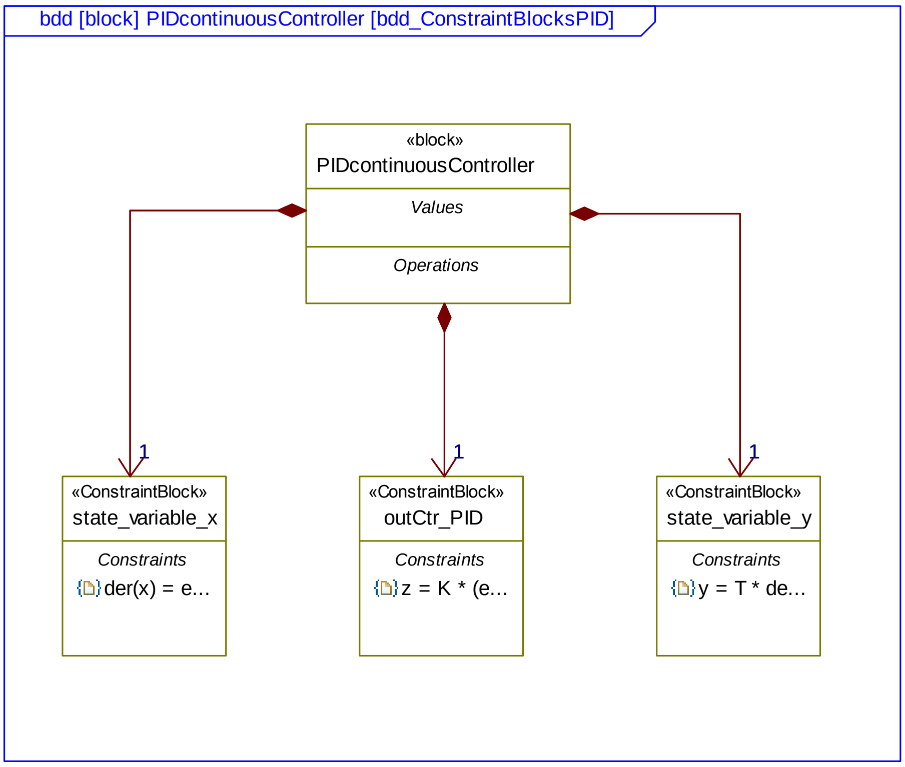


Figure 4-22 BDD Constraint Blocks of PID Continuous Controller

The constraints of the BaseController and the PIDcontinuousController belong together, since the PIDcontinuousController inherits its structure and behaviour from BaseController.

The constraints are very specific to control systems, therefore their equations won't be discussed future, but can be inspected in the [43].

4.2.5. **Parametric Diagrams of the parametric structure**

As mentioned in chapter 4.2.4 a constraint parameter needs to be linked to model elements in order to bring them in context. The SysML parametric diagram shows, as constraints represented, mathematical expressions in context to the designed system elements. Therefore parametric diagrams cannot exist in isolation. The result in Modelica will be the actual equation expression used in an equation section. Another very useful aspect of using constraints and parametric diagrams, instead of UML opaque expression, is the fact that a Modelica model has to be balanced with regard to the number of unknown variables and equations [4.7 Balanced Models]. A parametric diagram can easily be evaluated in order to determine missing links between a variables and constraints.

The following diagrams represent the relationship of constraint parameters of design model elements, such as attributes or flow ports. The translation into Modelica code will be done as described in chapter 4.1.3 with respect to the Modelica equation syntax and semantic ([23], 8).

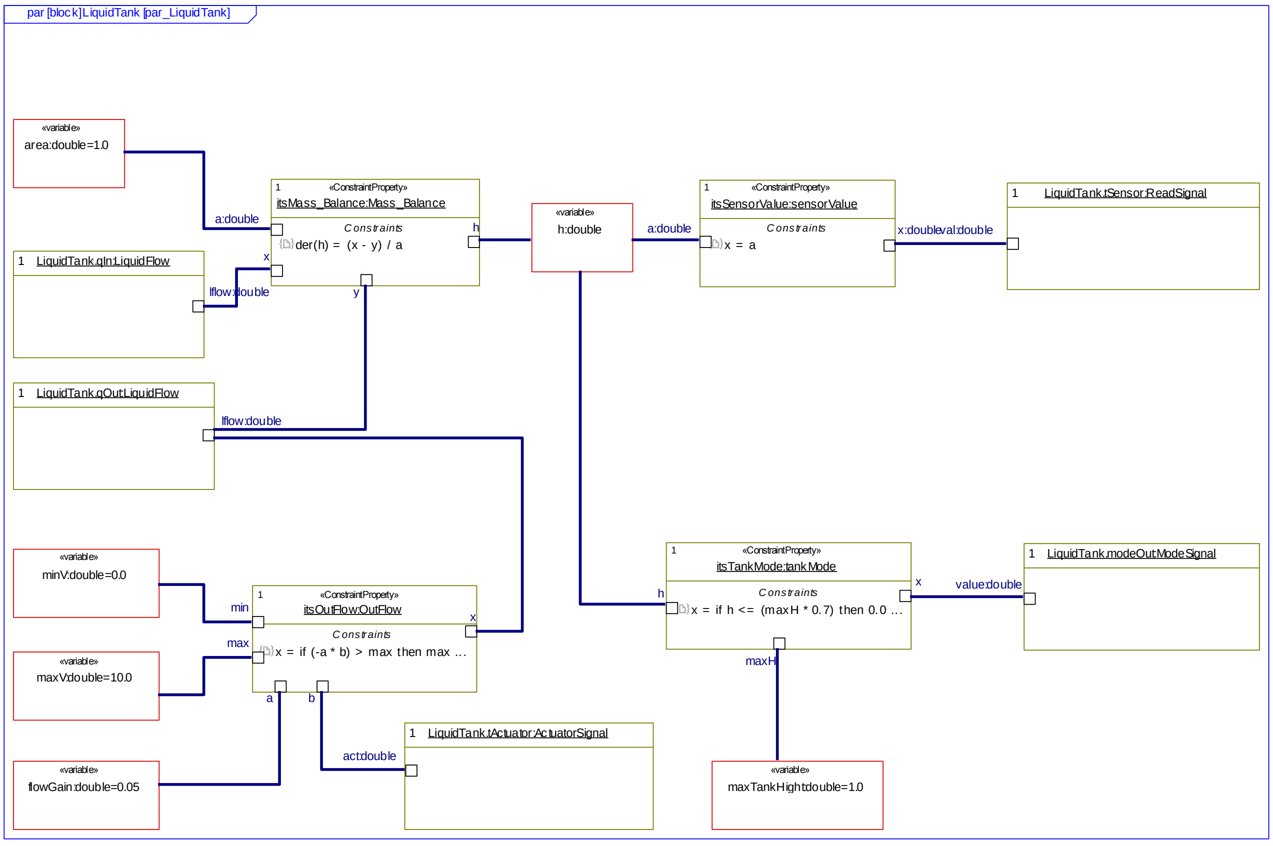


Figure 4-23 PAR Liquid Tank

In Figure 4-23 you can see all constraints and their parameters in relation to design model elements, such as block attributes or flow port attributes.

As mentioned in chapter 4.2.4 the constraint property “Mass_Balance” describes a special equation type. As depicted in Figure 4-23 the derivative of “h” will be allocated to the model attribute “h”, but the result of a differential calculation of a value is different to its real value.

However, this is possible because expressed as a Modelica equation the result of “der(h)” will be integrated automatically to get the value for “h”, so there is no need for a separate equation for this.

Resultant Modelica code using rule 3.1 – 3.5:

LiquidTank.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

block LiquidTank
...
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;
  modeOut.value = if h <= (maxTankHight * 0.7) then 0.0
                  else if h <= (maxTankHight * 0.8) then 1.0
                  else if h <= (maxTankHight * 0.95) then 2.0
                  else if h > (maxTankHight * 0.95) then 3.0
                  else -1.0;
end LiquidTank;
```

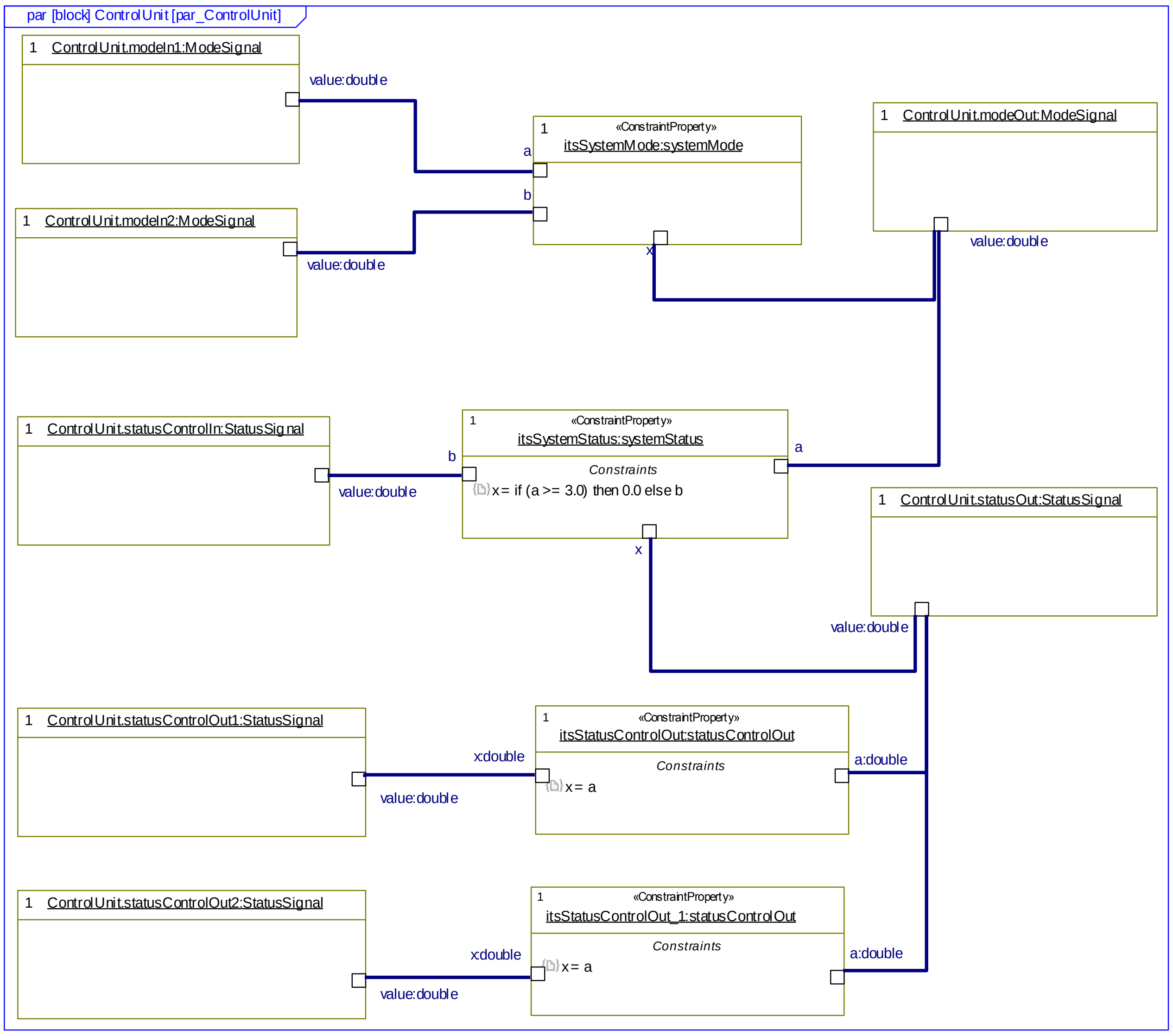

par [block] ControlUnit [par_ControlUnit]
1 ControlUnit.modeIn1:ModeSignal
value:double
1 ControlUnit.modeIn2:ModeSignal
value:double
1 «ConstraintProperty» itsSystemMode:systemMode
a
b
x
1 ControlUnit.modeOut:ModeSignal
value:double
1 ControlUnit.statusControlIn:StatusSignal
value:double
1 «ConstraintProperty» itsSystemStatus:systemStatus
Constraints
x= if (a >= 3.0) then 0.0 else b
a
b
x
1 ControlUnit.statusOut:StatusSignal
value:double
1 ControlUnit.statusControlOut1:StatusSignal
value:double
x:double
1 «ConstraintProperty» itsStatusControlOut:statusControlOut
Constraints
x= a
a:double
1 ControlUnit.statusControlOut2:StatusSignal
value:double
x:double
1 «ConstraintProperty» itsStatusControlOut_1:statusControlOut
Constraints
x= a
a:double

Figure 4-24 PAR Control Unit

Resultant Modelica code using rule 3.1 – 3.5:

ControlUnit.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

block ControlUnit
...
equation
  modeOut.value = if (modeIn1.value >= modeIn2.value) then modeIn1.value
                  else modeIn2.value;
  statusOut.value = if (modeOut.value >= 3.0) then false else
statusControlIn.value;
    statusControlOut1.value = statusOut.value;
    statusControlOut2.value = statusOut.value;
end ControlUnit;
```

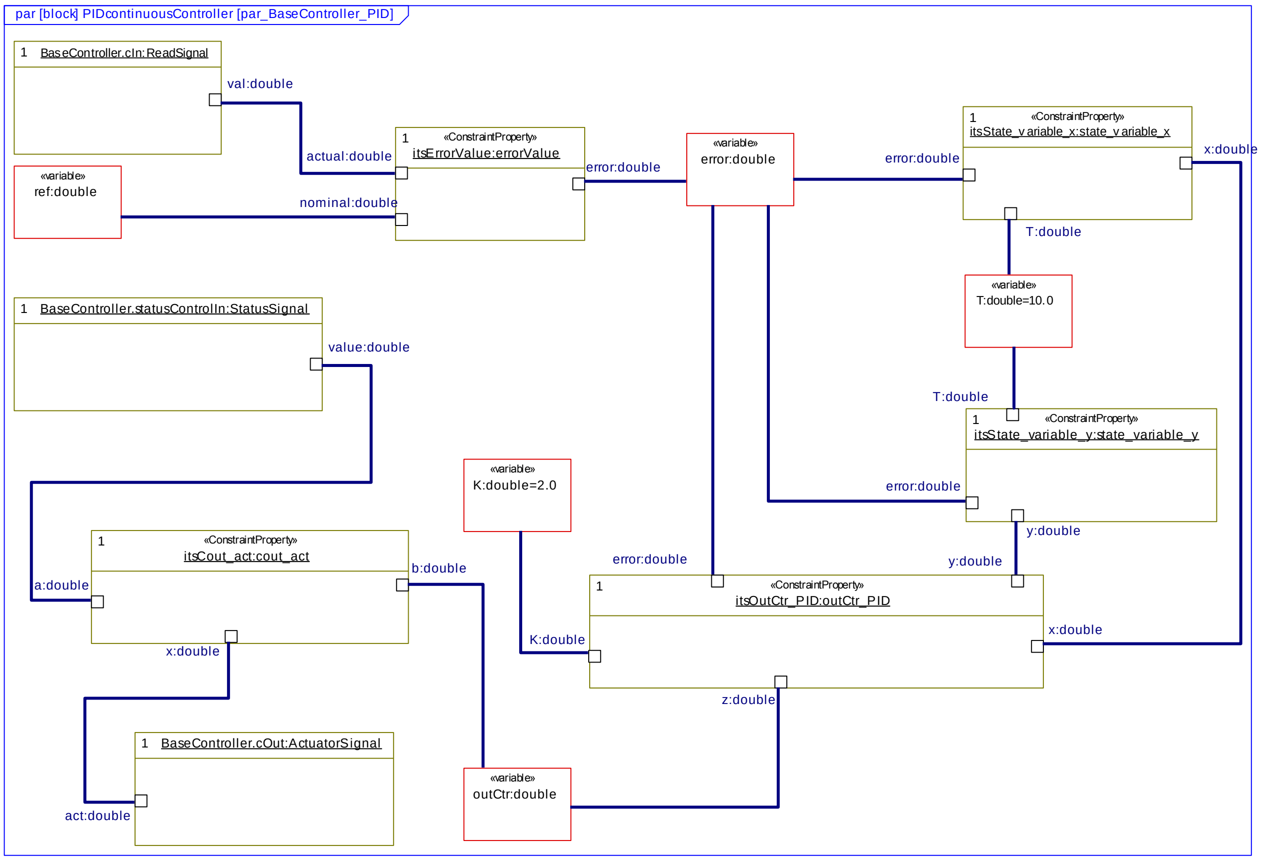

par [block] PIDcontinuousController [par_BaseController_PID]
1 BaseController.cIn:ReadSignal
val:double
actual:double
«variable» ref:double
nominal:double
1 «ConstraintProperty» itsErrorValue:errorValue
error:double
«variable» error:double
error:double
1 «ConstraintProperty» itsState_variable_x:state_variable_x
x:double
T:double
«variable» T:double=10.0
T:double
1 «ConstraintProperty» itsState_variable_y:state_variable_y
error:double
y:double
1 BaseController.statusControlIn:StatusSignal
value:double
«variable» K:double=2.0
1 «ConstraintProperty» itsCout_act:cout_act
a:double
b:double
x:double
error:double
y:double
1 «ConstraintProperty» itsOutCtr_PID:outCtr_PID
K:double
x:double
z:double
1 BaseController.cOut:ActuatorSignal
act:double
«variable» outCtr:double

Figure 4-25 PAR BaseControler and PIDcontinuousController

As mentioned in chapter 4.2.4 the constraints of the “BaseController” and “PIDcontinuousController” are coupled, since a PID controller is using the base controller as its core functionality of regulating values to a reference level. Therefore one parametric diagram will be used to illustrate their relationship.

<u>Resultant Modelica code using rule 3.1 – 3.5:</u>

BaseController.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

partial block BaseController
...
equation
  error = ref - cIn.val;
  cOut.act = if(statusControlIn.value) then outCtr else 0.0;
end BaseController;
```

PIDcontinuousController.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

block PIDcontinuousController extends BaseController(K = 2, T = 10);
...
equation
  der(x) = error/T;
  y = T*der(error);
  outCtr = K*(error + x + y);
end PIDcontinuousController;
```

The whole Modelica model code can be found in the Appendix A.

4.3. Automated Modelica Code Generation

Having a transformation approach which can be used to generate code for system components automatically is the key feature to its successful application in practice [20].

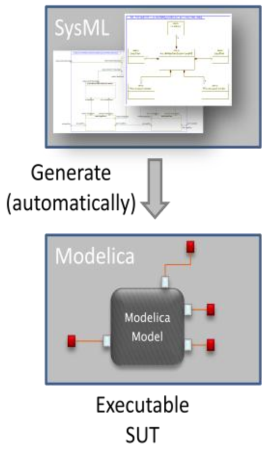


Figure 4-26 Automated Code Generation as a Key Feature for the Application in Practice

Since the transformation presented in 4.1 is self-contained, regarding the used SysML and Modelica language elements, this feature is given. So a full (100%) transformation from one language into the other is possible.

Since it is possible to present a model in the different levels of its implementation [9], the model, whether implemented in SysML or Modelica, mustn't be complete, regarding its components or the component functions. This means that a code generator won't force a developer to implement the model completely, but consistently. And it is also possible to translate just parts of a model, as longs as the related elements are available and consistent in the level of implementation.

By using IBM Rational Rhapsody 7.6, one can adapt and enhance an existing code generator [45], by implementing the transformation rules given in 4.2. Rhapsody allows generating code for the following elements:

- Entire configuration
- Several components
- Entire project
- Selected classes

But in general any SysML modelling tool with code generation capabilities can be used, for example the eclipse based open-source modelling tool Papyrus [46] in combination with Acceleo [47]. Acceleo is also an open-source software tool based on eclipse. A user can easily implement transformation rules using a special script language or the programming language Java [48].

5. Modelica as Test Specification and Implementation Language

Testing is part of the verification and validation process, since its purpose is to confirm parts of the design or implementation against the system specifications [3]. In general testing will help to improve quality, since it detects failures to minimize existing faults. In the context of dynamic systems, testing is an activity in which a system or component is executed under specified conditions. The results are observed or recorded, and an evaluation is made of some aspect of the system or component [13]. Tests are implemented as test cases. A test case consists of a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to verify compliance with a specific requirement [13].

A test specification language will be used to define tests in a more formal way, which is computable. In addition a test implementation language will be used to make the specified tests executable using a programming language. In the case of software it makes sense to implement the tests in the same language as the application is written. This will prevent the implementation of interfaces to ensure that the application and the test system understand each other. As an instance, in case of Java applications the test cases will be implemented in JUnit, a Java based test specification and implementation language [50]. Since the design model is implemented in the programming language Modelica, it makes sense to have a Modelica based test specification and implementation language.

5.1. *Modelica and TTCN-3*

Using Modelica as a test language assumes a test concept included in Modelica. So why not adopt such a concept from existing or standardised languages such as JUnit or the Testing and Test Control Notation (TTCN-3)? JUnit is specialized for testing Java based applications. In contrast TTCN-3 is a widely used, international standard language for testing software and hardware. In addition JUnit is more suitable for the low-level tests, such as component or integration level testing, TTCN-3 is more suitable for high-level testing [31]. This fits into the scope of this work.

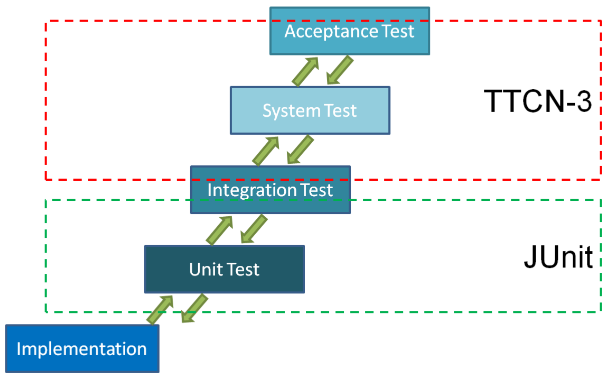


Figure 5-1 JUnit and TTCN-3 in relation to the V-Model XT

However, TTCN-3 is domain specific, since it has its roots in testing hardware and software components of IT and telecommunication systems. It is rather more suitable to be used with software and communication protocols than model-based systems. Modelica could fill the gap of having a model-based language especially for system testing.

This chapter shows an approach of adopting some TTCN-3 Core Language (CL) concepts and elements into Modelica, which are needed in the terms of testing. This is done by an informal mapping, more than adopting, of TTCN-3 CL concepts and elements.

5.2. TTCN-3 Core Language Definitions and Concepts

For the purposes of the present TTCN-3 (CL) concepts which have to be adapted to Modelica, the following terms and definitions given in ITU-T Recommendation X.290 [21], ITU-T Recommendation X.292 [22] and the ETSI ES 201 873-1 V4.3.1 [16] apply:

5.2.1. **TTCN-3 Built-In Data Types and Values**

TTCN-3 provides a set of basic types and values, like many other classical programming languages. A basic type does have predefined features, a specific usage and range.

Simple Basic Types

TTCN-3 provides the following simple basic types [6.1.0 Simple basic types and values]:

Name	Description
integer	A type with distinguished values which are the positive and negative whole numbers, including zero. Values of type integer can be arbitrarily large. Values of integer type shall be denoted by one or more digits; the first digit shall not be zero unless the value is 0.
Example	
`var integer v_number = 1;`	

Table 19 TTCN-3 Basic Type Integer

Name	Description
float	A type to describe floating-point numbers and special float values. In TTCN-3, the floating-point number value notation is restricted to a base with the value of 10.
Example	
`var float v_fp = 2.0;`	

Table 20 TTCN-3 Basic Type Float

Name	Description
boolean	A type consisting of two distinguished values. TTCN-3 has a genuine boolean built-in type, which can assume the two truth values "*true*" and "*false*". The Boolean operators "*and*", "*or*", "*xor*", and "*not*" can be used to form Boolean expressions.
Example	
`var boolean v_isActive = true;`	

Table 21 TTCN-3 Basic Type Boolean

Name	Description
verdicttype	TTCN-3 has a type to represent the possible outcomes – verdicts – of a test case, which is called verdicttype. A type for use with test verdicts consisting of five distinguished values. Values of verdicttype shall be denoted by "*pass", "fail", "inconc", "none"* and "*error"*.
Example	
`var verdicttype v_localVerdict;`	

Table 22 TTCN-3 Basic Type Verdicttype

User-Defined Structured Types

TTCN-3 provides the following simple basic types [6.1.0 Simple basic types and values]:

Name	Description
record	TTCN-3 supports ordered structured types in general known as record. Records can be used to group related fields into a single type. The data types used as fields in a record may be any of the basic types or user-defined data types. The values of a record shall be compatible with the types of the record fields.
Example	
`type record MyRecordType { integer id, boolean active }; var MyRecordType v_process := {4711, true};`	

```
type record MyRecordType
{
      integer id,
      boolean active
};
var MyRecordType v_process := {4711, true};
```

Table 23 TTCN-3 Structured Type Record

5.2.2. **TTCN-3 Test Structure Definitions**

The following TTCN-3 language elements and definitions shall be used to specify a test structure.

Name	Description
Module	A module is a top-level element containing all other elements and the main definitions of test behaviour. All TTCN-3 code must be specified within a *module*. A module is defined by the keyword "*module"* followed by a unique name. It contains a definitions part and an optional control part. The definition part defines the test cases and test components. The control part of a module executes the test cases.

Table 24 TTCN-3 Structure Definition: Module

Name	Description
System under Test (SUT)	A real open system which is to be studied by testing. In this example the Aircraft Water Tank System is the System under Test.

Table 25 TTCN-3 Structure Definition: System under Test (SUT)

Name	Description
Main Test Component (MTC)	Single Test Component in a Test Component Configuration responsible for creating and controlling Parallel Test Components and computing and assigning the test verdict. In this example the User Interface is the main test component.

Table 26 TTCN-3 Structure Definition: Main Test Component

Name	Description
Parallel Test Component (PTC)	Test component created by the main test component.

Table 27 TTCN-3 Structure Definition: Parallel Test Component

Name	Description
Test System Interface (TSI)	The (abstract) interface of the test system towards the SUT. In our example the MTC is the only test component communicating with the SUT. The component configurations will completely be defined by the ports of the main test component. Therefore, there is no need to define the test system interface separately. In addition a TTCN-3 abstract TSI is used to communicate over a protocol without implementing the concrete communication language.

Table 28 TTCN-3 Structure Definition: Test System Interface

Name	Description
Test Configuration	Ports of the test system interface and ports of components need to be connected together in order to provide communication. TTCN-3 allows to connect test component ports to other test component ports and to map a test component port to a port of the test system interface.

Table 29 TTCN-3 Structure Definition: Test Configuration

Name	Description
Port Mappings	A port of a test component is mapped to a port of the test system interface by using the map operation. This is part of the test configuration.

Table 30 TTCN-3 Structure Definition: Part Mappings

Name	Description
Port Connections	The ports of test components can be connected directly to exchange messages between the two test components. However, whereas in the map operation one of the component references must be system, in a connect statement both references are referring to test components and not to the test system interface. This is part of the test configuration.

Table 31 TTCN-3 Structure Definition: Port Connections

5.2.3. **TTCN-3 Test Behaviour Definitions**

The following TTCN-3 (CL) elements and concepts specify the behaviour of tests.

Name	Description
Test Case	A test case is a behaviour description of how to stimulate the SUT using inputs and the expected reactions of the SUT to this stimulations using observing the outputs. A test case is also condition dependant (precondition, post condition). Depending on the reactions, a verdict can be assigned. For example, a test case can pass or fail.

Table 32 TTCN-3 Behaviour Definition: Test Case

Name	Description
Module Control Part	The control part of a module executes test cases and controls their behaviour from outside. The control part may also declare (local) variables. Control statements such as if-else or do-while may be used to specify the selection and execution order of test cases.

Table 33 TTCN-3 Behaviour Definition: Module Control Part

Name	Description
Test Case Execution	After defining a test case in the definitions part of a module one can execute the test case using the "*execute()*" operation in the control part. The test case will be executed with actual parameters. Program statements can be used to specify the selection and execution order of the test cases. A test case execution returns a test verdict.

Table 34 TTCN-3 Behaviour Definition: Test Execution

Name	Description
Test Verdict	Each test component has an implicitly defined variable of type verdicttype. This implicit variable is called the "*local verdict"* of a test component. Per default its value is "*none".* In addition to the local verdicts of the test components, there is the "*overall verdict*" of the test case which will be returned after it terminates.

Table 35 TTCN-3 Behaviour Definition: Test Verdict (local and overall)

For more details about TTCN-3 CL please see [16].

5.3. Modelica4Testing: A Test Model Framework for Modelica

In order to adopt TTCN-3 CL concepts into Modelica the presented language elements from Chapter 5.2 will be compared and adapted to available standard Modelica v3.2 elements. This will result in a test specification and implementation language, future stated as Modelica4Testing (M4T). Modelica4Testing can be used as a test model framework. In this work the term model framework can be described as follow: A model framework is an abstraction in which generic functionality and structure can be enhanced by user and domain specific elements. It is a collection of libraries providing reusable and adaptable elements. However the semantic given by a model framework is not alterable in any case, since its semantic is expressed in this way and it shall be used as a guideline.

The Modelica4Testing framework bases on the Modelica meta-model without any enhancements or modifications. Meta modelling is a key technology to support the understanding and usage of modelling languages, since it allows defining a more abstract view on a model [17]. This includes for example the structure of the information been modelled. Figure 5-2 presents the different layers used in the context of Modelica and Modelica4Testing.

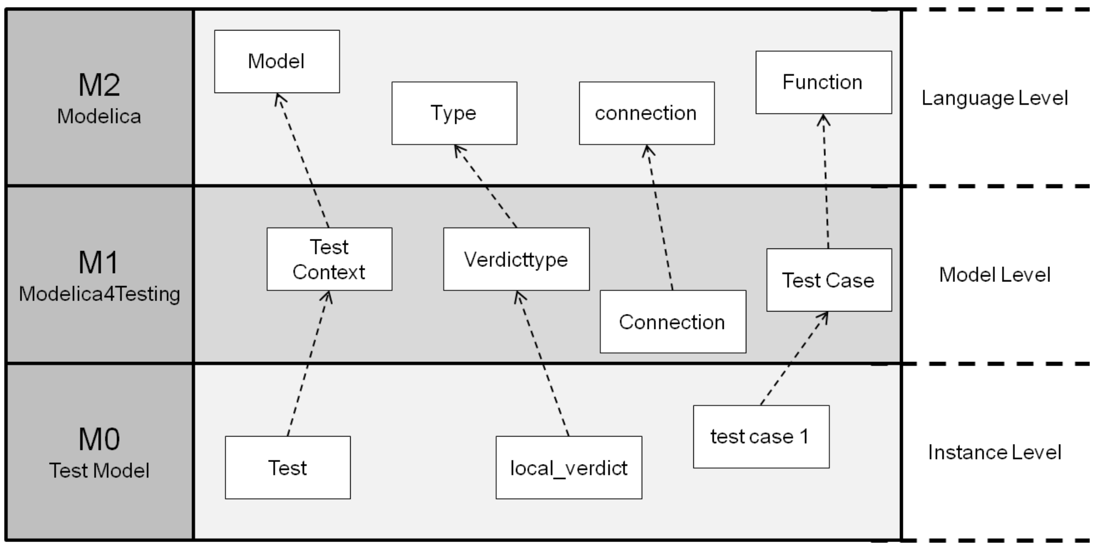


Figure 5-2 Modelica and Modelica4Testing Modelling Layers

Modelica4Testing is located in the model level (M1) since it bases on the Modelica meta-model (M2) and it describes the test model instance (M0). However M4T shall be used as a test

specification and implementation language, furthermore as a test model framework, it has also its own ontology to be used in the context of model-based system tests with Modelica.
For more details about the used Modelica elements please see [23].

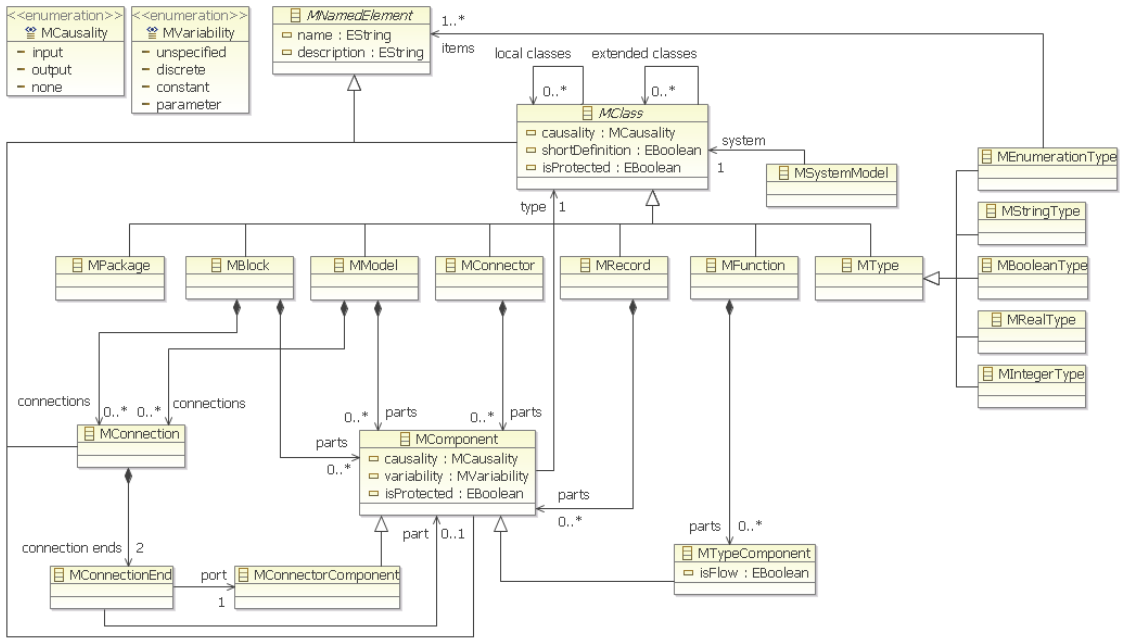


Figure 5-3 MathModelica Simple Modelica Meta-Model

A simple Modelica meta-model is presented in the Figure 5-3. However, as mentioned above, Modelica4Testing has an additional meta-model, but is based on the standard Modelica meta-model without exceptions. The elements specified in Modelica4Testing are neither special languages elements or keywords nor restricted classes in Modelica.

The following chapters compare available standard Modelica elements to be matched with the TTCN-3 Core Language concepts, presented in Chapter 5.2. The principal approach towards the mapping to TTCN-3 consists of one major step:

- Take Modelica elements and associations and assign them to TTCN-3 CL testing concepts.

5.3.1. **Modelica4Testing Test Structure Definitions**

The test structure section contains the concepts needed to describe the elements which defined a test.

Test Model

Since we are in the context of model-based systems engineering the definition of "test model" will be used to represent the overall Modelica system model, which is used to realize model testing [24]. This system model contains all model elements, such as the design-model and test components.

Name	(Short) Description
Test Model	The main Modelica system model. Contains all system model elements.
Note	This is an implicit language element.

Table 36 Modelica4Testing Test Model

Test Context

A test context is the top-level element of a test model, similar to a TTCN-3 Module. A test context shall be implemented as singleton in a test model. It is the overall control unit which controls and observes all test model elements (Test Cases, SUT, MTC, etc.) as well as the simulation and its results. In addition a test configuration will be defined within a test context. The test context will be the top level class of the instantiation hierarchy. In the Modelica program this will be executed as a kind of "main" class that is always implicitly instantiated. To avoid malapropism the term "test context" has been adopted from the UML2 Testing Profile, since Modelica has a very similar calling language element to the TTCN-3 "module", called "model".

Name	(Short) Description
Test Context	Top-level element of a test model. Controls and observes all test elements, such as test cases, the SUT and test components.
TTCN-3 Element	Module (see Table 24)
Modelica4Testing Type	TestContext
Modelica Type	Model
Multiplicity	1, A test model contains just one test context.
isAbstract	False
Note	By default this element is incomplete, but defines a general structure. The user must complete the body with domain and test specific data.

Table 37 Modelica4Testing Test Context

Design Model

A model which represents a real physical system and is to be studied by testing shall be used in the system under test. The design model (DM) offers inputs to be stimulated and outputs to be observed, in order to interact with test components over a specified test system interface (Table 28). In addition an output is used by a test case to observe the reactions of the design model. However, there is no direct connection to the test system, since the SUT (Table 25) is used as the design model interface.

Name	(Short) Description
Design Model	System model which is to be studied by testing.
TTCN-3 Element	SUT (see Table 25)
Modelica4Testing Type	DesignModel
Modelica Type	Class
Multiplicity	1, A test model contains just one design model.
isAbstract	False
Note	By default this element is not defined. The user has to define it.

Table 38 Modelica4Testing Design Model

Test System Interface

Since test components are not allowed to communicate directly with a design model, there is a need for an interface. A test system interface is the link between the test system and the system under test. A test system interface configuration connects test dependent ports together.
To avoid overloading of model elements the TSI configuration can be implemented within the SUT. But thinking of an abstract interface to communicate with real physical devices, it is also possible to implement the TSI as a separate component of the test model. This scenario will not be described future more in this work. The TSI offers the same inputs and outputs as the design model.

Name	(Short) Description
Test System Interface	The interface of a test system towards the SUT, respectively design model.
Note	This is an implicit language element. An abstract TSI element must be implemented separately.

Table 39 Modelica4Testing Test System Interface

System under Test (SUT)

A system under test is used as a wrapper over the real design model. All communication between the design model and the test system will happen via this component. The SUT offers the same inputs and outputs as the design model.

Name	(Short) Description
System under Test	System model which is to be studied by testing.
TTCN-3 Element	SUT (see Table 25), TSI (see Table 28)
Modelica4Testing Type	SUT
Modelica Type	Model
Multiplicity	1, A test model contains just one SUT.
isAbstract	False
Note	By default this element is incomplete, but defines a general structure. The user must complete the body with domain and test specific data.

Table 40 Modelica4Testing SUT

The figure below represents the relationship between the design model, the TSI and the SUT.

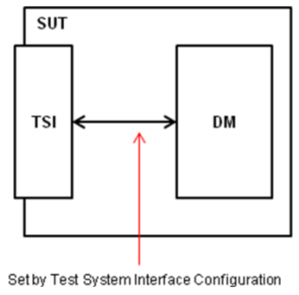


Figure 5-4 Modelica4Testing Overview of SUT and its Components

Test System

A test system represents test components which interact with a SUT. A test component emulates an actor or an external system. The level of implementation can be list as follow:

- Dummy: A very rudimentary implementation with no functionality or intelligence. In most cases a dummy is just used to proved ports and to pass data or messages to the SUT.
- Stub: Provides functionality for testing. It may be implemented for a special test, so it provides the exact reactions expected when getting results from a SUT.
- Mock Object: Has some intelligence implemented, in order to react more autonomous, so it may emulate a more complex system or actor.

When thinking of integration level testing, component or system integration, a test component can also be another design model, providing full functionality. However, this is not a recommended strategy, since the design model used as a test component may has its own failures, and this will lead the results of the test in the wrong direction.

The test system interface is the interaction point used by test components to communicate with the SUT.

Name	(Short) Description
Test System	Represents test components which interact with a SUT.
Note	This is an implicit language element.

Table 41 Modelica4Testing Test System

Test Component

Test components interact with the SUT, and will be controlled and configured by a test case. It realizes the behaviour defined by a test case. Modelica4Testing offers two kinds of test components, one main test component (MTC) and zero or many parallel test components (PTC). A local verdict represents the state of the component and is observed by a test case. Since this is a general behaviour of any test component this can be implemented as an abstract language element of M4T.

Name	(Short) Description
Test Component	Actor or system which interacts with a SUT in order to stimulate it.
TTCN-3 Element	Test Component (see Table 26, Table 27)
Modelica4Testing Type	TestComponent
Modelica Type	Class
Multiplicity	1…*, A test model contains one MTC and zero or many PTC.
isAbstract	True
Note	This element will be implemented as an abstract class. The user has to define a specific test component element as MTC or PTC.

Table 42 Modelica4Testing Test Component

Main Test Component

A main test component (MTC) is the only test component which interacts directly with the SUT, in terms of controlling it. Therefore the MTC will be implemented separately. The MTC controls one or many parallel test components, whereas itself will be controlled and configured by a test case. A MTC inherits its general test component behaviour from the abstract Test Component class.

Name	(Short) Description
Main Test Component	Actor or system which interacts with a SUT in order to stimulate it.
TTCN-3 Element	MTC (see Table 26)
Modelica4Testing Type	MTC
Modelica Type	Model
Multiplicity	1, A test model contains one MTC.
isAbstract	False
Note	By default this element is not defined. The user has to define it at least as a dummy.

Table 43 Modelica4Testing MTC

Parallel Test Component

A parallel test component may only exchanges data or objects with a SUT. It will not control the SUT directly. A PTC will be controlled by the MTC and not by a test case. A PTC inherits its general test component behaviour from the abstract Test Component class.

Name	(Short) Description
Parallel Test Component	Actor or system which may exchange data with a SUT.
TTCN-3 Element	PTC (see Table 27)
Modelica4Testing Type	PTC
Modelica Type	Model
Multiplicity	0…*, A test model contains zero or many PTC.
isAbstract	False
Note	By default this element is not defined. The user has to define it at least as a dummy.

Table 44 Modelica4Testing PTC

Test Configuration

A port is an interaction point used by the SUT or test components to interact with their environment. In order to connect ports of a test system interface or test component to available SUT ports, or test component ports amongst themselves, a user may define an individual port configuration within a test context. Unlike software systems, real physical systems do not allow a dynamic connection of ports at runtime, so this behaviour is not given in Modelica at all.

Name	(Short) Description
Test System	Represents a user defined configuration of interaction points.
Note	This is an implicit language element.

Table 45 Modelica4Testing Test Configuration

The Figure 5-5 shows the test system its components and the test configuration between the test system and the SUT.

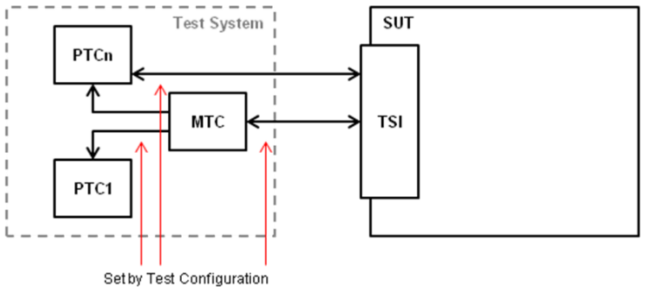


Figure 5-5 Modelica4Testing Overview of Test System Configuration

Port Connection

TTCN-3 allows to “connect” a test component port to another test component port and to “map” a test component port to a port of the test system interface or SUT as mentioned in Table 31. Since “connect” and “map” will have the same result, in M4T one element represents this concept. A SUT or test component must have at least one Modelica connector component, since a model element without any connectors can’t interact with its environment, and as a consequence it can’t be used in a Modelica test. A M4T port connection is a Modelica connection equation class, so it has exact two connection ends, as shown in Figure 5-3.

Name	(Short) Description
Port Connection	Actor or system which may exchange data with a SUT.
TTCN-3 Element	Port Connection (see Table 31), Port Mapping (see Table 30)
Modelica4Testing Type	PortConnection
Modelica Type	Connection Equation
Multiplicity	1…*, A SUT or test component contains one or many port connections.
isAbstract	False
Note	By default this element is not defined. The user has to define it.

Table 46 Modelica4Testing Port Connection

5.3.2. **Modelica4Testing Test Behaviour**

The test behaviour section contains concepts to specify the behaviour of tests.

Test Case

A test case is a behaviour description of how to stimulate a SUT using its inputs and the expected reaction of the SUT by observing the outputs. A test case is also condition dependent (precondition, post-condition). Unlike TTCN-3 test cases a Modelica4Testing test case can't be implemented as an operation which simply stimulates and observes the SUT sequentially and simultaneously at runtime, since a system in Modelica is simulated over time, and most models have time dependent behaviour [23]. Therefore a M4T test case consists of two parts. The first part will stimulate the MTC in order to pass this stimulus to the SUT and other test components, while the second part observes their reactions. In addition a test case has a verdict variable and also an optional test objective as a description, these variables shall be implemented in the test context definitions part. The local verdict of the test case will be assigned using the SUT reactions and verdict, as well as the verdict of test components.

Name	(Short) Description
Test Case	A set of test inputs, execution conditions, and expected results developed to confirm an implementation against its requirements.
TTCN-3 Element	Test Case (see Table 32)
Modelica4Testing Type	TestCase
Modelica Type	None – since implicit –
Multiplicity	1…*, A test context contains one or many test cases.
isAbstract	True – in the context of an implicit language element –
Note	This is an implicit language element, since it will be represented by a stimulation part and an observation part.

Table 47 Modelica4Testing Test Case

The Figure 5-6 describes the test case parts and the involved elements, as well as the data flow direction to stimulate and to evaluate the elements.

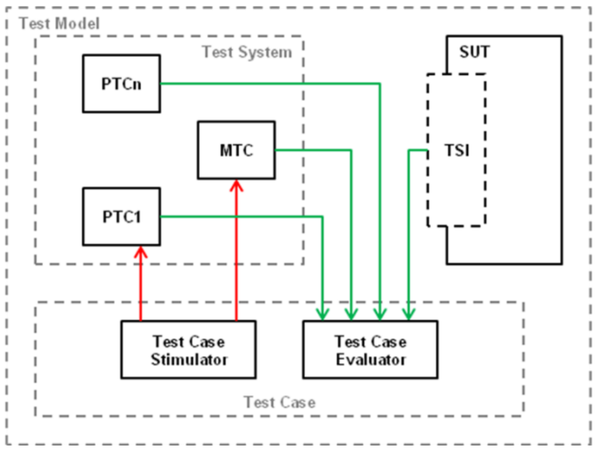


Figure 5-6 Modelica4Testing Overview of Test Case and its Stimulation and Observation Parts

Test Case Stimulator

This element will describe the concrete behaviour of a test case by stimulating the MTC in order to pass these stimuli to the SUT or test components at a specified simulation time or condition. But prior it checks if the test model is confirming specified preconditions, for example by checking the actual state of a SUT. This check will result in a local verdict variable. A stimulator should be aborted early if the verdict returns a bad result (inconc, fail, error, see 5.3.3). In this case the observation part should not be executed, since its preconditions are not satisfied.

Name	(Short) Description
Test Case Stimulator	Stimulates the SUT and the test components.
TTCN-3 Element	Test Case Behaviour (see Table 34)
Modelica4Testing Type	TestCaseStimulator
Modelica Type	Class
Multiplicity	1, A test case consists of one stimulator part.
isAbstract	True
Note	By default this element is incomplete, but defines a general structure. The user must complete the body with model and test specific data.

Table 48 Modelica4Testing Test Case Stimulator

Test Case Stimulator Function

The actual stimulation part will be implemented as a Modelica function using its algorithm section, see (Chapter 12, [23]) Although equations are eminently suitable for modeling physical systems, there are situations where non-declarative algorithmic constructs are needed. This is typically the case for algorithms, i.e., procedural descriptions of how to carry out specific computations, usually consisting of a number of statements that should be executed in the specified order. In Modelica, algorithmic statements can occur only within algorithm sections, starting with the keyword algorithm. It may not contain any condition statements, since the behaviour of a test case and its usage should be determined before its start time. The Modelica function return-statement will be used to terminate the stimulator function if the precondition verdict fails [Modelica]. As a feature the actual stimulation part body can be empty, to support just the observation of system behaviour without manipulating it.

Name	(Short) Description
Test Case Stimulator	Stimulates the SUT and the test components.
TTCN-3 Element	Test Case Behaviour (see Table 34)
Modelica4Testing Type	TestCaseStimulatorFunction
Modelica Type	Function
Multiplicity	1, A test case consists of one stimulator part.
isAbstract	True
Note	By default this element is incomplete, but defines a general structure. The user must complete the body with model and test specific data.

Table 49 Modelica4Testing Test Case Stimulator Function

Test Case Evaluator

After stimulating the MTC using its inputs, a set of test case dependent outputs will be observed, in order to evaluate the SUT, MTC and PTC reactions. Modelica4Testing supports two different evaluation methods, which are natural in the terms of system simulations over time. The reaction can be evaluated at a specified time (t_n) or during a specified time interval ($t_n – t_{n+m}$). The result of this evaluation will be returned as a local verdict, calculated using the verdict mechanism described in 5.3.3. In addition to the expected output values or states of an SUT, the local verdicts of all involved components will be requested. This covers the evaluation of a post condition, which is expected by a test case. As a feature, a test case can have multiplied evaluation parts, in order to check the system several times.

Name	(Short) Description
Test Case Stimulator	Evaluates the SUT and the test components.
TTCN-3 Element	Test Case Verdict Operation
Modelica4Testing Type	TestCaseEvaluator
Modelica Type	Model, Function
Multiplicity	1…*, A test case consists of one or many evaluation part.
isAbstract	True
Note	This element will be implemented as an abstract class. The user has to define a test case evaluator model or function. An evaluation process shall not have any side effects.

Table 50 Modelica4Testing Test Case Evaluator

Test Case Evaluator Function

In some cases an expected behaviour may occur and be evaluated at a specified time (t_n) of the simulation. In this case the expected output values of a SUT and the local verdicts of the test components at this time (t_n) can be evaluated, using a Modelica function. The actual evaluation part will be implemented as a Modelica algorithm section, see (Chapter 12, [23]). The Modelica function return-statement will be used to terminate the evaluator [Modelica].

Name	(Short) Description
Test Case Stimulator	Evaluates the SUT and the test components at a specified time (tn).
TTCN-3 Element	Part of the Test Case Verdict Operation
Modelica4Testing Type	TestCaseEvaluatorFunction
Modelica Type	Function
Multiplicity	1…*, A test case consists of one or many evaluation part.
isAbstract	False
Note	By default this element is incomplete, but defines a general structure. The user must complete the body with model and test specific data.

Table 51 Modelica4Testing Test Case Evaluator Function

Test Case Evaluator Model

Unlike expected behaviour at one specified time, a test case can expect behaviour during a time interval or without any knowledge of the actual occurrence time. In this case the evaluation will be implemented as a Modelica model. Its actual evaluation part takes place in a Modelica algorithm section or an equation section respectively. Since behaviour expressed in a Modelica model will be evaluated continuously after starting a test model, and the local verdict will only be returned at its termination, additional start (t_n) and stop (t_{n+m}) time variables will limit the execution time. The minimum start time (t_n) is the test case stimulus execution time. The termination time is at least one time step after the start time (t_{n+1}), but at an outside estimate one time step below the overall simulation stop time (t_{stop-1}), since the overall test case verdict value must also be evaluated from the top-level test context. In addition to the stop time an expected state can terminate the model execution early.

Name	(Short) Description
Test Case Stimulator	Evaluates the SUT and the test components during a specified time interval ($t_n – t_{n+m}$).
TTCN-3 Element	Part of the Test Case Verdict Operation
Modelica4Testing Type	TestCaseEvaluatorModel
Modelica Type	Model
Multiplicity	1…*, A test case consists of one or many evaluation part.
isAbstract	False
Note	By default this element is incomplete, but defines a general structure. The

	user must complete the body with model and test specific data.

Table 52 Modelica4Testing Test Case Evaluator Model

5.3.3. **Modeica4Testing Verdict Type and Values**

A verdict variable represents the possible outcomes of a test case, future stated as Verdicttype. A Verdicttype consists of five distinguished values. Values of Verdicttype shall be denoted by “pass”, “fail”, “inconc” (inconclusive), “none” and “error”. This type is implemented as a Modelica enumeration type. Since “fail” is a Modelica keyword the prefix “t_” has been attached to the values. Each test case, SUT and test component of the active configuration shall maintain its own local verdict, be collected at the top-level text context as an overall verdict.

Name	(Short) Description
Verdicttype	Represents the possible outcomes of a test case. Values shall be denoted by “t_none”, “t_pass”, “t_inconc”, “t_fail” and “t_error”.
TTCN-3 Element	Verdicttype
Modelica4Testing Type	Verdicttype
Modelica Type	Enumeration
isAbstract	False
Note	By default this element is complete and ready to be used as a variable type.

Table 53 Modelica4Testing Verdicttype

The following table will list the single possible verdict values and describe them shortly:

Verdicttype	Description
t_none	This is the default value at start time. When a SUT or test component is instantiated, its local verdict variable is set to the value none. This value should change at runtime otherwise this signals an unaffected behaviour.
t_pass	If everything goes correctly, this value will return as a verdict of test cases, test components, the system under test and finally the test context.
t_inconc	The inconc value means an inconclusive verdict.
t_fail	This signal is the opposite of pass. It will appear if a test case failed.
t_error	The error verdict is special in that it is set by the test system to indicate that a test case or component error has occurred. No other verdict value can override an error verdict. This means that an error verdict can only be a result of an execute test case operation.

Table 54 Modelica4Testing Verdicttype possible Values

5.3.4. **The Verdict Mechanism**

A verdict mechanism is used to assign and update the value of a verdict variable, in order to represent different levels of verdicts, such as an overall test context verdict or a local test case verdict.

Test Case Verdict

In order to evaluate a test case its local verdict is assigned collection the following data:

Name	(Short) Description
Precondition	Represents a required test model state, to run a test case correctly. The precondition check will run in the test case stimulus.
Involved Components	SUT, MTC and PTC
Involved Variable Type	local verdict
Impact if pass	Starts actual stimulation part of a TestCaseStimulator (Table 48).
Impact if none, inconc, fail or error	Terminates the whole TestCase early.

Table 55 Modelica4Testing Test Case Verdict Precondition

Name	(Short) Description
Precondition	Represents a required test model state after stimulating a component, in order to check unexpected behaviour. The post-condition check will run in the test case evaluator.
Involved Components	SUT, MTC and PTC
Involved Variable Type	local verdict
Impact if pass	Check expected behaviour of the SUT (Table 48).
Impact if none, inconc, fail or error	Terminates the TestCaseStimulator early, since the results are corrupt.

Table 56 Modelica4Testing Test Case Verdict Post-Condition

Name	(Short) Description
Evaluate Behaviour	Expected behaviour of a SUT in order to confirm a requirement. The behaviour check runs in the test case evaluator part.
Involved Components	SUT
Impact if pass or fail	Terminates a test case correctly.
Impact if inconc, or error	Terminates the whole TestCase early.

Table 57 Modelica4Testing Test Case Verdict SUT Behaviour

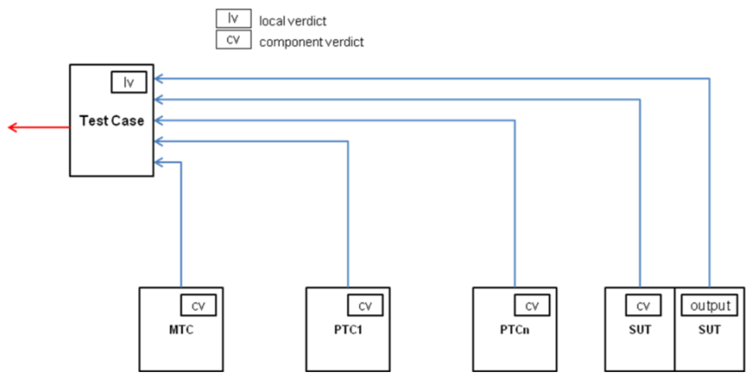


Figure 5-7 Modelica4Testing Verdict Mechanism to assign a local test case verdict

In Figure 5-7 all elements are depicted which are involved to assign the test case local verdict. As depicted in the figure above different verdict values can occur for a test case, so there is a need to prior the values. These overwriting rules are directly adopted from TCCN-3 CL (Chapter 24.1, [16]).

Current value of Verdict	New verdict assignment value			
	pass	inconc	fail	none
t_none	t_pass	t_inconc	t_fail	t_none
t_pass	t_pass	t_inconc	t_fail	t_pass
t_inconc	t_inconc	t_inconc	t_fail	t_inconc
t_fail	t_fail	t_fail	t_fail	t_fail
t_error	t_error	t_error	t_error	t_error

Table 58 Modelica4Testing Overwriting Rules for the Verdict

The actual assigning algorithm can be implemented using the Modelica reduction function max(A) (Chapter 10.3.4, [23]). The “max(A)” function returns the largest element of array expression “A”. This is useful since the verdict type is a Modelica enumeration and the ordinal number will represent the value. This requires requesting the verdict element ordinal number by using the Modelica type conversion of enumeration values into integer (Chapter 4.8.5.2, [23]). It returns the ordinal number of a verdict element. To confirm the overwriting rules from Table 58, the verdict enumeration elements order in Modelica must implemented as follow:

```
//test_none = 1, test_pass = 2, test_inconclusive = 3, test_fail  = 4, test_error = 5
type Verdicttype = enumeration(t_none, t_pass, t_inconc, t_fail, t_error);

max(i for i in {Integer(componentVerdict2),...,Integer(localVerdict)});
```

Table 59 Modelica4Testing Verdict Mechanism element order and implementation in Modelica

Test Context Verdict

- *Single Test Case*

Unlike software or service oriented tests, where many standalone test cases can simply be executed together without any side effects, system tests in Modelica are different, since a simulation is running overtime and changes in the past will affect the future. As a result not related test cases can only be implemented as standalone in a test context.

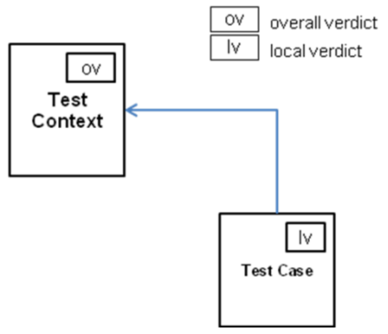


Figure 5-8 Modelica4Testing Assigning Overall Verdict: Single Test Case

- *Test Suite*

A test suite is a collection of test cases running together in order to speed up testing or use related behaviour in the same test run. As mentioned above it is not suitable to execute unrelated test cases sequentially, because of side effects and unexpected behaviour. This feature should be implemented by a test tool, since a tool can restart tests and execute tests in a loop and so on.

However as mentioned in section 0 a test case stimulator body can be empty, in order to use a observer test case without any stimulations of the SUT. In this case not related test cases can be collected together as a test suite. Figure 5-9 shows such a test suite implemented in a test context.

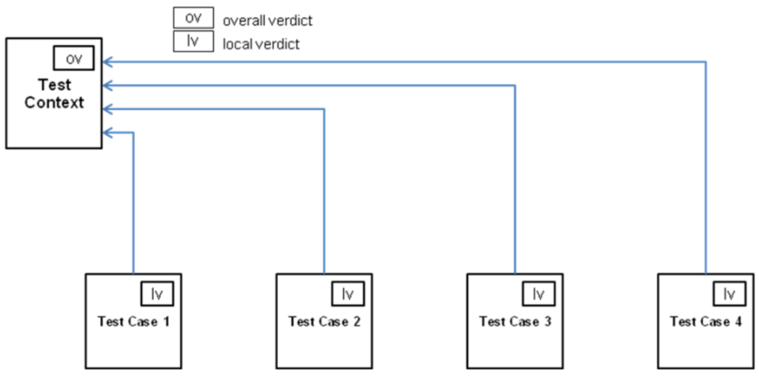


Figure 5-9 Modelica4Testing Assigning Overall Verdict: Test Suite with independent Test Cases

Far more interesting, is the situation to related test cases, as depicted in Figure 5-10. Related test cases can be executed sequentially in order to use the result of a previous test case as the precondition of a follower test case. However it should be noted, that in this constellation a failed test case will affect its follower test case negatively. The test run should be aborted.

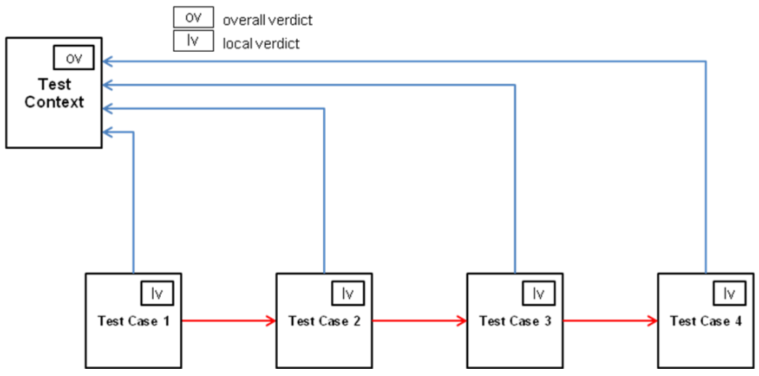


Figure 5-10 Modelica4Testing Assigning Overall Verdict: Test Suite with dependent Test Cases

5.4. Modelica4Testing Meta-Model and Scope

In order to bring all concepts from chapter 5.3 in a context and to illustrate the ontology of Modelica4Testing in an understandable, more visual way, one can use meta-models. In the context of this work ontology is the explicit specification of the Modelica4Testing conceptualisation. In software engineering and model-based engineering ontology is represented using a meta-model. A meta-model is an abstract view of data used in a domain. The Modelica4Testing meta-model will be illustrated in an “Ecore” diagram using the Eclipse Modelling Framework Technology (EMFT) [51]. The Ecore meta-model contains the information about the defined abstract classes.

The following elements can be represented within the Ecore model:

- The EClass element represents an abstract class, with zero or many attributes and zero or many references to other Ecore classes.
- The EAttribute element represents an attribute, its name and type.
- An EReference represents the association between two classes. Associations can also be specified as a containment relation.
- The EDataType elements allow creating user defined data types used with attributes. However, Ecore contains a set a very basic data types, such as integer, Boolean and float.

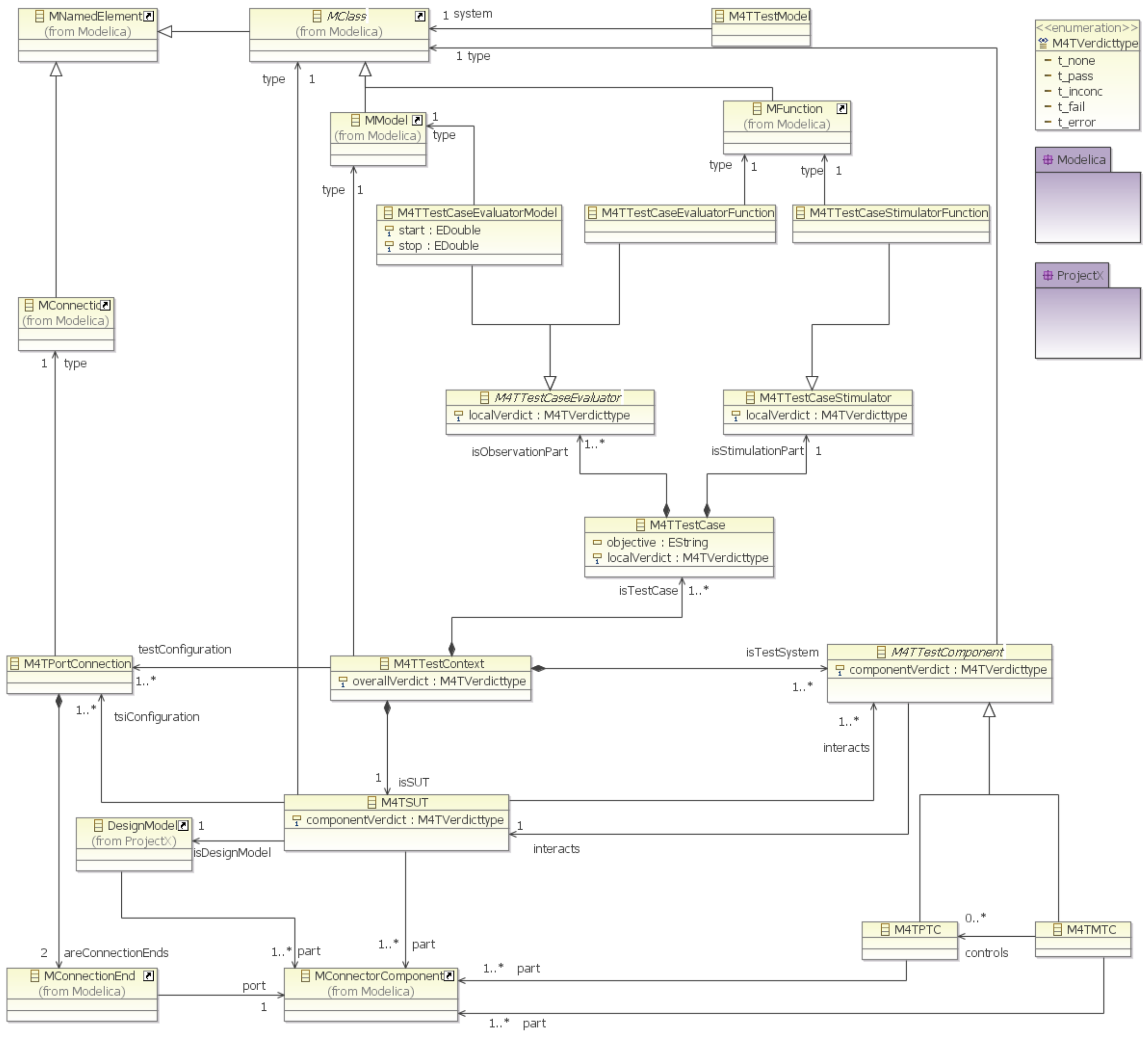


Figure 5-11 Modelica4Testing Meta-Model: Test Model Framework for Modelica

The M4T meta-model presented in Figure 5-11 shall represent the abstract structure of a test model defined in Modelica. A tool can adopt this meta-model to provide a simple but powerful framework for model-based testing using Modelica.

Illustrating the hierarchical scope of language elements, can help to classify elements. The scope represented in Figure 5-12 contains implicit language elements, as well as concrete elements, since it should support the user to understand the semantic of Modelica4Testing.

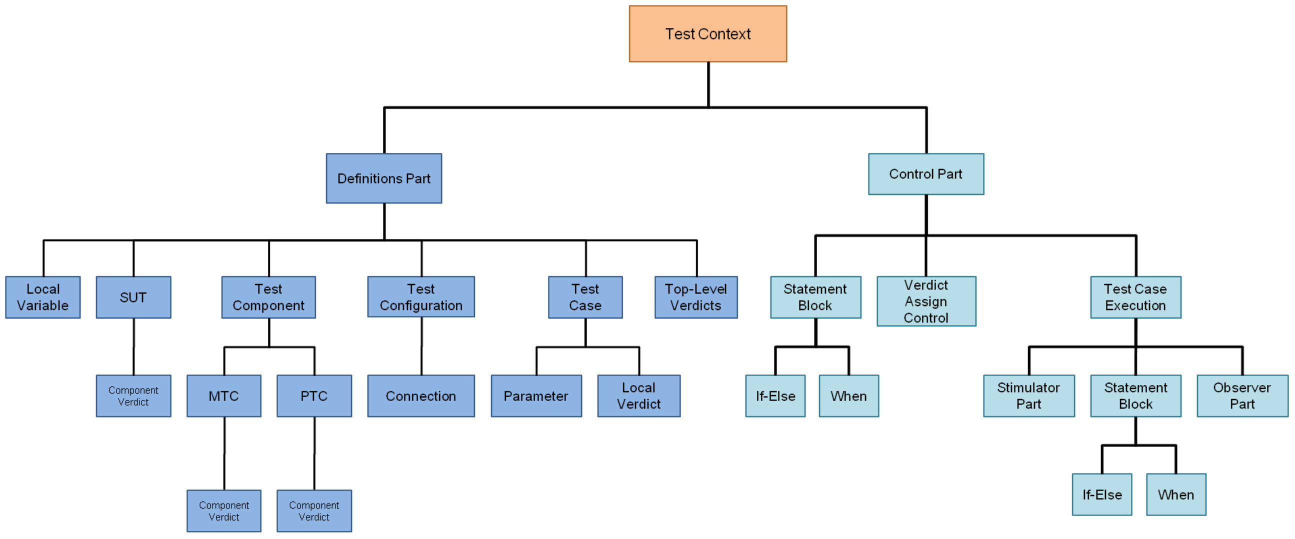


Figure 5-12 Modelica4Testing Hierarchical Scope

5.5. The Modelica4Testing Test Model Framework

To support the understanding and usage of Modelica4Testing by a user, a Modelica4Testing tool should provide the following Modelica files, as a test model framework. This will also support the reuse and gives at least a useful starting point for a user. A tool should separate the test model framework elements into different packages. For example the partial classes and final data types shall be implemented within a package called “Modelica4Testing_Library”. And the reusable skeletons should be implemented in a “verification and validation” package.

5.5.1. **Modelica4Testing_Library**

A language library contains predefined language elements. A library element shall not be changeable in its structure. Common elements are special data types, predefined interfaces and abstract classes.

Build-In Datatype Verdicttype

TestVerdict.mo (Modelica Type)

```
within Modelica4Testing_Library;
//test_none = 1, test_pass = 2, test_inconclusive = 3, test_fail  = 4,
test_error = 5

type TestVerdict = enumeration(test_none, test_pass, test_inconclusive,
test_fail, test_error);
```

Abstract Test Component

TestComponent.mo (Modelica)

```
within Modelica4Testing_Library;

partial model TestComponent
  Modelica4Testing_Library.Verdicttype componentVerdict (start =
Modelica4Testing_Library.Verdicttype.t_none);
end TestComponent;
```

Abstract Test Case Evaluator

TestCaseEvaluator.mo (Modelica)

```
within Modelica4Testing_Library;

partial class TestCaseEvaluator
  output Modelica4Testing_Library.Verdicttype localVerdict (start =
Modelica4Testing_Library.Verdicttype.t_none);
end TestCaseEvaluator;
```

Abstract Test Case Stimulator

TestCaseStimulator.mo (Modelica)

```
within Modelica4Testing_Library;

partial class TestCaseStimulator
  output Modelica4Testing_Library.Verdicttype localVerdict (start =
Modelica4Testing_Library.Verdicttype.t_none);
end TestCaseStimulator;
```

5.5.2. **Modelica4Testing Verification and Validation Model Skeletons**

Unlike library elements which are fixed, a set of predefined model skeletons can be used as a framework for a test model and as a development guideline. Indeed the structure of a skeleton is also fixed, but the user can complete the body with test and domain specific code. Below a set of predefined skeletons will be presented, which can be adopted by a Modelica4Testing tool.

System under Test (SUT)

SUT.mo (Modelica)

```
within Verification_and_Validation;
//A model containing the system under test
model SUT "System Under Test"
  output Modelica4Testing_Library.Verdicttype componentVerdict (start =
Modelica4Testing_Library.Verdicttype.t_none);

  //Test System Interface
  input TypeX in;
  output TypeY out;

  //Design Model to be tested as a system under test
  Type designModel;

equation
  //Test System Interface Configuration
  connect(in, designModel.in);
  connect(designModel.out, out);
algorithm
  //TBD use full component verdict impl.
  componentVerdict := Modelica4Testing_Library.Verdicttype.t_pass;

end SUT;
```

Main Test Component (MTC)

MTC.mo (Modelica)

```
within Verification_and_Validation;

model MTC "Main Test Component" extends Modelica4Testing_Library.TestComponent;
  //Ports
equation

algorithm
  //TBD use full component verdict impl.
  componentVerdict := Modelica4Testing_Library.Verdicttype.t_pass;

end MTC;
```

Parallel Test Component (PTC)

PTC.mo (Modelica)

```
within Verification_and_Validation;

model PTC "Parallel Test Compo." extends Modelica4Testing_Library.TestComponent;
  //Ports
equation

algorithm
  //TBD use full component verdict impl.
  componentVerdict := Modelica4Testing_Library.Verdicttype.t_pass;

end PTC;
```

Test Context

TestContext.mo (Modelica)

```
within Verification_and_Validation;

model TestContext

//Definition Part
  //Test Context Overall Verdict
  Modelica4Testing_Library.Verdicttype overallVerdict (start =
Modelica4Testing_Library.Verdicttype.t_none);
  //Test Case verdicts
  Modelica4Testing_Library.Verdicttype tc01_verdict (start =
Modelica4Testing_Library.Verdicttype.t_none);

  //System under Test
  Verification_and_Validation.SUT mySUT; //A model containing the SUT
  //Test Components
  Verification_and_Validation.MTC myMTC; //A model representing the MTC
  Verification_and_Validation.PTC myPTC; //A model representing a PTC

  //Local Variables
  Boolean b_var1 (start = true);
  Real r_var2  (start = 0.00);
  //Test Configuration
equation
      connect(mySUT.Out1, myPTC.In);
      connect(mySUT.Out2, myMTC.In);
      connect(myMTC.Out1, mySUT.In);

      myMTC.status.value = status;
      myMTC.flowLevel.value = flowLevel;
//Control Part
algorithm
  when time >= xx then
        (tc01_verdict, status, flowLevel) :=
TestCase_01_StimulatorFunction(mySUT.componentVerdict, myMTC.componentVerdict,
myPTC.componentVerdict);
        myMTC.status := status;
        myMTC.flowLevel := flowLevel;
  end when;
  when time >= 100 then
    if(tc01_verdict == 1) then  (tc01_verdict) :=
TestCase_01_EvaluatorFunction(mySUT.componentVerdict, myMTC.componentVerdict,
myPTC.componentVerdict, mySUT.qOut, mySUT.modeOut, mySUT.statusOut); end if;
  end when;

  //Same for test case 02

  //Assigning Overall Verdict
  overallVerdict := max(i for i in
{Integer(componentVerdict_SUT),Integer(componentVerdict_MTC),Integer(componentVe
rdict_PTC)});
end TestContext;
```

6. Application Example of System Level Testing

Finally, the approach to generate an executable design model from chapter 4 and the test model framework developed in chapter 5 can be combined to test the system implementation, of the Aircraft Water Tank System, against its functional specification, defined in chapter 3.

As mentioned above this approach focuses on the functional system design level, illustrated in Figure 2-3 V-Model XT on the left side. The correspondent test activity is the system level test. The goal of system level testing, in the term of early verification and validation, is to confirm a system model and a simulation prototype against the system specification requirements [9]. Since the system design level uses a more abstract view on the system, the validation is done using functional testing, also referred to as Black-Box testing. Black-Box testing is testing against the functional requirements of a SUT, without knowledge of the internal structure. It uses the SUT inputs as the point of control (PoC) and its outputs as the point of observation (PoO). The decision if a test case passes or fails mainly depends on the results given by the SUT outputs. Figure 6-1 represents the aircraft water tank system as Black-Box. The used inputs and outputs are represented as SysML ports.

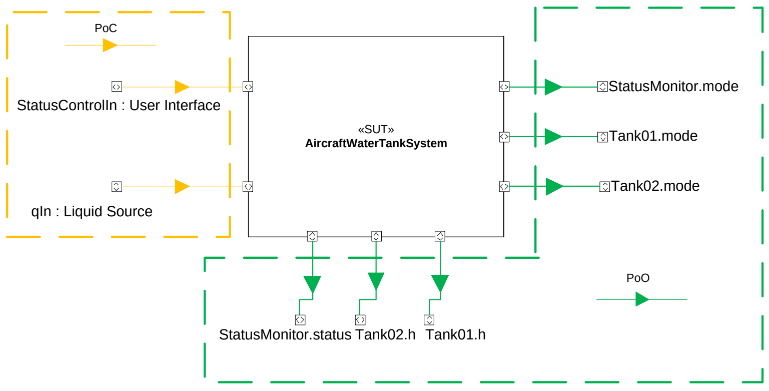


Figure 6-1 Black-Box view of the Aircraft Water Tank System as the SUT

Derived from the State Machine (SM) developed in chapter 0, a test case will be developed to test for errors in the system. From a testing point of view, a system may fail a test if it is exposed to an event, the guard conditions are not appropriated or the system either does not transition to another or to a wrong state [3]. This chapter will test the system based on the following typical SM tests:

- Test for state fault: There might be either extra or missing states.
- Test for action fault: The actions on a transition are incorrect or missing.
- Test for transition fault: The transition on a legal event is incorrect or missing.
- Test for guard condition fault: The guard condition on a transition is incorrect.

6.1. OpenModelica as Simulation and Test Execution Tool

A simulation tool is needed to execute the Modelica models. As mentioned in chapter 2.1.2 OpenModelica is an open-source and free available modelling and simulation environment for Modelica. Its Modelica compiler generates an executable file, including the model as C/C++ code and different solvers to simulate the model behaviour. The Figure 6-2 illustrates the used simulation environment, the SUT and the test model.

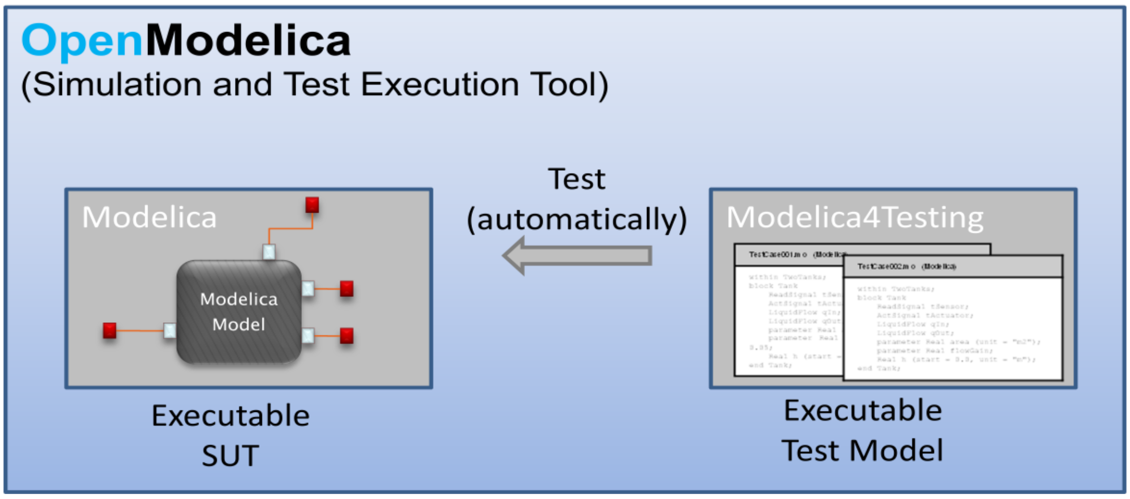
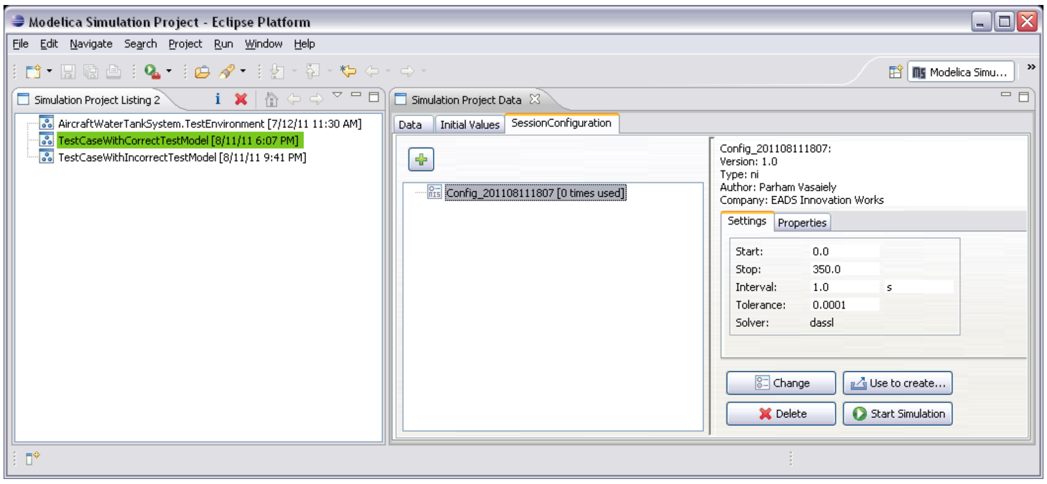


Figure 6-2 Executing the Test Model and the SUT using the Modelica Tool OpenModelica

In order to manage test sessions and to demonstrate the results more efficiently an Eclipse 3.6 [49] based simulation and test environment has been developed as an Eclipse Plug-In [52]. The environment is implemented in Java [48]. The developed simulation and test environment will use the OpenModelica Compiler [29] functionality, provided by OM to compile the model. A simulation and test configuration system provides the management of created simulation projects, test models, simulation sessions and their results.



AircraftWaterTankSystem.TestEnvironment [7/12/11 11:30 AM]
TestCaseWithCorrectTestModel [8/11/11 6:07 PM]
TestCaseWithIncorrectTestModel [8/11/11 9:41 PM]

Config_201108111807 [0 times used]

Config_201108111807:
Version: 1.0
Type: ni
Author: Parham Vasaiely
Company: EADS Innovation Works

Start:	0.0	
Stop:	350.0	
Interval:	1.0	s
Tolerance:	0.0001	
Solver:	dassl	

Figure 6-3 Eclipse based Simulation and Test Environment for OM: Management View

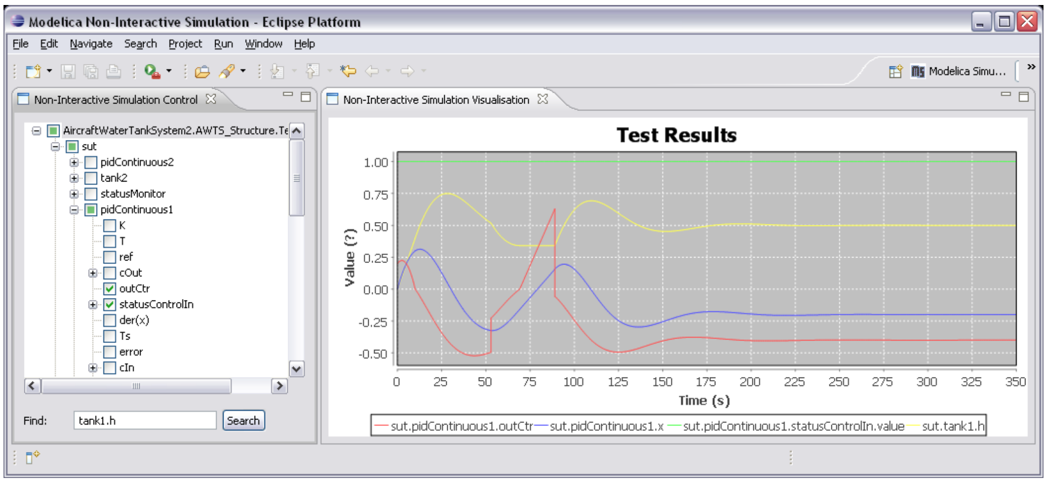

Modelica Non-Interactive Simulation - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Non-Interactive Simulation Control
AircraftWaterTankSystem2.AWTS_Structure.Te
sut
pidContinuous2
tank2
statusMonitor
pidContinuous1
K
T
ref
cOut
outCtr
statusControlIn
der(x)
Ts
error
cIn
Find: tank1.h Search
Non-Interactive Simulation Visualisation
Test Results
Value (?)
Time (s)
sut.pidContinuous1.outCtr
sut.pidContinuous1.x
sut.pidContinuous1.statusControlIn.value
sut.tank1.h

Figure 6-4 Eclipse based Simulation and Test Environment for OM: Result View

The tool and a full documentation are available for download on the OpenModelica.org website [36]. However by using the tools provided in the OM tools set, one can get the same result, but more inconvenient and without the mentioned configuration management features.

6.2. Test Case Specification

As mentioned above, the test case defined in this chapter will validate the expected system behaviour, described in section 0. The state machine depicted in Figure 6-5, expects that the system changes its internal state, when the control unit recognizes a different “mode” signal as a guard condition (see event definition in section 0.). The simple test case will not cover the correct implementation of the “online” and “offline” states, since the result is redundant.

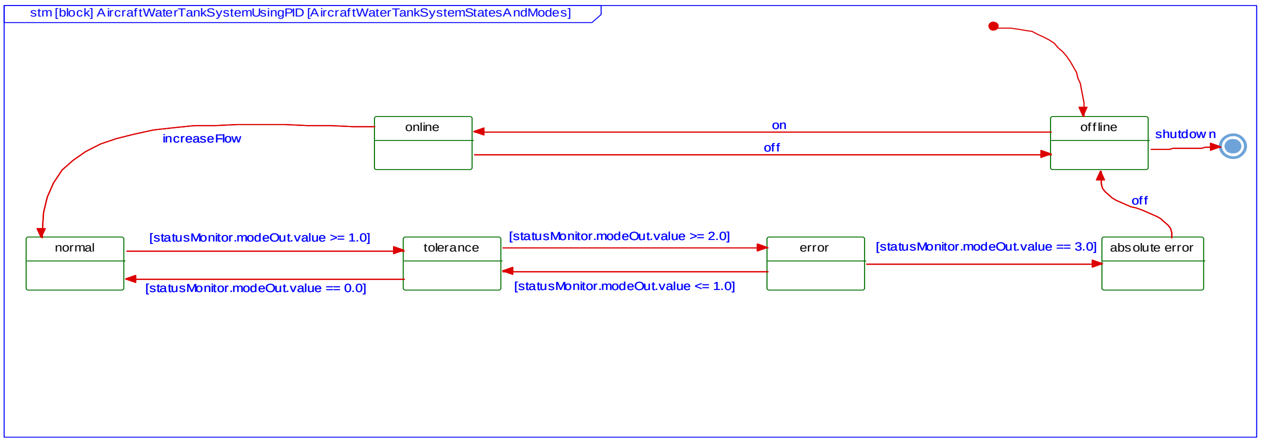

stm [block] AircraftWaterTankSystemUsingPID [AircraftWaterTankSystemStatesAndModes]
online
offline
on
off
shutdown
increaseFlow
off
normal
tolerance
error
absolute error
[statusMonitor.modeOut.value >= 1.0]
[statusMonitor.modeOut.value == 0.0]
[statusMonitor.modeOut.value >= 2.0]
[statusMonitor.modeOut.value <= 1.0]
[statusMonitor.modeOut.value == 3.0]

Figure 6-5 Copy of AWTS State Machine described in section 0.

6.2.1. **Test Case Specification as Text**

The following specification describes the needed test case as text:

- **Name:** State Machine Testing for correct Modes
- **Objective:** In this test case the user interface will increase the out flow level of the liquid source, in order to increase the level of liquid in the tanks. A liquid tank assigns its mode

using the actual tank level "h" and guard conditions. A control unit collects the modes of all tanks and assigns an overall mode for the system. In the given configuration the SUT should not went into the absolute error mode. Otherwise this test is failed. But a change from normal into tolerance and then error is allowed, and will be accepted as pass. The result after 250s can be observed using the "StatusMonitor.status", "StatusMonitor.mode" and "Tank1.h", "Tank2.h" outputs.

- **Precondition (At Simulation Time = 50s):**
    - System status is "Online" = "StatusMonitor.status" = true
    - "StatusMonitor.mode" = Normal
    - "Tank1.h" = 0.0 m, "Tank2.h" = 0.0 m
- **Input:** UserInterface.flowOut = 0.02 l/m^3;
- **Post-Condition (At Simulation Time = 350s):**
    - System status is "Online" = "StatusMonitor.status" = true
    - "StatusMonitor.mode" = Normal.
- **Output:** "Tank1.h" = 0.5 m, "Tank2.h" = 0.5 m

6.2.2. **Test Case implemented in Modelica4Testing**

The following is the test case implemented in Modelica using the developed test model framework Modelica4Testing from chapter 5.3. The test case is implemented as a single test case (5.3.4). As mentioned in section 5.3.2, a test case in Modelica4Testing is divided into two parts. The stimulator part is checks the preconditions and defines needed values. The evaluation part checks the post-conditions and evaluates the outputs to decide if a test is pass or fail.

TestCase_01_StimulatorFunction.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Verification_and_Validation;

function TestCase_01_StimulatorFunction extends
Modelica4Testing_Library.TestCaseStimulator;
  input Modelica4Testing_Library.Verdicttype componentVerdict_SUT;
  input Modelica4Testing_Library.Verdicttype componentVerdict_MTC;
  input Modelica4Testing_Library.Verdicttype componentVerdict_PTC;

  output Boolean status "false=offline, true=online";
  output Real flowLevel;

algorithm
  localVerdict := max(i for i in
{Integer(componentVerdict_SUT),Integer(componentVerdict_MTC),Integer(componentVe
rdict_PTC)});
    if(localVerdict == 1) then //only at pass the test case can proceed
      status := true;
      flowLevel := 0.02;
      return;
    end if;
    return;
end TestCase_01_StimulatorFunction;
```

Table 60 Test Case implementation in Modelica4Testing, Stimulator Part

TestCase_01_EvaluatorFunction.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Verification_and_Validation;

function TestCase_01_EvaluatorFunction extends
Modelica4Testing_Library.TestCaseEvaluator;
  input Modelica4Testing_Library.Verdicttype componentVerdict_SUT;
  input Modelica4Testing_Library.Verdicttype componentVerdict_MTC;
  input Modelica4Testing_Library.Verdicttype componentVerdict_PTC;

  input Real tank1h;
  input Real tank2h;
  input Model_Library.ModeSignal modeOut;

  output Modelica4Testing_Library.Verdicttype localVerdict;

algorithm
  localVerdict := max(i for i in
{Integer(componentVerdict_SUT),Integer(componentVerdict_MTC),Integer(componentVe
rdict_PTC)});
  if(localVerdict == 1) then // is pass?
    if (modeOut.value == 3.0 and ( tank1.h > 0.95 and tank2.h > 0.95 )) then
Modelica4Testing_Library.Verdicttype.t_pass
        else if ( (modeOut.value == 2.0 and ( (tank1.h > 0.8 and tank1.h <=
0.95) and (tank2.h >= 0.8 and tank2.h <= 0.95) )) and localVerdict ==
Modelica4Testing_Library.Verdicttype.t_pass ) then
Modelica4Testing_Library.Verdicttype.t_pass
        else if ( (modeOut.value == 1.0 and ( (tank1.h > 0.70 and tank1.h <=
0.80) and (tank2.h >= 0.70 and tank2.h < 0.80) )) and localVerdict ==
Modelica4Testing_Library.Verdicttype.t_pass ) then
Modelica4Testing_Library.Verdicttype.t_pass
        else if ( (modeOut.value == 0.0 and ( (tank1.h >= 0.0 and tank1.h <=
0.7) and (tank2.h >= 0.0 and tank2.h <= 0.7) )) and localVerdict ==
Modelica4Testing_Library.Verdicttype.t_pass ) then
Modelica4Testing_Library.Verdicttype.t_pass
    else localVerdict := Modelica4Testing_Library.Verdicttype.t_fail;
    end if;
  else localVerdict := Modelica4Testing_Library.Verdicttype.t_fail;
  end if;
  return;
end TestCase_01_EvaluatorFunction;
```

Table 61 Test Case implementation in Modelica4Testing, Evaluator Part

The complete test model can be found in the Appendix B.

6.3. System Simulation and Test Execution

In order to demonstrate the test case defined in section 6.2, once passed and once failed, the design model is implemented once correctly and with a fault respectively. The failure is implemented in the liquid tank when assigning its “mode” based on the tank level value “h”. As mentioned above, the correct system behaviour is to transit to another mode based on the level of liquid in a tank and guard conditions.

6.3.1. System under Test Preparation

The equation code presented in the Table 62 calculates the tolerance mode correctly, by assigning the tolerance mode when the level is smaller or equal 80% of the maximum tank height. Whereas the equation used in Table 63 assigns an incorrect value. Since the test case checks the dynamic behaviour in terms of compliance with the requirements expressed as a state machine, it should fail when using the fault equation.

LiquidTank.mo (Modelica) - **correct -**

```
within AircraftWaterTankSystem.AWTS_Structure;

block LiquidTank
  ...
Equation
  ...
      modeOut.value =   if h <= (maxTankHight * 0.7) then 0.0
                        else if h <= (maxTankHight * 0.8) then 1.0 ← Correct
                        else if h <= (maxTankHight * 0.95) then 2.0
                        else if h > (maxTankHight * 0.95) then 3.0
                        else -1.0;
end LiquidTank;
```

Table 62 Liquid Tank with correct implemented tank mode equation

LiquidTank.mo (Modelica) - **incorrect -**

```
within AircraftWaterTankSystem.AWTS_Structure;

block LiquidTank
  ...
Equation
  ...
      modeOut.value =   if h <= (maxTankHight * 0.7) then 0.0
                        else if h <= (maxTankHight * 0.8) then 0.0 ← Incorrect
                        else if h <= (maxTankHight * 0.95) then 2.0
                        else if h > (maxTankHight * 0.95) then 3.0
                        else -1.0;
end LiquidTank;
```

Table 63 Liquid Tank with incorrect implemented tank mode equation

6.3.2. **Test Execution and Evaluation Process**

Using the developed eclipse application one can load the Modelica code and start the simulation. Using the verdict type definition from chapter 5.3.3 a test is pass if the verdict value is “2”, and fail if the value is “4”. The figures below are representing the test results as graphs. The following lines are represented:

Attribute	Description	Color
tank1.h	Level of water in the first tank.	Yellow
tank2.h	Level of water in the second tank.	Green
statusMonitor.mode	System mode to represent the different states. Normal=0, Tolerance=1, Error=2, Abs.Error=3.	Red
overallVerdict	The overall verdict of the system. Where among others, pass=2 and fail=4	Blue

Table 64 Attributes as graphs in the simulation result plot

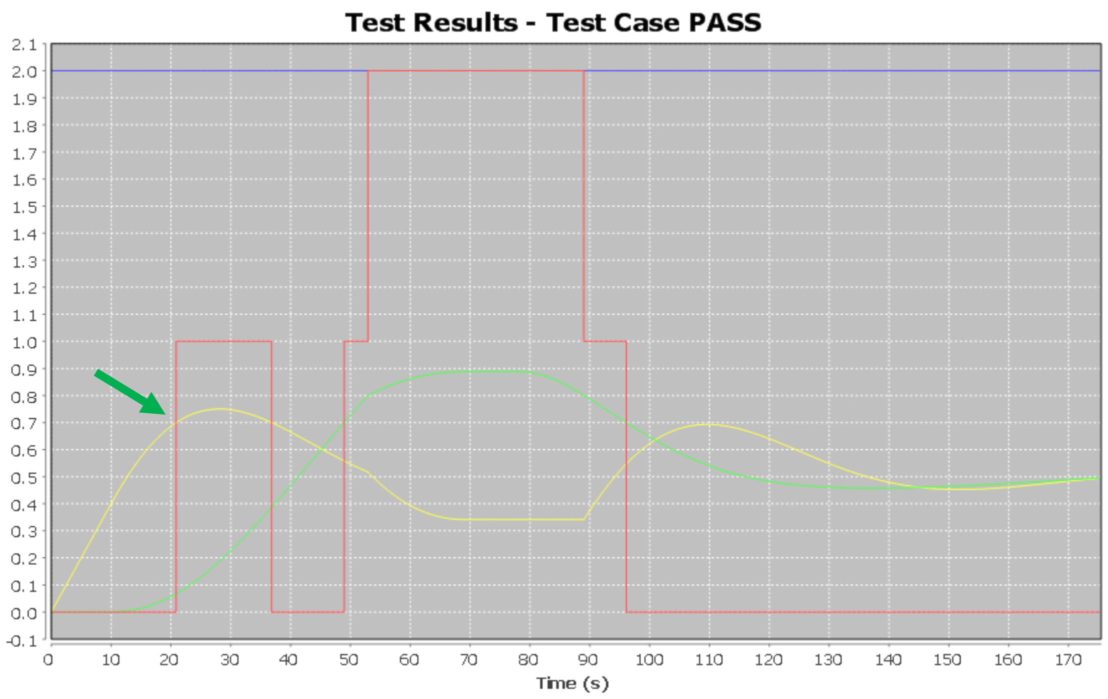


Figure 6-6 Simulation Results for a correct Model, Test Case is Pass

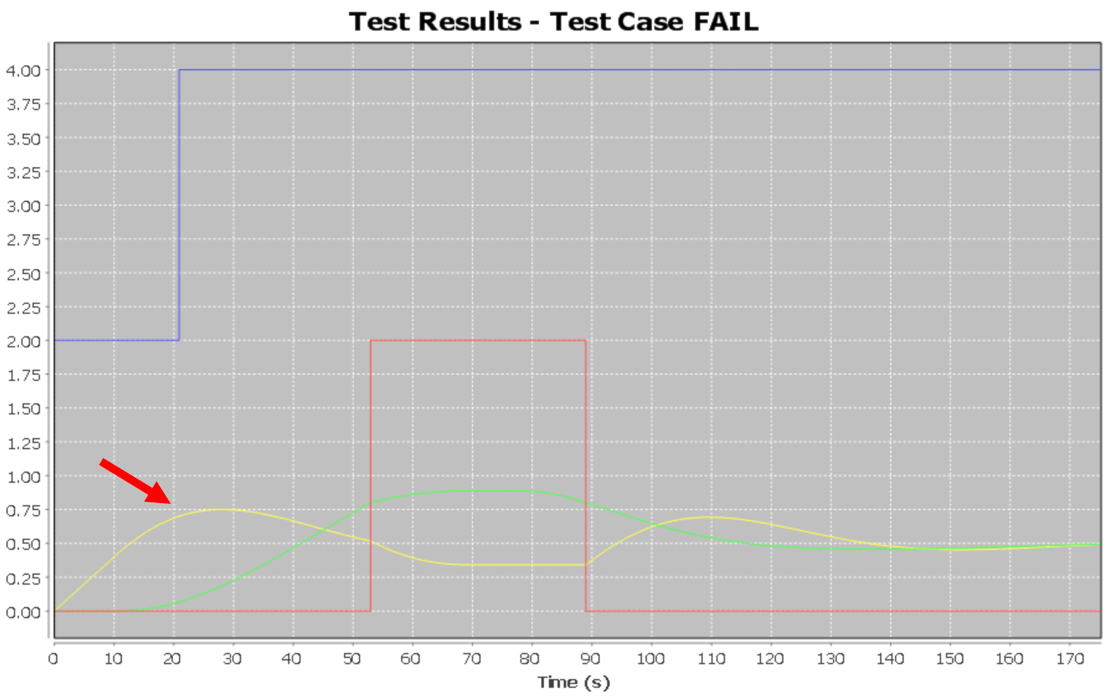


Figure 6-7 Simulation Results for a incorrect Model, the Test Case is Fail

The results of the first test session, where the correct system was in use, are presented in the Figure 6-6. The test is pass, since the system transition at ~21s, ~50s and ~90s is changing to the tolerance mode.

The second test session, using the incorrect system model, is presented in Figure 6-7. The system transition to the tolerance state is not happening at ~20s. Once a test is failed it can't be pass anymore, therefore the missing transition at ~50s and ~90s are not respected.

The Figure 6-6 and the Figure 6-7 are presenting the reason why the second test session failed in more detail.

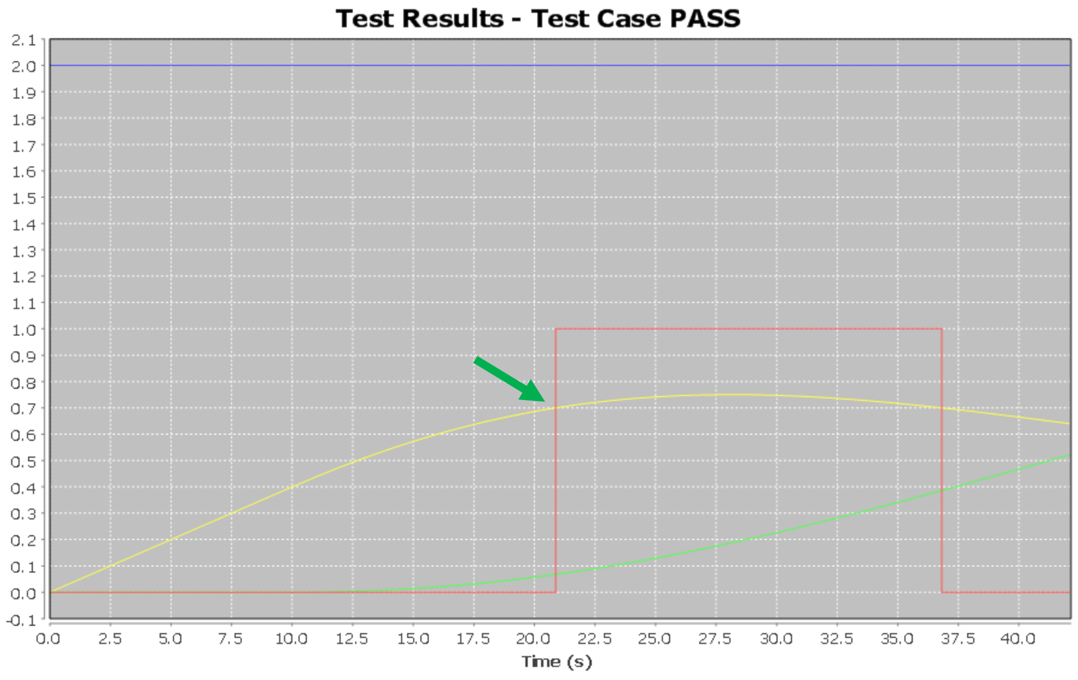


Figure 6-8 Simulation Results for the first test session in detail

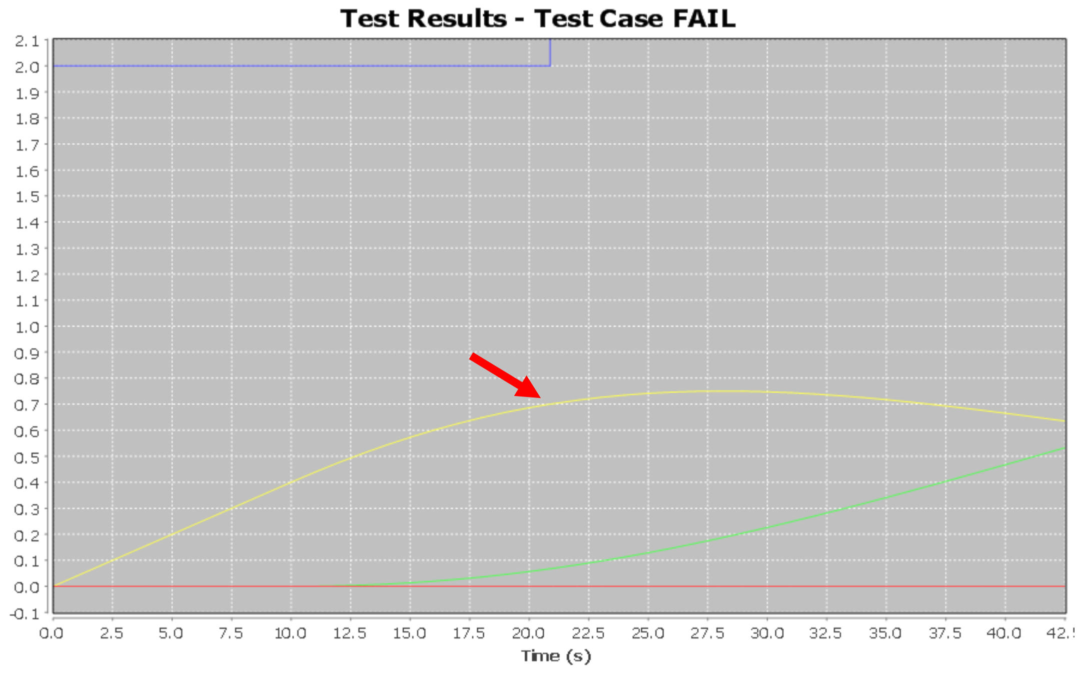


Figure 6-9 Simulation Results for the second test session in detail

7. Conclusions and Future Work

7.1. Conclusions

The approach presented in this work proved the possibility to translate an entire SysML system design model into the programming language Modelica, in order to make it executable for early prototyping, verification and validation. An aircraft water tank system is used as a concrete system example, to illustrate this approach. But the used subset of SysML and Modelica language elements has to be extended to fulfil the needs of real engineering project. The results of the SysML-Modelica Integration Group can support to fill this gap, since it is a wider approach.

The test specification and implementation language Modelica4Testing has been developed in Modelica by using existing concepts from the standardized test specification language TTCN-3. A testing tool can implement this specification language as a test model framework, to support users by developing tests in Modelica. Figure 7-1 illustrates the testing approach given by the test model framework.

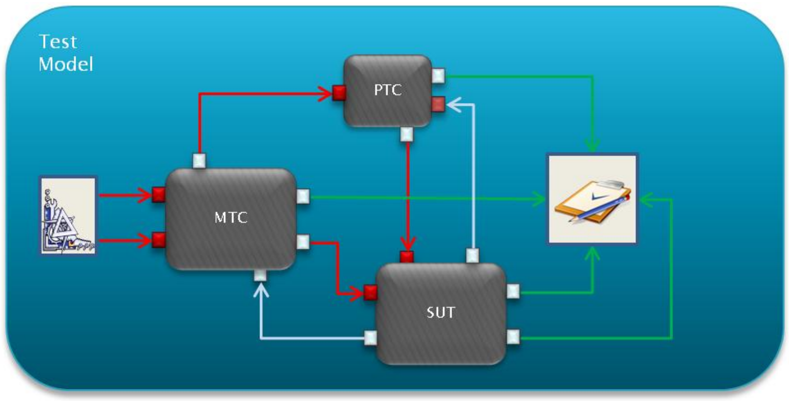


Figure 7-1 Illustration of the test model used in the M4T approach

In addition the work shows that existing IT standards, defined by the IEEE and the OMG, used as a basis for new technologies in research, will not only support their global understanding and dissemination, but also reduce time and cost of developing new technologies and of finding acceptance by developers when used in projects.

7.2. Future Work

As mentioned in section 4.3, automated code generation is the key feature to make a model-based approach more applicable and useful in operational field. A major improvement to the presented approach in this work will be the implementation of a graphical and model-based test specification language, such as the OMG UML2 Testing Profile (UTP), to specify tests and to generate Modelica4Testing code automatically. UTP provides extensions to UML to support the design, visualization, specification, analysis, construction, and documentation of the artefacts involved in testing. It has been standardized by the OMG [25].

In our case, SysML is the language for specifying models and UTP will be the formalism to describe the derived tests. Using UTP, test relevant behaviour described by SysML behaviour diagrams, like Sequence or Activity Diagrams can be used to derive these tests more automatically. In addition tests can be communicated more efficient between the system design team and a test engineering team.

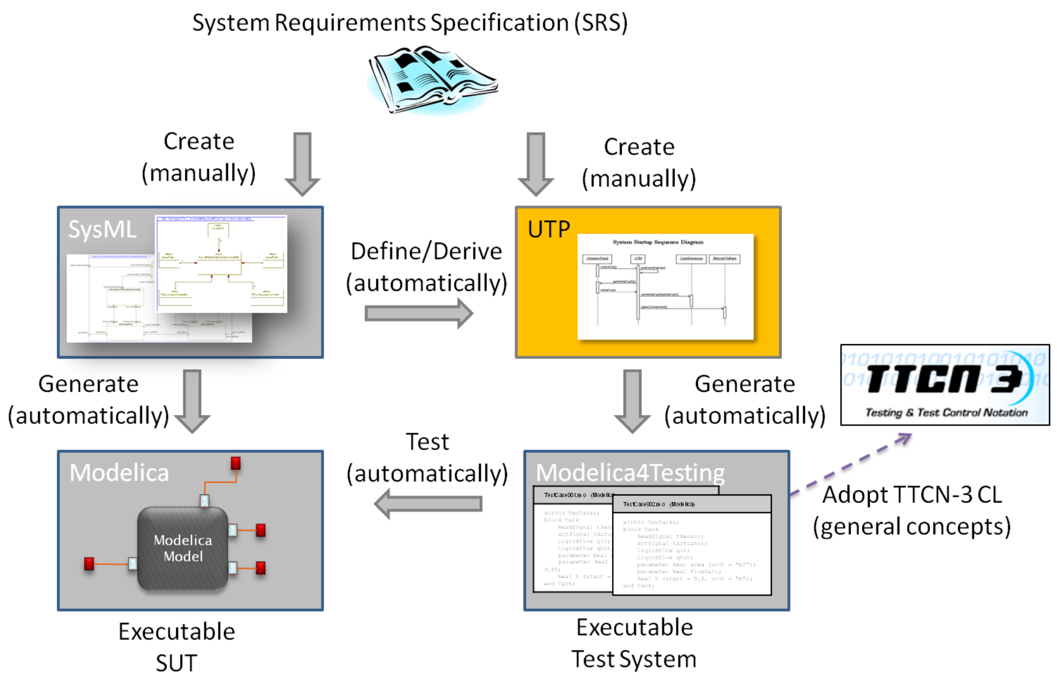


Figure 7-2 Including the UML2 Testing Profile as Future Work

The Appendix C will present an approach of mapping UTP to Modelica4Testing, in order to support future research projects from this field.

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] Avner Engel, 2010, Verification, Validation, and Testing of Engineered Systems, Wiley Press

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

[5] Baker, P., Dai, Z.R., Grabowski, J., Schieferdecker, I., Williams, C., 2007, Model-Driven Testing: Using the UML Testing Profile, Springer

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

[7] 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

[8] EADS Innovation Works, HAW Hamburg, Parham Vasaiely, Bachelor Thesis: "Interactive Simulation of SysML Models using Modelica.pdf", August 2009. http://opus.haw-hamburg.de/volltexte/2009/842/

[9] THE V-MODELL® XT Version 1.3, 2009, http://v-modell.iabg.de/dmdocuments/V-Modell-XT-Gesamt-Englisch-V1.3.pdf

[10] PELAB, Peter Fritzson, "OpenModelica Users Guide, Version 2011-06-13 for OM 1.7.0", http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/releases/current/doc/OpenModelicaUsersGuide.pdf, June 2011.

[11] PELAB, Peter Fritzson, “OpenModelica System Documentation, Ver. 2011-04-19 for OM 1.7.0”, http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/releases/current/doc/OpenModelicaSystem.pdf, April 2011.

[12] IBM Systems Engineering Tutorial for Rational Rhapsody, Rhapsody 7.4, http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.rhapsody.doc/pdf76/tutorial_Systems_Eng.pdf, 2009

[13] The Institute of Electrical and Electronic Engineers, IEEE Std. 610, IEEE Standard Computer Dictionary, http://ieeexplore.ieee.org/servlet/opac?punumber=2267, 1991

[14] The Institute of Electrical and Electronic Engineers, IEEE Std. 829-2008, IEEE Standard for Software and System Test Documentation, http://ieeexplore.ieee.org/servlet/opac?punumber=4578271, 2008

[15] The Institute of Electrical and Electronic Engineers, IEEE Std. 1012-2004, IEEE Standard for Software Verification and Validation, http://ieeexplore.ieee.org/servlet/opac?punumber=9958, 2005

[16] The Testing and Test Control Notation v3, TTCN-3 Core Language, ETSI ES 201 873-1 V4.3., http://www.etsi.org/deliver/etsi_es/201800_201899/20187301/04.03.01_60/es_20187301v040301p.pdf, 2011

[17] Object Management Group MOF QVT, Meta Object Facility (MOF) 2.0 Query / View / Transformation Specification, Version 1.0, http://www.omg.org/spec/QVT/1.0/PDF/, 2008

[18] Object Management Group UML, “OMG Unified Modeling Language (OMG UML), Superstructure, V2.3, http://www.omg.org/spec/UML/2.3/Superstructure/PDF/, 2010

[19] Object Management Group SysML, “OMG Systems Modeling Language 1.2 (OMG SysML™) Specification, Version 1.2”, http://www.omg.org/spec/SysML/1.2/PDF/, 2010.

[20] Sparx Systems Pty. Ltd and ICONIX, Doug Rosenberg, Sam Mancarella, Embedded Systems Development using SysML, http://www.sparxsystems.com/downloads/ebooks/Embedded_Systems_Development_using_SysML.pdf, 2010

[21] ITU-T Study Group VII - Data networks and open system communications, Open Systems Interconnection – Conformance testing, ITU-T Recommendation X.290, http://www.itu.int/ITU-T/recommendations/rec.aspx?rec=X.290, 1995

[22] ITU-T Study Group VII - Data networks and open system communications, Open Systems Interconnection – Conformance testing, ITU-T Recommendation X.292, http://www.itu.int/ITU-T/recommendations/rec.aspx?rec=X.292, 2002

[23] Modelica Association, "Modelica Language Specification Version 3.2", www.modelica.org/documents/ModelicaSpec32.pdf, March, 2010.

[24] Department of Defense, DoD Modeling and Simulation (M&S) Management, http://www.cotf.navy.mil/files/ms/dodd%20m&s%20mgt%205000.59.pdf, 1994

[25] Object Management Group UTP, UML 2.0 Testing Profile Specification version 2.0, http://www.fokus.fraunhofer.de/u2tp/documents/UMLTestingProfile_FinalAdoptedSpecification.pdf, 2004

[26] The International Council on Systems Engineering (INCOSE), Last Accessed: 2011 http://www.incose.org/

[27] Object Management Group (OMG), Last Accessed: 2011 http://www.omg.org/

[28] Object Management Group (OMG) Unified Modeling Language (UML), Last Accessed: 2011 http://www.uml.org/

[29] Object Management Group (OMG) Systems Modelling Language, Last Accessed: 2011 http://www.omgsysml.org/

[30] The Institute of Electrical and Electronic Engineers IEEE, Last Accessed: 2011 www.ieee.org/

[31] The Testing and Test Control Notation Version 3 (TTCN-3), Last Accessed: 2011 www.ttcn-3.org/

[32] Modelica and the Modelica Association, Last Accessed: 2011 http://www.modelica.org/

[33] Modelica and the Modelica Association, Modelica Libraries, Last Accessed: 2011 http://www.modelica.org/libraries

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

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

[36] OpenModelica, Modelica Modeling and Simulation Tool, Last Accessed: 2009 http://www.openmodelica.org/

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

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

[39] IBM Rational Rhapsody, Systems-Engineering Tool Rhapsody 7.6, Last Accessed: 2011 http://www-01.ibm.com/software/awdtools/rhapsody/

[40] AUTomotive Open System Architecture (AUTOSAR), Last Accessed: 2011 www.autosar.org/

[41] Department of Defense Architecture Framework, DoDAF, Last Accessed: 2011 http://cio-nii.defense.gov/sites/dodaf20/

[42] SmartCockpit, Airbus A330 Water & Waste System, Last Accessed: 2011 http://www.smartcockpit.com/data/pdfs/plane/airbus/A330/

[43] Control-Systems-Principles Organisation, Last Accessed: 2011 http://www.control-systems-principles.co.uk/

[44] SysML and Modelica Integration, OMG Working Group, Last Accessed: 2011 http://www.omgwiki.org/OMGSysML/doku.php?id=sysml-modelica:sysml_and_modelica_integration

[45] IBM Rational Rhapsody 7.6 Tutorial and Users Guide, Last Accessed: 2011 http://publib.boulder.ibm.com/infocenter/rhaphlp/v7r6/index.jsp?topic=/com.ibm.rhapsody.designing.doc/topics/rhp_c_dm_system_eng_reqmts_rhap.html

[46] Papyrus UML2 Modeling Tool, Last Accessed: 2011, http://www.papyrusuml.org/

[47] Acceleo, Model 2 Code Transformation, Code Generator, Last Accessed: 2011, http://www.acceleo.org/pages/home/en

[48] Oracle, Java, Last Accessed: 2011, http://www.oracle.com/us/java/index.html

[49] Eclipse 3.6, Multi-Language Software Development Environment, Last Accessed: 2011 http://www.eclipse.org/

[50] JUnit, unit testing framework for Java, Last Accessed: 2011 https://www.junit.org/

[51] Eclipse Modeling Framework Project (EMF), Last Accessed: 2011, http://www.eclipse.org/modeling/emf/

[52] The Eclipse Plug-in Development Environment (PDE), Version 3.6, Last Accessed: 2011 http://www.eclipse.org/pde/

V. Appendix

Appendix A. Modelica design model code of Aircraft Water Tank System

package.mo (AircraftWaterTankSystem)

```
package AircraftWaterTankSystem
end AircraftWaterTankSystem;
```

package.mo (AWTS_Structure)

```
package AWTS_Structure
end AWTS_Structure;
```

package.mo (Model_Library)

```
package Model_Library
end Model_Library;
```

package.mo (External_Systems)

```
package External_Systems
end External_Systems;
```

ReadSignal.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

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

ActuatorSignal.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

connector ActuatorSignal // Signal to an actuator for setting valve position
  Real act;
end ActuatorSignal;
```

LiquidFlow.mo (Modelica)

```
within AircraftWaterTankSystem.Model_Library;

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

ModeSignal.mo (Modelica)

```
within AircraftWaterTankSystem.Model_Library;

connector ModeSignal // Signal to represent the system or component mode
  Real value; // 0=normal, 1=tolerance, 2=error, 3=abs. error
end ModeSignal;
```

<< flowProperty >> StatusSignal → StatusSignal.mo (Modelica)

```
within AircraftWaterTankSystem.Model_Library;

connector StatusSignal // Signal to represent the system or component status
  Real value; // 0=offline, 1=online
end StatusSignal;
```

<< flowProperty >> FlowLevelSignal → FlowLevelSignal.mo (Modelica)

```
within AircraftWaterTankSystem.Model_Library;

connector FlowLevelSignal // Liquid flow level as measurement
  Real value (unit = "m3/s");
end FlowLevelSignal;
```

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

```
within AircraftWaterTankSystem.AWTS_Structure;

partial block BaseController
  input AircraftWaterTankSystem.Model_Library.StatusSignal statusControlIn
"status control signal to turn the controller on or off";
  input AircraftWaterTankSystem.AWTS_Structure.ReadSignal cIn "Input sensor
level, connector";
  output AircraftWaterTankSystem.AWTS_Structure.ActuatorSignal cOut "Control to
actuator, connector";

  Real Ts(unit = "s") = 0.1;
  Real K = 2 "Gain";
  Real T(unit = "s") = 10 "Time constant";
  Real ref "Reference level";
  Real error "Deviation from reference level";
  Real outCtr "Output control signal";
equation
      error = ref - cIn.val;
      cOut.act = if(statusControlIn.value) then outCtr else 0.0;
end BaseController;
```

<<block>> PIDcontinuousController → PIDcontinuousController.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

block PIDcontinuousController extends BaseController(K = 2, T = 10);
      Real x; // State variable of continuous PID controller
      Real y; // State variable of continuous PID controller
equation
      der(x) = error/T;
      y = T*der(error);
      outCtr = K*(error + x + y);
end PIDcontinuousController;
```

<<block>> LiquidTank → LiquidTank.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

block LiquidTank
  input AircraftWaterTankSystem.AWTS_Structure.ActuatorSignal tActuator; //
Connector, actuator controlling input flow
  input AircraftWaterTankSystem.Model_Library.LiquidFlow qIn; // Connector, flow
(m3/s) through input valve
```

```
  output AircraftWaterTankSystem.AWTS_Structure.ReadSignal tSensor; //
Connector, sensor reading tank level (m)
  output AircraftWaterTankSystem.Model_Library.ModeSignal modeOut; // 0.0 when
system is in normal mode otherwise 1.0
  output AircraftWaterTankSystem.Model_Library.LiquidFlow qOut; // Connector,
flow (m3/s) through output valve
  //output Real h(start = 0.0, unit = "m"); //Tank level
  Real h(start = 0.0, unit = "m"); //Tank level
  Real area(unit = "m2") = 1; //Will be given as a parameter
  Real maxTankHight (unit = "m") = 1.0; //Will be given as a parameter
  Real flowGain (start = 1.99, unit = "m2/s") = 0.05;
  Real minV = 0; // Minimum for output valve flow
  Real maxV = 10; // Limit for output valve flow
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;
      modeOut.value =   if h <= (maxTankHight * 0.7) then 0.0
                        else if h <= (maxTankHight * 0.8) then 1.0
                        else if h <= (maxTankHight * 0.95) then 2.0
                        else if h > (maxTankHight * 0.95) then 3.0
                        else -1.0;
end LiquidTank;
```

<<block>> ControlUnit → ControlUnit.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

block ControlUnit
  input Model_Library.ModeSignal modeIn1;
  input Model_Library.ModeSignal modeIn2;
  input Model_Library.StatusSignal statusControlIn; // status control signal
from a user interface to turn the system on or off
  output Model_Library.ModeSignal modeOut;
  output Model_Library.StatusSignal statusControlOut1; // status control signal
to turn a controller on or off
  output Model_Library.StatusSignal statusControlOut2; // status control signal
to turn a controller on or off
  output Model_Library.StatusSignal statusOut;
  equation
    modeOut.value =
    if (modeIn1.value >= modeIn2.value) then modeIn1.value
      else modeIn2.value;
    statusOut.value = if (modeOut.value >= 3.0) then false else
statusControlIn.value;
    statusControlOut1.value = statusOut.value;
    statusControlOut2.value = statusOut.value;
end ControlUnit;
```

AircraftWaterTankSystemUsingPID.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Structure;

block AircraftWaterTankSystemUsingPID
  AircraftWaterTankSystem.AWTS_Structure.ControlUnit statusMonitor;
  AircraftWaterTankSystem.AWTS_Structure.LiquidTank tank1(area = 0.5,
maxTankHight = 1);
  AircraftWaterTankSystem.AWTS_Structure.LiquidTank tank2(area = 1,
maxTankHight = 1);
```

```
  AircraftWaterTankSystem.AWTS_Structure.PIDcontinuousController
pidContinuous1(ref = 0.5); //1 = 100% of maxTankHight
  AircraftWaterTankSystem.AWTS_Structure.PIDcontinuousController
pidContinuous2(ref = 0.5); //1 =  100% of maxTankHight

  input AircraftWaterTankSystem.Model_Library.LiquidFlow qIn; // Connector, flow
(m3/s) through input valve
  input AircraftWaterTankSystem.Model_Library.StatusSignal statusControlIn; //
status control signal from a user interface to turn the system on or off
  output AircraftWaterTankSystem.Model_Library.LiquidFlow qOut;
  output AircraftWaterTankSystem.Model_Library.ModeSignal modeOut;
  output AircraftWaterTankSystem.Model_Library.StatusSignal statusOut;
  //output Real h(start = 0.0);
equation
      connect(statusMonitor.statusControlOut1, pidContinuous1.statusControlIn);
      connect(statusMonitor.statusControlOut2, pidContinuous2.statusControlIn);
      connect(tank1.modeOut, statusMonitor.modeIn1);
      connect(tank2.modeOut, statusMonitor.modeIn2);
      connect(tank1.qOut, tank2.qIn);
      connect(tank2.qOut, qOut);
      connect(tank1.tSensor, pidContinuous1.cIn);
      connect(pidContinuous1.cOut, tank1.tActuator);
      connect(tank2.tSensor, pidContinuous2.cIn);
      connect(pidContinuous2.cOut, tank2.tActuator);
      connect(statusMonitor.modeOut, modeOut);
      connect(statusControlIn, statusMonitor.statusControlIn);
      connect(statusMonitor.statusOut, statusOut);
      connect(qIn, tank1.qIn);
end AircraftWaterTankSystemUsingPID;
```

Appendix B. Test model code developed in Modelica4Testing

SUT.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Verification_and_Validation;

model SUT "System Under Test"
  output Modelica4Testing_Library.Verdicttype componentVerdict (start =
Modelica4Testing_Library.Verdicttype.t_none);

  //Test System Interface
  input Model_Library.LiquidFlow qIn;
  input Model_Library.StatusSignal statusControlIn;
  output Model_Library.LiquidFlow qOut;
  output Model_Library.ModeSignal modeOut;
  output Model_Library.StatusSignal statusOut;

  //Design Model to be tested as a system under test
  AWTS_Structure.AircraftWaterTankSystemUsingPID designModel;

equation
  //Test System Interface Configuration
  connect(qIn, designModel.qIn);
  connect(statusControlIn, designModel.statusControlIn);
  connect(designModel.qOut, qOut);
  connect(designModel.modeOut, modeOut);
  connect(designModel.statusOut, statusOut);

algorithm
  //TBD use full component verdict impl.
  componentVerdict := Modelica4Testing_Library.Verdicttype.t_pass;

end SUT;
```

MTC.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Verification_and_Validation;

//The MTC will be the user interface
model MTC "Main Test Component" extends Modelica4Testing_Library.TestComponent;
  input Model_Library.StatusSignal statusIn "System status signal from a control
unit";
  input Model_Library.ModeSignal modeIn "System status signal from a control
unit";

  input Model_Library.StatusSignal status"false=offline, true=online";
  input Model_Library.FlowLevelSignal flowLevel;

  output Model_Library.StatusSignal statusControlOut "status control signal to
turn the system on or off";
  output Model_Library.FlowLevelSignal fLevelOut "Control to actuator, connector
in m3/s";

  constant Real maxFlowLevel = 0.12 "m3/s";
      constant Real minFlowLevel = 0.00 "m3/s";
equation
  statusControlOut.value = status.value;
  fLevelOut.value = if(statusIn.value) then flowLevel.value else 0.0;
algorithm
  componentVerdict := if(flowLevel.value >= minFlowLevel and flowLevel.value <=
maxFlowLevel) then Modelica4Testing_Library.Verdicttype.t_pass else
Modelica4Testing_Library.Verdicttype.t_fail;
end MTC;
```

PTC.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Verification_and_Validation;

//The PTC will be the liquid source
model PTC_01 "Parallel Test Component" extends
Modelica4Testing_Library.TestComponent;
  input AircraftWaterTankSystem.Model_Library.ModeSignal modeIn "if this signal
is >1.0 than the system is in error mode and the qOut must been closed during a
process is running";
      input AircraftWaterTankSystem.Model_Library.FlowLevelSignal fLevelIn
"m3/s";
      output AircraftWaterTankSystem.Model_Library.LiquidFlow qOut;
      constant Real maxFlowLevel = 0.12 "m3/s";
      constant Real minFlowLevel = 0.00 "m3/s";
      Real flowLevel(start = 0.0, unit = "m3/s");
equation
  flowLevel = if(fLevelIn.value < minFlowLevel) then minFlowLevel
  else if (fLevelIn.value > maxFlowLevel) then maxFlowLevel
  else fLevelIn.value;
      qOut.lflow = if (modeIn.value <= 1.0) then flowLevel else 0.0;
algorithm
  componentVerdict := if(flowLevel >= minFlowLevel and flowLevel <=
maxFlowLevel) then Modelica4Testing_Library.Verdicttype.t_pass else
Modelica4Testing_Library.Verdicttype.t_fail;
end PTC_01;
```

TestContext.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Verification_and_Validation;

model TestContext

//Definition Part
  //Test Context Overall Verdict
  Modelica4Testing_Library.Verdicttype overallVerdict (start =
Modelica4Testing_Library.Verdicttype.t_none); //an build-in Arbiter will collect
all
  //Test Case verdicts
  Modelica4Testing_Library.Verdicttype tc01_verdict (start =
Modelica4Testing_Library.Verdicttype.t_none);
  Modelica4Testing_Library.Verdicttype tc02_verdict (start =
Modelica4Testing_Library.Verdicttype.t_pass);

  //System under Test
  AWTS_Verification_and_Validation.SUT mySUT; //A model containing the system
under test
  //Test Components
  AWTS_Verification_and_Validation.MTC myMTC; //A model representing the main
test component
  AWTS_Verification_and_Validation.PTC myPTC; //A model representing a parallel
test component

  //Local Variables
  Boolean status (start = true) "false=offline, true=online";
  Real flowLevel  (start = 0.02);

  //Test Configuration

equation
      connect(mySUT.modeOut, myPTC.modeIn);
      connect(mySUT.modeOut, myMTC.modeIn);
      connect(myMTC.statusControlOut, mySUT.statusControlIn);
      connect(mySUT.statusOut, myMTC.statusIn);
      connect(myPTC.qOut, mySUT.qIn);
      connect(myMTC.fLevelOut, myPTC.fLevelIn);

      myMTC.status.value = status;
      myMTC.flowLevel.value = flowLevel;
  status = true;
  flowLevel = 0.05;
//Control Part
algorithm
  tc01_verdict := Modelica4Testing_Library.Verdicttype.t_none;
  tc02_verdict := Modelica4Testing_Library.Verdicttype.t_inconc;
  overallVerdict := Modelica4Testing_Library.Verdicttype.t_none;
end TestContext;
```

TestCase_01_StimulatorFunction.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Verification_and_Validation;

function TestCase_01_StimulatorFunction extends
Modelica4Testing_Library.TestCaseStimulator;
  input Modelica4Testing_Library.Verdicttype componentVerdict_SUT;
  input Modelica4Testing_Library.Verdicttype componentVerdict_MTC;
  input Modelica4Testing_Library.Verdicttype componentVerdict_PTC;

  output Boolean status "false=offline, true=online";
  output Real flowLevel;

algorithm
  localVerdict := max(i for i in
{Integer(componentVerdict_SUT),Integer(componentVerdict_MTC),Integer(componentVe
rdict_PTC)});
    if(localVerdict == 1) then //only at pass the test case can proceed
      status := true;
      flowLevel := 0.02;
      return;
    end if;
    return;
end TestCase_01_StimulatorFunction;
```

TestCase_01_StimulatorFunction.mo (Modelica)

```
within AircraftWaterTankSystem.AWTS_Verification_and_Validation;

function TestCase_01_StimulatorFunction extends
Modelica4Testing_Library.TestCaseStimulator;
  input Modelica4Testing_Library.Verdicttype componentVerdict_SUT;
  input Modelica4Testing_Library.Verdicttype componentVerdict_MTC;
  input Modelica4Testing_Library.Verdicttype componentVerdict_PTC;

  output Boolean status "false=offline, true=online";
  output Real flowLevel;

algorithm
  localVerdict := max(i for i in
{Integer(componentVerdict_SUT),Integer(componentVerdict_MTC),Integer(componentVe
rdict_PTC)});
    if(localVerdict == 1) then //only at pass the test case can proceed
      status := true;
      flowLevel := 0.02;
      return;
    end if;
    return;
end TestCase_01_StimulatorFunction;
```

Appendix C. Approach of mapping UTP to Modelica4Testing

The OMG UML2 Testing Profile provides extensions to UML to support the design, visualization, specification, analysis, construction, and documentation of the artefacts involved in testing [25]. It is independent of implementation languages and technologies, and can be applied in a variety of domains of development. UTP addresses concepts like test suites, test cases, test configuration, test component and test results, thus enabling the specification of different types of testing like, functional, interoperability, scalability and even load testing.

Since TTCN-3 was one basis for the development of the UML 2 Testing Profile and the UTP developers created a well defined mapping between these languages [5], it is useful for future research projects to create also a mapping between Modelica4Testing and UTP. The fact that, the M4T concepts are derived from TTCN-3 and some of its concepts bases on UTP makes this step even more reasonable. However it should be noted that, the UML2 Testing Profile is targeted at UML based software and protocol tests, but the basic test concepts of software and systems are similar in the meaning of verification and validation [14].

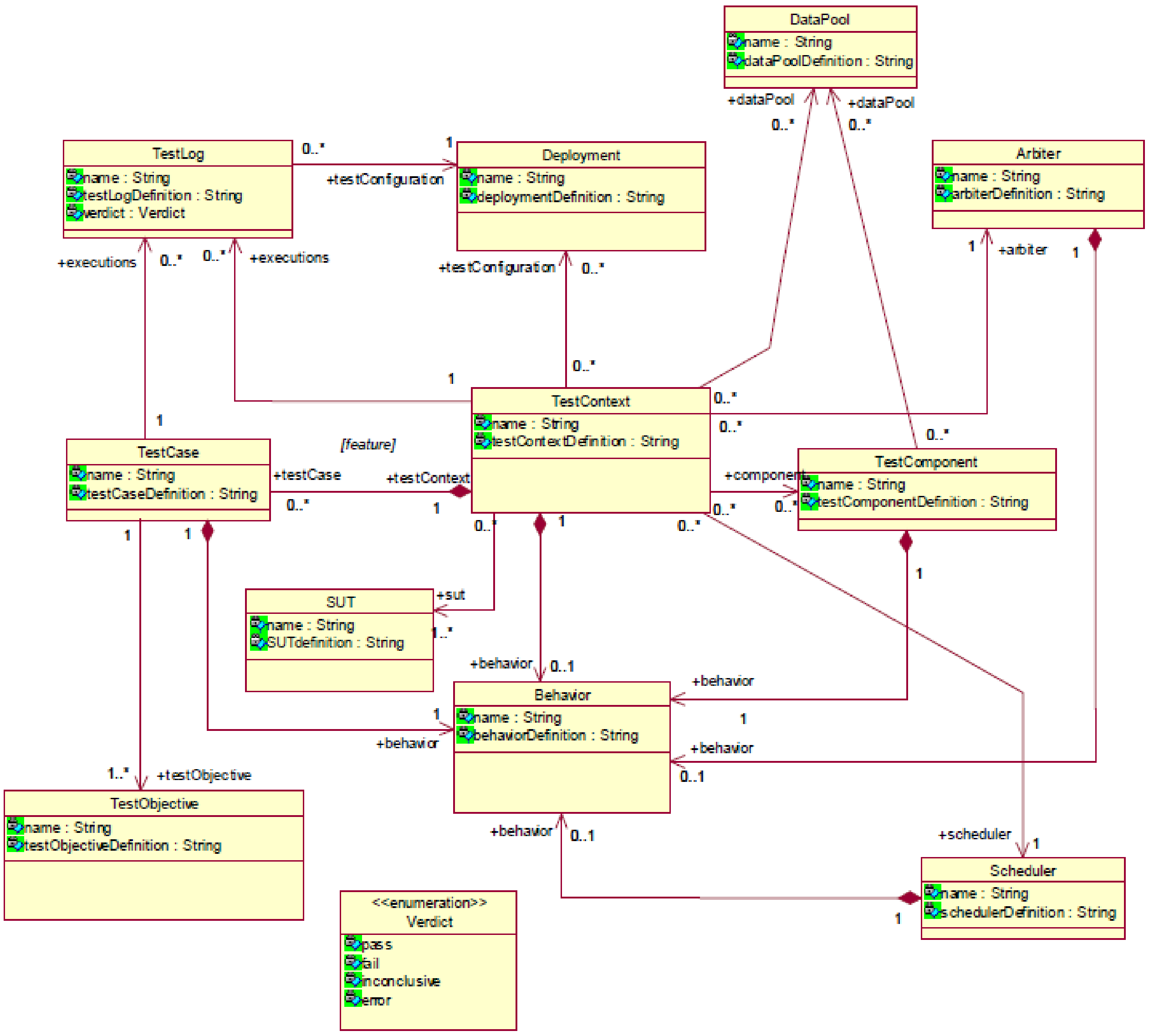


Figure V-1 OMG illustreation of the UTP Meta-Model (Test Architecture and Behavior)

The core concepts of UTP and M4T are quite similar, since both bases on TTCN-3 and are suited for testing. The Figure V-1 represents the UTP meta-model of test structure and behaviour elements illustrated at the UTP website [25]. When comparing to the M4T meta-model in Figure 5-11, one can see the similarities.

A mapping from the Testing Profile to Modelica4Testing is possible but not the other way around. The principal approach towards the mapping to M4T consists of two major steps:

- Take Testing Profile stereotypes and associations and assign them to M4T concepts
- Define procedures how to collect required information for the generated M4T elements.

The following tables compare UTP concepts with existing M4T concepts. Not all UTP language elements have direct correspondence or can be mapped to Modelica4Testing concepts yet.

The mapping table between UTP and M4T consist of four sections:

1. ID of a mapping rule. The ID for mapped elements does have the form “4.x”.
2. UTP element which should be mapped.

3. Modelica4Testing element which matches at most with the UTP element.

a. *UTP to Modelica4Testing Test Structure*

Rule	UTP	Modelica4Testing	Reference
ID	**Test Context**	**Test Context (Definition Part)**	Table 37
4.1	Contains the test cases as operations, its composite structure defines the test configuration.	The definition part covering all test cases, components and related definitions of a test context.	
ID	**Test Configuration**	**Test Configuration**	Table 45
4.2	Compositional structure of the test context element and associations between other components.	Test configuration part within the test context. It connects variables and ports of the, test system interface and the design model together.	
ID	**Test Component**	**Test Component**	Table 42
4.3	A test component is a structured classifier participating in test behaviors.	Is responsible to stimulate the SUT. A test component can be implemented as a dummy. Test components will be divided into main a parallel test components.	
ID	**System under Test (SUT)**	**System under Test (SUT)**	Table 40
4.4	System model which is to be studied by testing. The interaction is running over operation calls.	Contains the design which is to be studied by testing. A test system interface (TSI) is the only connection between the SUT and test components. A TSI configuration defines the same pots as the design model and connects them to the corresponding TSI ports.	
ID	**Arbiter**	**Test Case Termination**	Chapter 5.3.4
4.5	The purpose of an arbiter implementation is to determine the final verdict for a test case.	Since a simulation is running over time the last time to determine the final verdict is the simulation stop time (t_{n-1}). An arbiter is used after the test case evaluation.	
ID	**Scheduler**	**Modelica Tool build-in, MTC**	
4.6	The purpose of a scheduler implementation is to control the execution of the different test components.	A scheduler is a Modelica build-in mechanism. In addition the test system contains all test components. The main test component is the main point of contact which controls the other components.	

Table 65 UTP to Modelica4Testing Test Structure

b. *UTP to Modelica4Testing Test Behaviour*

The section of test behavior includes concepts to specify the behavior of tests in the context of a test context.

Rule	UTP	Modelica4Testing	Reference
ID	**Test Control**	**Test Context (Control Part)**	Table 37
4.7	A test control is a specification for the invocation of test cases within a test context. It is a technical specification of how the SUT should be tested with the given test context.	The control part of a M4T test context. Control execution and assigns the overall test verdict using a verdict mechanism.	
ID	**Test Case**	**Test Case**	Table 47
4.8	A test case is a specification of one case to test the system, including what to test with which input, result, and under which conditions.	A test case is a specification of one case to test the system, including what to test with which input, result, and under which conditions. In M4T a test case is divided into a simulation and evaluation part.	
ID	**Test Invocation**	**Test Case Part Call**	Table 47
4.9	A test case can be invoked with specific parameters and within a specific context. The test invocation leads to the execution of the test case. The test invocation is denoted in the test log.	Using the Modelica function call a test case can be invoked. In addition parametrisation is also possible. A test case part call shall not have any side effects.	
ID	**Test Objective**	**Test Objective**	Table 47
4.10	A test objective is a named element describing what should be tested. It is associated to a test case.	A test case does have an optional test objective as a description. This variable can be implemented in to test context definitions part.	
ID	**Stimulus**	**Test Case Stimulator**	Table 48
4.11	Test data sent to the SUT in order to control it and to make assessments about the SUT when receiving the SUT reactions to these stimuli.	A test case stimulator part checks the test case preconditions and returns test case dependent values for the input variables of the MTC. It will be implemented as a function or a model.	
ID	**Observation**	**Test Case Evaluator**	Table 50
4.12	Test data reflecting the reactions from the SUT and used to assess the SUT reactions which are typically the result of a stimulus sent to the SUT.	The evaluator part of an test case checks the post-conditions of all used components and also the actual test results. Its verdict result will assign the value of the test case verdict.	
ID	**Validation Action**	**Verdict Mechanism**	Chapter 5.3.4
4.14	An action to evaluate the status of the execution of a test case by assessing the SUT observations and/or additional characteristics/parameters of the SUT. A validation action is performed by a test component and sets the local verdict of that test component.	A verdict mechanism is used to assign and update the value of a verdict variable, in order to represent different levels of verdicts, such as an overall test context verdict or a local test case verdict.	

Table 66 UTP to Modelica4Testing Test Behaviour

c. *UTP to Modelica4Testing Test Data and Time Mechanism*

The Test Data section contains concepts additional to UML data concepts needed to describe test data.

Rule	UTP	Modelica4Testing	Reference
ID	**Verdict**	**Verdict**	Table 53
4.13	The verdict is a predefined enumeration datatype which represents a test case, component or overall test result.	The verdict is a predefined enumeration datatype which represents a test case, component or overall test result.	
ID	**Timer**	**Simulation Time**	
4.16	Timers are mechanisms that may generate a timeout event when a specified time value occurs. This may be when a pre-specified time interval has expired relative to a given instant (usually the instant when the timer is started).	Modelica tool simulation time. This is important since the actual version of M4T uses timer to cal a test case stimulator function at a specified time in the simulation.	

Table 67 UTP to Modelica4Testing Test Data and Time Mechanism

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, 24.08.2011

Ort, Datum	Unterschrift