



FAKULTÄT FÜR INFORMATIK
DER TECHNISCHE UNIVERSITÄT MÜNCHEN

Masterarbeit in Angewandte Informatik

Specification, Implementation and Evaluation of a Tool for Multiple Modeling

Tesfaye Regassa





FAKULTÄT FÜR INFORMATIK
DER TECHNISCHE UNIVERSITÄT MÜNCHEN

Masterarbeit in Angewandte Informatik

Specification, Implementation and Evaluation of a Tool for Multiple Modeling

**Spezifikation, Implementierung und Evaluierung eines Werkzeuges
für multiple Modellierung**

Author:

Tesfaye Regassa

Supervisor:

Prof. Dr. Peter Struss

Advisor:

Prof. Dr. Peter Struss

Submission Date: 15. November 2009



Ich versichere, dass ich diese Masters Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I assure that I single handedly composed this Master's thesis, supported only by the acknowledged resources.

Datum: 15 November, 2009

Acknowledgements

I express my sincere gratitude to my supervisor and advisor Prof. Dr. Peter Struss for his untired guidance, supervision and advice on the daily progress of the work, without which it would have not been completed.

I also thank Dr. Alessandro Fraracci for his advice and useful tips.

This work is dedicated to Kedest, Leello and Amanti, without whose support, patience and love I would have not endured the arduous study.

Tesfaye Regassa

"Budooma tabu illee barachuun gaarii dha!"¹

In memory of my late father Regassa Badhasa (1933 – 31.07.2007), whose words above have been my guiding principle.

¹ In Oromo language, it means "Learning is always noble, even how to be the evil-eyed!"

Zusammenfassung

Model-basierte Problemlösungsansätze haben in vielen Anwendungsgebieten Einzug gehalten und werden für viele Zwecke benutzt. Die Phase der Modell-Erstellung ist jedoch beschwerlich und zeitraubend. Daher ist diese Phase die größte Hürde gegen die Erschaffung von Problemlöser für industriellen Einsatz. Obwohl einige Software-Werkzeuge entwickelt und in einige industriellen Projekten eingesetzt wurden, fehlen noch integrierte Entwicklungsumgebung und Werkzeug für Bildung und Integration von Modellen und Algorithmen für die Generierung und Transformation von Modellen.

Das MOM-Projekt (Model Manipulation Environment) wurde gestartet um diese Lücke zu schließen. Das Projekt hat es zum Ziel gemacht, ein Software-Werkzeug zu entwickeln, das generische Schnittstellen zur Modell-Manipulation und Transformation zur Verfügung stellt. Diese Schnittstellen sollen generische Software-Module umfassen, die die automatische Generierung von qualitativen Modellen aus solche numerischen Simulationen wie Matlab/Simulink-Modellen ermöglichen. Die Schnittstellen sollen auch die Einbindung von externen Modellierungswerkzeuge erlauben. Sie sollen auch Software-Komponenten enthalten, die die Beschreibung von Struktur und Verhalten der Modell-Elementen in eine kontextfreie und wieder verwendbare Form ermöglichen. Dies sollen auch ermöglichen, die hierarchische Struktur von System darzustellen, deren Elementen mit mehreren Modell-Beschreibungen assoziiert sind.

Dieser Thesis legte die Grundarbeit für das Projekt und entwickelte die allgemeine Rahmen und implementierte einige der Software Grundkomponenten.

Abstract

Model-based problem solvers have received increasing attention in various application domains and for different tasks. Since model building is a laborious and time-consuming endeavor, creating appropriate models is the main obstacle faced by the attempt to build a model-based problem solver for industrial application. Although some tools have been already developed and applied in industrial projects, an integrated environment or toolbox for model development and integration, and algorithms required for generating and transforming models are still lacking.

The Model Manipulation Environment (MOM) project was initiated, whose goal is to close this gap. It aims at offering a software tool that provides generic interfaces for manipulation and transformation of models. These include generic modules that enable automated generation of qualitative models from a number of numerical simulators such as MatlabTM/SimulinkTM. The abstract interfaces can also be used to integrate other existing tools as specializations. Included are also software components that allow the description of the structure and behavior of model components in a context-free and reusable manner. The idea is to enable a hierarchical structural representation of systems whose elements may have multiple models associated with them.

The current thesis laid the groundwork for the project, designed the general framework and implemented some of the basic software components.

Contents

1.	INTRODUCTION	1
1.1	<i>NECESSITY OF A TOOL FOR MULTIPLE MODELING</i>	2
1.2	<i>OBJECTIVES OF THE WORK</i>	3
1.3	<i>OVERVIEW</i>	3
2.	MODELING AND MODEL-BASED SYSTEMS	5
2.1	<i>TYPES OF MODELS</i>	5
2.2	<i>DIFFERENT APPROACHES TO QUALITATIVE MODELING</i>	6
2.3	<i>COMPONENT-BASED MODELING</i>	8
2.4	<i>EXISTING SOFTWARE FOR MODEL-BUILDING</i>	8
3.	THE MODEL BUILDING PROCESS	10
3.1	<i>BASIC CONCEPTS</i>	11
3.1.1	Structural Entity	11
3.1.2	Terminal	11
3.1.3	Quantity	12
3.1.4	Variables	13
3.1.5	Behavior Description	14
3.2	<i>STEPS OF MODEL BUILDING</i>	14
3.2.1	Physical Abstraction	14
3.2.2	Behavior Abstraction	17
3.2.3	Composing the Model	18
3.3	<i>OTHER MODEL OPERATIONS</i>	21
3.3.1	Qualitative Abstraction of a Numerical Model	21
3.3.2	Generation of Deviation Model	23
4.	MOM SPECIFICATION	25
4.1	<i>INTRODUCTION</i>	25
4.2	<i>FUNCTIONAL REQUIREMENTS OF MOM</i>	27
4.2.1	Composition of Qualitative Model	28
4.2.2	Qualitative Abstraction of Numerical Models	32
4.3	<i>SYSTEM ARCHITECTURE</i>	34
4.4	<i>THE CORE CLASSES AND ARCHITECTURE OF MOM</i>	34
4.4.1	Describing the Structure of a System	35
4.4.2	Associating Quantities	37
4.4.3	Decision-based Associations	38
4.4.4	Physical Entity	39
4.4.5	Choosing Variables and Domains and Creating Model Frames	39
4.4.6	ModelFrame	41
4.4.7	Associating Behavior Descriptions	42
4.4.8	BehaviorDescription	43

4.4.9	Special Domains Built-in.....	46
4.5	<i>MATLABBEHAVIORDESCRIPTION</i>	48
4.6	<i>MOM OPERATORS</i>	48
4.7	<i>A SUGGESTION FOR APPLICATION GUI</i>	49
5.	DESIGN AND IMPLEMENTATION	50
5.1	<i>SYSTEM ARCHITECTURE</i>	50
5.2	<i>SOFTWARE PACKAGES</i>	50
5.3	<i>PACKAGE STATIC STRUCTURE</i>	51
5.3.1	StructuralEntity	51
5.3.2	Terminal	52
5.3.3	TerminalJunction	52
5.3.4	Quantity.....	52
5.4	<i>PACKAGE VALUES AND DOMAINS</i>	52
5.4.1	Interval	52
5.4.2	Value	53
5.4.3	ValueSets	53
5.4.4	Domains	55
5.5	<i>PACKAGE MODELFRAME</i>	55
5.5.1	Variables	55
5.5.2	Behavior Description	56
5.5.3	ModelFrame	57
5.5.4	BehaviorDescriptionAssociation.....	57
5.6	<i>PACKAGE MAPPING</i>	57
5.6.1	VariableMapping	58
5.6.2	DomainMapping.....	58
5.6.3	ValueMapping	59
5.7	<i>INTERFACES TO EXTERNAL ENVIRONMENTS</i>	60
5.8	<i>MOM OPERATORS</i>	60
6.	EVALUATION AND DISCUSSION	62
6.1	<i>THE ABSTRACTION OPERATOR</i>	62
6.2	<i>OTHER BUILT-IN MODULES</i>	67
6.3	<i>MODEL COMPOSITION</i>	67
7.	CONCLUSION AND SUGGESTIONS.....	69
7.1	<i>REVIEW</i>	69
7.2	<i>CONCLUSIONS</i>	70
7.3	<i>OUTLOOK</i>	70
	APPENDIX.....	71
	REFERENCES	74

1. Introduction

Explanation and prediction are central to science. Most of these scientific explanations are based on the so called “classical mechanics”, which is a mathematical formalization of the relation between the quantitative theory and its real-world subject. This variable-oriented mathematical description of objects is operationalized by a measurement procedure which assigns the respective variable a real number. This approach resulted in generally accurate mathematical laws in science, engineering and technology.

However, one central problem in using quantitative theories to correctly describe real world phenomena is that the complexity of the reasoning process increases tremendously with the size and complexity of the phenomena, so that often formulating a true quantitative model is not possible.

The research on Qualitative Reasoning (QR) arose from a desire to capture human abilities to reason about physical systems without detailed numerical models. We are usually able to state, for example, that a bathtub that starts filling with water coming in at a steady rate will continue filling until it either reaches a point where its drain rate is equal to the inflow rate or it will fill up and overflow. We do not need complete knowledge about the dimensions of the tub or the exact flow rates to arrive at such a conclusion. Qualitative reasoning is mainly motivated by this observation, and qualitative reasoning methods have been applied to modeling problems mostly in the sciences and engineering domain ([Kle84], [For84], [Kui86]), but have also been extended to economics and other social science fields. For summary see for example [Hin03] .

The process of (qualitative) modeling encompasses three tasks:

- how to represent *structure of the system* to be modeled. This is the task of model building, which involves analyzing the physical system under question to determine the primitives from which it is built and the structural relationship between them.
- how to describe the *behavior* of the system primitives, i.e. how their physical parameters and their interaction with external influences should be represented;
- how these structural and behavioral descriptions are used to *derive the overall behavior* of the system. This is concerned with the inference of the global behavior of the system from the behavior description of the components. It requires some truth maintenance system.

The present work is concerned with the first two aspects, namely, to implement tools for the description of the structure and behavior of system components in a context-free and reusable manner so that these descriptions could be used for building and behavior prediction of models of more complex systems.

1.1 Necessity of a Tool for Multiple Modeling

Model-based problem solvers have received increasing attention in various application domains and for different tasks. For example, automated diagnostic systems have been already successfully moved into industrial applications. Obviously, creating the appropriate model is the main problem faced by the attempt to build a model-based problem solver for industrial application.

Model building is a laborious and time-consuming endeavor. For example, model building for the sake of diagnosing a device is of no use if the cost and time so incurred outweighs the effort of 'manually' inspecting the device. This issue calls for a library of context-free, re-usable model fragments that could be integrated into different projects of similar nature. Automated generation of qualitative models from quantitative (numerical) models of systems and components that often exist in industrial practices could also assist in the model-building process [Str02].

Research in qualitative reasoning is mostly concerned with modeling and simulation, but less attention is given to the process of building models itself. Although some tools have been already developed and applied in industrial projects, an integrated environment or toolbox for model development and integration, and algorithms required for generating and transforming models are still lacking.

In the closing statements to his paper in [Str08], Struss underlined as one of "the most essential current challenges" in model-based problem solving, as follows:

*"All projects that solved a problem relevant to industrial practice had to build component models. However, to our knowledge, none of them developed a serious library of behavior models that could be easily reused in another project. ... Especially, a well-founded theory and methodology for developing **reusable** model libraries is needed."*

To close this gap, and offer a kind of environment that allows manipulating models in a systematic way by re-using, adapting, or modifying pre-existing ones, a project termed **Model Manipulation Environment (MOM)** was initiated at the MQM Group, Faculty of Information of the University of Munich. The project aims particularly at developing tools for the transformation of numerical

models into qualitative ones, and for the abstraction of models according to particular application requirements. This environment should also support the users by recording and inferring relations between the various models resulting from manual modeling or automatic transformations.

1.2 Objectives of the Work

The current work aims at laying the ground work for the tasks laid down in the "Requirements Document" for the MOM project [MOMa]. The key issues for the environment is to provide a **generic interface** independent of specific modeling formalisms and systems that can be shared by various model operators. These abstract interfaces can then be used to integrate different existing tools as specializations. For instance, an algorithm that generates a qualitative abstraction from a numerical simulation model can be specified and implemented in a single and general way so that it can exploit computational capabilities of different numerical simulators, such as Matlab™ or Spice. Similarly, a finite relational model can be generated which can then be turned into representations that are processed by FCS (finite constraint satisfaction) algorithms or MMD (multiple decision diagrams) operations. In more detail, the framework has to allow for a hierarchical structural representation of a system whose elements may have multiple models associated with them.

The project aims at implementing the general framework and some of the operators, and re-design and re-implement existing algorithms in this framework.

1.3 Overview

Chapter 2 presents an introduction to modeling and to important concepts in model-based reasoning. It reviews existing approaches to qualitative modeling and enumerates some of the significant modeling software currently in use. Chapter 3 discusses the model building process. The chapter introduces basic concepts of compositional modeling and the various steps taken and the operations needed to build a model of a given system. It discusses the background principles and some of the important operation in modeling.

Chapter 4 details the specification of the components of MOM. In this section, basic use cases relevant to the major operations expected in MOM are outlined and described. In addition, the concept of decision-based model building is introduced and illustrated. The section also identified object classes for later design and implementation.

Chapter 5 discusses key points of the software design and implementation, and outlines which of the generic interfaces and basic software components have been designed and implemented. Within the

framework of the current thesis, much of the basic components have been implemented, including classes related to structural entities, variables, domains and values. Implemented are also generic interfaces for behavior description and the service classes for variable and domain mapping. The abstraction operator for abstraction of qualitative relation from a Matlab/Simulink model has been implemented and tested.

Chapter 6 presents sample runs of some of the implementations, specifically of the abstraction operation, and evaluates the results. The last chapter presents some conclusions and points some lessons for future work.

2. Modeling and Model-based Systems

Modeling may be defined as the application of methods to analyze complex, real-world issues in order to explain particular questions or predict unobserved properties, (e.g. internal states, faults, etc.) or what might happen with various actions. According to Kuipers, a model is a finite description of an infinitely complex reality, constructed for the purpose of answering particular problems. It embodies 'a closed-world assumption' – that the set of objects and relations in the model include everything necessary for that purpose [Kui94].

In general, a model may include any mental construction of how a real-world (or theoretical) system would be described, but here we would limit the concept to the commonly understood (computer-based) mathematical or logical description of a given reality. In this section, some reviews on model types and a short summary of modeling software would be presented.

2.1 Types of Models

Computer-based models could be classified in several ways. As commonly used in practice, quantitative (numerical) models are termed *deterministic* if the outputs are only dependent on the initial input, whereas so called *probabilistic* or *stochastic* models include elements of probability such that the results exhibits random effects.

A model could be *dynamic*, which simulates a system over time. In a static model, time is not considered, so the model is comparable to a snapshot. A dynamic model could also be discrete or continuous, depending on whether the time variable changes in incremental steps or continuously.

Relational modeling represents data as a collection of predicates over a finite set of predicate variables, i.e. as mathematical n-ary relations. It is the central idea underlying modern databases.

Models are also classified according to how the model parameters are represented: *Quantitative* (numerical) models are usually standard mathematical descriptions of continuous behavior in the form of (differential) equations or inequalities combined with logical sentences. In such models, variables may have an infinite domain, i.e. sets of natural or rational numbers or intervals thereof. In contrast, a *qualitative model* is an abstract characterization of the real system in qualitative yet formal terms, that is, in terms of a finite set of qualitative values and relationships. In model-based systems (see below), sometimes hybrid or mixed models that combine quantitative and qualitative

aspects of a system are used. The terms “model” and “model-based system” are being used in diverse contexts. Picardi, et al. [Pic07] provides a brief discussion on definitions of essential terms, and an overview of the main types of models typically associated to model-based systems.

2.2 *Different Approaches to Qualitative Modeling*

Using different approaches towards representing structural primitives of physical systems and their behavioral functionalities, a number of researchers produced various methods for describing and simulating systems qualitatively. Among the different approaches which were influential in qualitative modeling research are notably, ENVISION [Kle84], Qualitative Process Theory [For84], and QSIM [Kui86]. Their principal differences and commonalities are summarized by Struss [Str97].

Struss, in [Str97], surveys some of the important techniques that have been applied to qualitatively model physical systems, and explains principles of model-based reasoning in diagnostic systems. In this survey, he identified three qualitative modeling "ontologies" and analyzed systems representative of each ontology, their principal differences and commonalities.

Accordingly, these are:

- 1) The component-oriented ontology, typically employed in ENVISION [Kle84], which describes the system as a collection of components which have connective ports for interaction between them. The behavior of the components are internal laws, decomposed into distinct states or operating regions;
- 2) the process-oriented ontology, exemplified in Qualitative Process Theory [For84, For86], describe changes in the physical system in terms of processes representing active changes, and sets of interacting objects; and
- 3) constraint-oriented “ontology”, representative of which is QSIM [Kui86], in which the compositionality issue is not considered. QSIM rather uses sets of *qualitative differential equations* (QDEs), which are defined as an abstraction of ordinary differential equations, and sets of constraints relating these functions.

The Component-oriented ontology, inspired mainly by network theory, presumes a fixed topology of the physical system, with fixed, well-specified set of components. This does not hold true for certain domains. For example, in the idealized water treatment system below, although the reservoir,

the pump, tanks and other appurtenances can be considered as components, the system behavior is affected by the chemical, biological and physical processes taking place in these components. Nutrients, algae and water layers are not fixed "components", as these could be created or destroyed dynamically. Algal blooms would be promoted by the inflow of certain nutrients and sunshine, themselves introducing extra organic matter into the reservoir. Dying algae settle to the bottom inspiring aerobic bacteria that consume O_2 in the lower water layers, thus creating anaerobic and low pH situations at the sediment layer, that in turn resulting in the re-dissolving of iron and other minerals from the sediment. These could in turn influence algal growth further.

This example demonstrates the limits of component-based ontology. The system includes a dynamically changing pattern of active processes, objects, substances, etc., each being created or destroyed. However, a process is considered as some elementary phenomenon, which, as argued by Struss in [Str08], can be modeled independently of others and is, therefore, suited to compositional modeling. There are two important points to consider, however. First, for a process to take place, certain structural and quantity preconditions, i.e. particular configuration of interacting objects and constraints on involved quantities, should be fulfilled. Secondly, each quantity involved in a process has only a partial effect. This means the overall effect can only be determined after all individual influences are determined, because the combination of effects may not be linear.

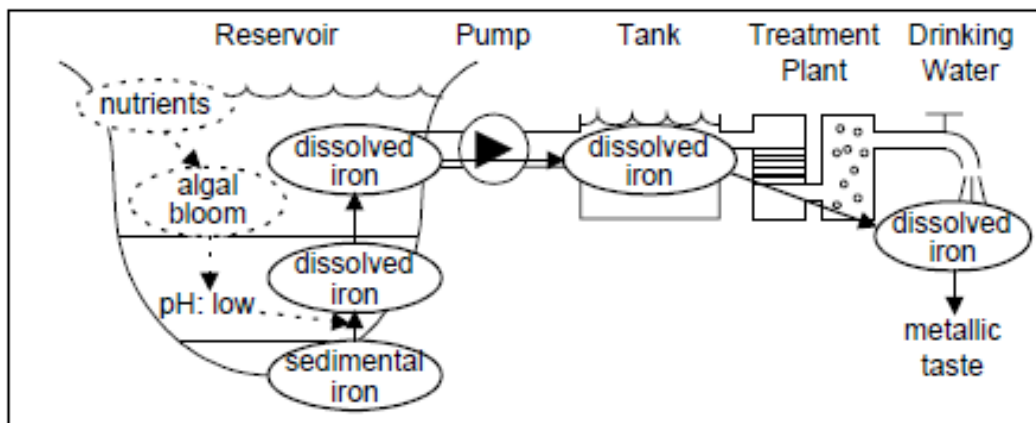


Figure 1 An example scenario. From [Heller & Struss 2001]

Therefore, in process-oriented modeling, simulation needs to be conducted iteratively, starting with some initial conditions, processes being activated and deactivated according to the actual structural and quantity conditions, thus establishing a permanently changing *dynamic structure*.

The aim of the work in this thesis is not simulating a system, but primarily concerned with establishing an environment for model construction and operation. The primary goal is to provide a framework for setting up a library of context-free, reusable model components, which could be used independent of the modeling ontology.

2.3 *Component-based Modeling*

Much problem solving involves **domain-specific knowledge**, i.e. expert knowledge relevant to a certain *domain*, environment or situation or class of problems. A model-based system is usually designed to incorporate the domain-specific knowledge as a model library that contains a declarative, composable representation of the behavior of elementary constituents of systems in the domain with a maximum of versatility and re-use to different system instances and for different tasks.

In addition, a model-based system aims at providing **problem solving engines** that support or automate the exploitation of such models to solve certain tasks (See [STR08]). Thus, the separation of the problem-solving algorithm from the structural description and the means of model composition are central to model-based systems.

2.4 *Existing Software for Model-Building*

Diverse modeling and simulation techniques have been applied in science and engineering. The aim here is not to present a review of all available modeling and simulation software, but to point out typical ones that have found wide usage for building models of physical system.

1) **SPICE** (**S**imulation **P**rogram with **I**ntegrated **C**ircuit **E**mphasis) is a computer simulation and modeling program used by engineers to mathematically predict the behavior of electronics circuits. XSPICE, an extension to the SPICE language, allows behavioral modeling of components which can drastically improve the speeds of mixed-mode and digital simulations. **Multisim** from *National Instruments Corp.* is based on SPICE and XSPICE and provides additional convergence and speed improvements to complement these powerful simulation languages [Spice].

A circuit must be presented to SPICE in the form of a net list, which is a text description of all circuit elements such as transistors and capacitors, and their corresponding connections. Modern schematic capture and simulation tools such as *Multisim* allow users to draw circuit schematics in a user-friendly environment, and automatically translate the circuit diagrams into net lists.

2) **Simulink™** is a program for graphical modeling and simulation of dynamic systems using signal flow. It is usually used to solve linear and non-linear differential and discrete difference equations. It is coupled to Matlab™ as a special toolbox. MATLAB is a high-level technical computing language and interactive environment for numerical computation. see <http://www.mathworks.com/>

3) **Modelica** is a non-proprietary, declarative, object-oriented multi-domain modeling language, with visual component programming. Acausal class concept of Modelica makes its library classes more reusable than traditional classes containing assignment statements where the input-output causality is fixed. [*Peter Fritzson, et al. 2009*]

4) One of the commercial software systems for model-based diagnosis is **RAZ'R** from the company OCC'M Software GmbH [Razr].

5) RODON from UpTime Solutions AB, Sweden, is another commercial Model Based Reasoning (MBR) tool. It is used to perform diagnostics and reliability analyses such as FMEA [Rodon].

Others, more of educational software systems, cited in the literature include:

CYCLEPAD (Forbus et al., 1999), an ‘articulate virtual laboratory’ that supports engineering students in designing and analyzing thermodynamic cycles. It enables students to construct such systems by assembling components from a predefined library.

MOBUM, a model building environment aimed at supporting learners in building qualitative simulations, a prototype implemented in JAVA [Bassa, et al.].

3. The Model Building Process

Developing a conceptual model encompasses identifying meaningful physical constituents of the system, and the relevant quantities, understanding how these interact in determining the system's behavior, and deciding upon the appropriate aspects of such quantities and space of values.

The first step towards model building involves modularization. Depending on the purpose of modeling (design, diagnosis, etc.), analysis of the system could be a top-down or a bottom up process. In diagnosis and similar model-based tasks, the first step is to discover the primitive constituents of the given system. The depth to which the system should be decomposed depends on the level of granularity demanded by the task under question and the desired reusability of models. In the case of design, the task involves experimenting with various compositional arrangements of known parts to approximate a system with a preset behavior. An important step is, thus, to determine the relevant structure of the system in terms of components, objects, relevant processes, etc., and the physical parameters involved.

In general, the model-building process involves first finding out, on a conceptual basis, what constitutes the physical system in question, how these constituents are interconnected, and how the individual constituents behave. The detailing should be made to such a degree of decomposition necessary to solve the problem at hand, but also the availability and/or targeted production of generic, re-usable elements for a model library.

In effect, modeling involves two separate tasks: building a library of context-free and re-usable model elements, and using these elements to build a model of a system of higher complexity for the purpose of behavior prediction.

For building a library, the first step involves describing the behavior of the elementary constituent entities in a general context-free manner. It also includes considering all possible ways the entity may behave and the different formalisms for describing its functionality. This fully utilizes the core idea of multiple modeling.

Multiple modeling is an approach to produce complex models by piecing together a number of simpler subsystems. It aims at building systems which operate robustly over a wide range of operating conditions by decomposing them into a number of simpler linear modeling problems, even for nonlinear ones. Decomposing knowledge of a domain into small fragments can greatly

simplify the task of generating and using a large domain model. It has computational advantages, in that it lends itself to adaptation, and allows direct incorporation of high-level and qualitative domain-specific knowledge into the model.

In this section, general concepts of structural entities, connections, quantities, and variables as related to quantity and "quantity space" will be presented. The analyses can be understood from a component-oriented perspective, but most of the concepts can be applied independently of this.

3.1 Basic Concepts

From a compositional point of view, a real-world system can be seen as a structural hierarchy of *entities* (physical components, objects, processes, etc.) with part-of-relationships, and in terms of how their internal states are reciprocally influenced. This section recounts the basic constituting elements and relations.

3.1.1 Structural Entity

A **structural entity** represents one of the units that are the building blocks of a (physical) system to be modeled. It can be an aggregated one, i.e. it could itself be composed of several other structural entities or it is an elementary one that does not need to be further decomposed. An aggregate structural entity can be seen as a set of interconnected elementary or other aggregate entities.

Physical entities usually possess **quantities** which represent **state variables** – internal attributes that may change due to external influences (input), e.g. temperature, temperature-dependent resistance, and **parameters** – attributes or properties of the entity that can be taken as constant under the situation under consideration, e.g. a pipe has a diameter, a resistor has a certain resistance.

A structural entity is idealized to have points of contact with its surroundings – called **terminals** or **interfaces**. It is presumed that all of its interactions with the outside world will take place through such contact interfaces. A **terminal junction** is a node where such two or more terminals of differing entities make contact.

3.1.2 Terminal

A **Terminal** (interface) – is a real or conceptual point of contact of a component with other entities in its surroundings. A structural entity has at least one terminal. Over a single terminal, an entity can share with another entity a number of quantities that mediate their reciprocal influences. Electrical circuits and hydraulic networks are paradigm examples. At an electrical contact point, current will

flow through. Fluid flow will take place into or out of the pipes connected at a junction. At a sensor terminal, one is usually interested in the transmitted signal, single quantity. Also the connected entities share voltage and pressure, respectively.

A given physical entity usually has a definite number of quantities with which it interacts with its surroundings. The type of these quantities determines the types of its terminals. The type of a terminal in turn determines the type of connection that can be made. For example, an electrical terminal (wire) can not be simply connected to, say, a mechanical shaft without an intermediate component (electric motor) between them. A **terminal type**, thus, indicates the compatibility of a connection, i.e. whether the connection is possible or meaningful.

Certain physical elements in classical component-connection modeling usually possess interfaces through which two inseparable quantities are exchanged. Typical examples are voltage & current in electrical elements; pressure & mass/volumetric flow rate in hydraulics and pneumatics; force & position/displacement and torque & angular position in mechanics, etc.. Connecting such terminals demands extra consistency consideration (see below). Paying attention to this fact, one can categorize terminal types according to the general technical field in which the entity is used, e.g. electrical, mechanical, hydraulic etc.. These could be further sub-classified hierarchically.

3.1.3 *Quantity*

Quantities are the perceptual means by which we understand the world around us. The term quantity is differently used under various modeling concepts. Paritosh [Paritosh 2003] attempted to ground the basis for representation of quantity in linguistics and qualitative reasoning. As used in model-based systems, a quantity is an attribute of a physical system. It represents parameters, states or physical properties or information exchanged between entities.

To be context-free and reusable, the behavior of a component must be specified independent of the context in which its instances may be used. In this work, a quantity is not used as a variable but as an attribute of an entity without specificities about value and value domains. The intuition is to provide just the primitives for an ontology for behavior models of the entities. Variables are then defined as different ways of characterizing properties of quantities (e.g. magnitude, derivative, deviation, etc.). This way, the structural description of a system is unique and remains unchanged when different ways of modeling the behavior are chosen.

Generic quantities have been defined and used, above all in the theory of bond graphs [Karnopp et al., 1990] to exploit structural isomorphism among variables and constraints describing flow of

power and energy across a range of scientific domains. Accordingly, one can classify quantities (at least those that are exchanged at terminals) into two: Flow types, which are a measure of a certain flow of material or energy, such as electrical current (electrons), mass/volumetric flow rate (fluids), heat, etc., and effort types that indicate some potential level such as pressure, voltage or temperature difference. The importance of this distinction is that, during connection of entity interfaces, flow types must obey the law of conservation that, at the junction, they must sum to zero (Kirchhoff's current law), whereas effort (potential) types must have the same level at a given point (Kirchhoff's voltage law) [Kui94]. This general classification of a quantity as either a flow type or not is used in Modelica [Fritzson 2006].

Thus, at a connection between Terminals:

- Non-flow variables must obey *Equality coupling*: $v_0 = v_1 = v_2 = \dots v_n$
- Flow variables must *Sum-to-zero* : $i_1 + i_2 + i_3 + \dots + i_k + \dots i_n = 0$
- The sign of a flow variable is *positive* when the 'flow' is *into* the component, and negative if it is leaving the component.

Special problem in applying the connection rules comes from the flow-type quantities. If more than two flow quantities are involved at a given node, the sum-to-zero equation can not be solved directly.

On a different dimension, quantities can be categorized according to the type of physical properties they represent, e.g. pressure, force, energy, etc.. Such a **category** is essential to identify corresponding quantities at a terminal connection.

3.1.4 Variables

In a mathematical sense, a **variable** is a name or a symbolic representation of a parametric value that may change. In models of physical systems, it is often a real-valued representation of a selected attribute of a system. Variables are used to describe the behavior of a system and/or its elements. Quantity and variable are often used interchangeably, and in Modelica, quantity is used to mean category of a physical attribute represented by a variable. As discussed above, the distinction between the two is necessary to distinguish between the structural description of a system and the space of quantities used to characterize its physical properties from special choices of how to actually model its behavior. A quantity simply represents a physical property, whereas a variable characterizes it in different ways (e.g. magnitude, absolute value, derivative, deviation, etc.). A

variable is thus related to a quantity through these *quantity aspects*. There is no fixed set of aspects a quantity should have, and every quantity does not possess all of them. Also, for a given quantity, the number of aspects to consider depends on the intended model.

In one way, aspects can be seen as **attributes** of a quantity or as different **forms** of the same quantity. Seen differently, especially in a physical world, a derivative of a quantity may be another quantity (e.g. if x is position of an object, $dx/dt = \text{velocity}$; $d^2x/dt^2 = \text{acceleration}$, etc.). In such a situation, whether to model the derivatives as independent quantities or as aspects of one central quantity will be a question of design.

3.1.5 Behavior Description

Behavior description is some mathematical, logical or other formalism that characterizes the state of the system or the sequence of system states occurring over a particular span of time. It represents a concrete technical realization of the mathematical behavior or functionality in a specific system. Component-connection models decompose the task of determining all possible behaviors according to the principle: *The behavioral repertoire of a mechanism is determined by the behavioral repertoire of its components, constrained by their connections* [Kui94]. This reductionist approach implies that one can obtain behavior model of a system as the composition of behavior models of its sub-systems in a bottom-up fashion (e.g., by transforming a constraint network to a single constraint relating state and interface variables of the aggregate). Although some times the whole is not the sum of its parts, this approach is also used in process-oriented modeling to build first estimates in iterative solution findings.

The next section introduces some of the basic principles mentioned above using real examples, and illustrates how complex models could be built up using these elements.

3.2 Steps of Model Building

3.2.1 Physical Abstraction

As is stated in the introductory remarks, model building involves first analyzing the physical system under question to determine the primitives from which it is built and the structural relationship between them, and to describe the behavior of these primitives which could later be used to build the model of the overall system.

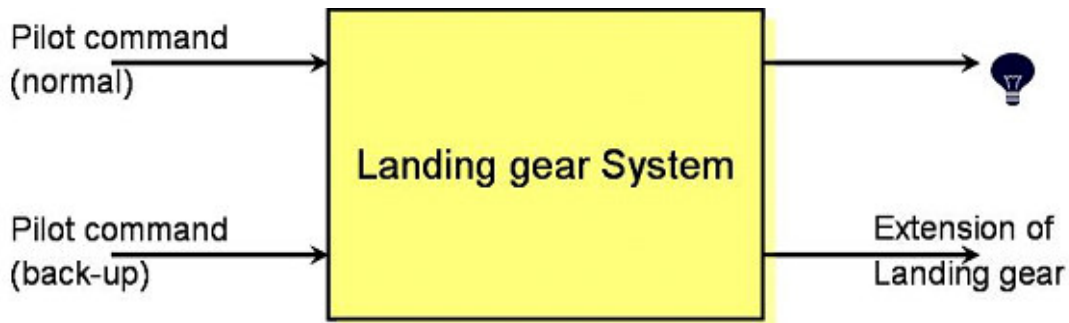


Figure 2 System structure of an Aircraft Landing Gear System

The concepts can best be elucidated using a modeling example. For this purpose, we analyze an aircraft landing gear that has been modeled in a previous work for Failure Mode and Effects Analysis (FMEA) qualitatively [Fraracci & Struss 2006] and quantitatively in Matlab™/Simulink™ [Fraracci 2008].

Seen as a black box, the landing gear system can be depicted as in Figure below. It takes an

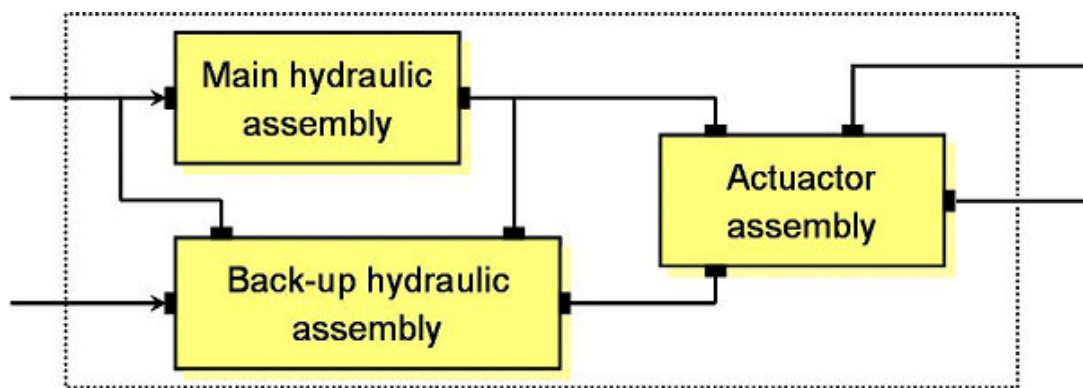


Figure 3 Major elements of the landing gear system

electrical command from the pilot (extend or retract) which will be effected into the extension or retraction of the landing gear and a status signal to this effect. The system is also provided with a backup command in case the main one fails to produce the desired action.

The purpose of the whole system is to extend the landing gear. It is composed of the main hydraulic system consisting a main pump and a check valve, an auxiliary hydraulic system as a back-up and an actuator assembly that extends or retracts the landing gear.

The next task is to decompose the system into its building blocks. The first level of decomposition reveals the three main components, and further decomposition details elements that make up each assembly, as shown in the figures 3 and 4 [Fraracci06].

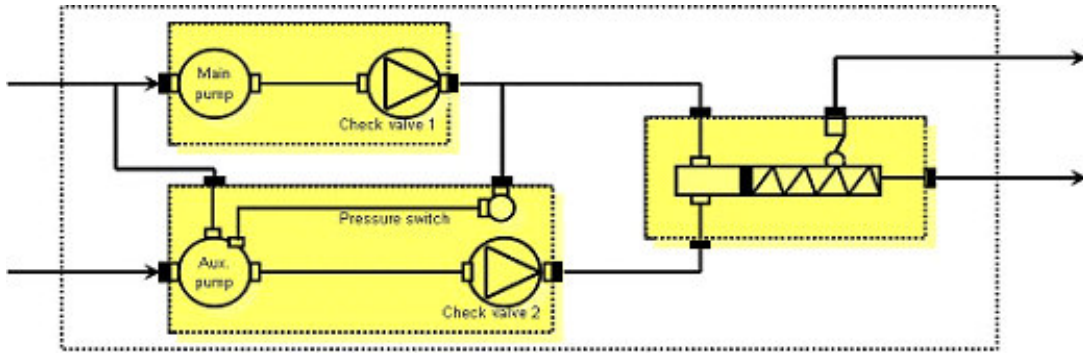


Figure 4 Detail structure of the landing gear system

Excluding the electrical cables and the pipes connecting the different elements, one can identify 7 different important components, including 2 pumps, 2 check valves, an hydraulic actuator, a pressure sensor (switch) and a position sensor for feed back. One can include the pipes into the list of components to be modeled, if for example one wants to model leakage or other pipe behavior. This level of granularity is sufficient for the present purpose.

Some components appear in the system more than once, which exemplifies the need for context-free models that can be reused in a variety of settings. The process of model-building will further be illustrated taking the example of a check valve, which is abstracted as shown below [Fraracci06].

CHECK VALVE	TERMINALS	
	Inlet	inlet fluid (type hydraulic)
	Outlet	outlet fluid (type hydraulic)
FUNCTION - Allow flow only in one direction, i.e. from inlet to outlet - Prevent reverse flow from outlet to inlet.		

Mathematical function describing flow through the valve:

- Normal operation

$$Q_{in} + Q_{out} = 0$$

$$\begin{cases} Q = \alpha \cdot \sqrt{p_{in} - p_{out}} & p_{in} - p_{out} > 0 \\ Q = 0 & p_{in} - p_{out} \leq 0 \end{cases}$$

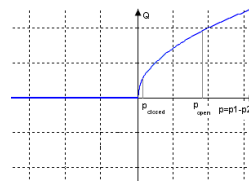


Figure 5: Structural Abstraction of the Check Valve Function

The next important step is identifying attributes of the isolated entity and discriminating between those that play a role in the intended model and those to be considered irrelevant. Identifying

relevant quantities and their domains and abstracting their physical interplay into some logical or mathematical formulation is needed for successful modeling. For a structured approach towards model-building, see [Bredeweg et al; 2006].

The abstracted entity - check valve - thus possesses two hydraulic terminals, and at each terminal, fluid flow will take place into and out of it depending on the pressure difference. Each hydraulic terminal is thus associated with two quantities – pressure and volumetric flow rate. Other influences can be ignored. The relationship between these variables is given by the equations shown in Figure 5.

3.2.2 Behavior Abstraction

The central idea of qualitative reasoning is the abstraction of the value domain of real-valued variables into a finite set of ordered symbols representing **qualitative values** that make relevant behavioral distinctions. This is performed by identifying for each variable a set of distinguished points called *landmarks*, where significant changes in values of the variable take place. One way to specify a qualitative value of a variable is by either a landmark or an open **interval** between two adjacent landmarks. The finite, totally ordered set of all the possible qualitative values of a variable is called its **quantity space** or **domain**.

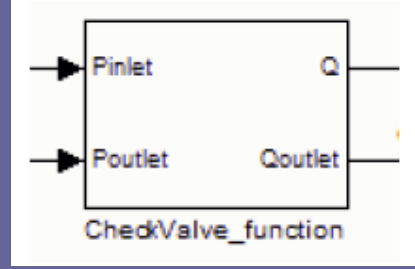
Different domain abstractions capture different ways of reasoning qualitatively. Partitioning real numbers according to signs, in particular, is extensively used by the qualitative reasoning community to formalize reasoning about tendencies: +, –, and 0 are used when the variable increases, decreases, and does not change, respectively. Different intervals that partition real numbers can be identified as demanded by the task under consideration.

In the current example, from physical considerations, the quantity – volumetric flow rate – represents flow of matter which implies that the sum of flows into and out of a joint must balance. Following the concepts used in bond graphs and in Modelica, this quantity will be typed as 'flow'. Similarly, pressure represents the effort or potential. Physically, pressures can not have negative values, whereas the flow rate can have signs following the convention that it is positive if flow is into, and negative if flow is out of the entity.

Thus, the domains chosen for qualitative modeling are: $\{0, +\}$ for the pressure P to indicate "zero pressure" and a "positive pressure", respectively, and $\{-, 0, +\}$ for the flow Q to indicate "flow out of the component" (-), "no flow" (0), and "flow into the component" (+). Manual

computation then produces the tabular relation between the inlet and outlet pressures and the inflow as shown in Table 1. The outflow is simply the negation of the inflow [Fraracci06].

Table 1. Qualitative description of the check valve function

	P_inlet	P_outlet	Q	Comment
	0	0	0	No pressure; no flow
	0	+	0	pressure at outlet, but reverse flow is prevented $P_{inlet} < P_{outlet}$
	+	0	+	Flow from inlet to outlet. Pressure at inlet and no pressure at outlet
	+	+	0	No flow if $P_{inlet} \leq P_{outlet}$
	+	+	+	Flow from inlet to outlet if pressure $P_{inlet} > P_{outlet}$

Manual compilation of such a relation, especially for complex systems, is laborious and time consuming. One way of saving such an effort is abstracting qualitative models from existing numerical (quantitative) models automatically. One of the aims of this work is to provide a general interface to external modeling tools that would enable such qualitative abstractions. Section 3.3 describes general principles of abstraction of a qualitative model from a numerical one. Implementation of an interface to Matlab™/Simulink™ for the purpose of such abstraction is part of this thesis. The abstraction operation will later be demonstrated using the check valve function which has been implemented in Simulink™.

The idea behind component-connection modeling is to compose behavior model of a system by compiling (possibly automatically) behavior models of its sub-systems in a bottom-up fashion. In the chapters that follow, a general framework for compilation of models from models of constituent elements will be presented.

3.2.3 Composing the Model

The association between a structural entity and a behavior description is a multi-dimensional issue. On the one hand, a given behavior description may describe only a certain functionality of the entity, and only in conjunction with others that the overall behavior of the structural entity (even of an elementary one) would be described. For example, for an electrical resistor, one behavior description may represent Ohm's law, another one may describe Kirchhoff's law and a third one may relate its resistance to temperature or voltage. On the other hand, the same functionality or behavior

may be described using different formalisms (e.g. qualitative versus quantitative), varying degrees of granularity, different modes of operation (e.g. OK or faulty), or using different modeling tools. On another level, the given structural entity may be composed of other sub-entities (i.e. is an aggregate one), in which case its overall behavior will be given by a selected combination of the behavior models of its sub-entities. Therefore, fixing a behavior description of a given entity for a particular modeling task requires decisions on such different dimensions.

At the heart of compositional modeling strategy lies the wish to instantiate and assemble as needed a domain model from elementary models taken from a library, to form a system model based on a combination of the properties of the physical scenario and the modeling assumptions appropriate to the task.

[Falkenhainer and Forbus, 1990] presented a compositional modeling framework for modeling continuous physical systems. The framework is based on three classes of predicates they call "grain assumptions", "ontology assumptions", and "approximations and perspectives". Grain assumptions control which objects to explicitly consider in an analysis (through existence, part-of or contains relations). Ontology assumptions are used to divide the physical system into a number of system types. "Approximations and perspectives" are simplifying assumptions used to construct elementary reusable models which can be instantiated and assembled as needed to form a "scenario model" - a complete and consistent set of modeling assumptions. Approximations are assumptions used to construct simplified and easier to use (but less accurate) models, whereas perspectives represent modeling choices, such as how particular objects are modeled. As an example of perspective, a fluid valve can be modeled either as a discrete, on/off switch or as resistance that can vary continuously.

[Addanki et al., 1989], on the other hand, presented a paradigm where domain knowledge is represented as a graph of models linked by directed edges. Each node represents a model of the system being analyzed, and each edge indicates which assumptions differ between the models it connects. Addanki et al. proposed methods that automatically change models when the running model is inadequate. For example, whether a laminar or a turbulent flow regime prevails in a pipe carrying a fluid depends on the Reynolds number, which is a dynamic parameter of the flow process. The two flow regimes are characterized by different models, but how to model the flow dynamics as such can not be chosen beforehand. Therefore, they suggest, the running model process must possess: 1) the ability to detect conflicts, 2) the ability to determine how parameters must change in order to eliminate the conflicts, 3) the ability to represent how assumption transitions

affect parameter values, and 4) the ability to use this knowledge in selecting the next model. see also [Addanki et. al., 1991] .

What Addanki et. al. refers to as a model is, however, what in the compositional modeling framework of Falkenhainer & Forbus is a complete scenario model. Since it is not based on compositionality, the graph of models approach is inflexible and leads to too many scenario models difficult to enumerate. The approach taken by [Falkenhainer and Forbus, 1990], though it uses compositional modeling, builds a number of decisions, that have to be taken at different levels, into the elementary models.

Whether to model a valve as a discrete or a continuous process is a decision to be taken by the human modeler from the outset, whereas what flow regime prevails in a pipe is governed by the dynamics internal to the fluid flow. The first aspect requires that there are different pieces of models for the same physical object to choose from, whereas the second aspect is concerned with how to describe the same model under different scenarios in which the model may be active and how transitions between the different characteristics are effected. This second aspect concerns basically dynamic systems, but generally, it involves alternative behaviors that a given may possess under different settings. Our current effort is to present a consistent and unified approach.

As stated earlier, how to model a given system involves a multi-level decision. In other words, a modeling environment (at least one that enables multiple modeling) must provide the modeler with all possible alternatives at each level. Identification of the decision levels and enumeration of the alternative options at each level enables provision of generic and context free model building blocks and is an essential part of a multi-modeling environment. An environment for model building must thus provide the modeler with a library of elementary models fitted with (all) possible alternative behaviors.

The current approach is, thus, to divide the model-building process into four relatively distinct layers, the transition from one to the other being coupled to user decisions. The central assumption is that a set of elementary model building blocks are available in a library with alternative, consistent combination of states and parameters, which the modeler can select from. We call this approach Decision-based model building. Details will be presented in the next chapter.

3.3 Other Model Operations

The whole purpose of modeling is to simulate a given physical system on the computer. In an abstract form, one can state an operation of simulation as computing a given set of output variables from a given set of input variables. How the output variables would internally be manipulated is the essence of the model design.

In a simple way, one can state a model as a simplified mathematical formulation of the physical structure of the system, the processes taking place within it, and its interaction with the surrounding.

As stated before, qualitative models use finite qualitative relations over variables. Hence, the behavior model of given component or system is a relation R over the set of system variables v_i , each with its value domain $DOM(v_i)$:

$$R \subset DOM(v_s)$$

where $DOM(v_s) = DOM(v_1) \times DOM(v_2) \times \dots \times DOM(v_n)$, representing all possible states or behavior space of the system or component.

So, a relation R (or *constraint*) is a subset of the possible behavior space; an element of a relation, $val \in R$, is a *tuple* and a set of tuples forms a relation. The ordered set of variables that a relation R is defined on is called *scheme* of R and is denoted $scheme(R)$; the notation $R.v_i$ makes explicit that a certain variable v_i belongs to $scheme(R)$. The operations on relation then involve projection, selection and conjunction of some sets of tuples [Fraracci08].

On a different level, modeling operation involve abstraction of a qualitative relation from existing numerical models and transforming a given model into another. A few of such operations are illustrated in the next section.

3.3.1 Qualitative Abstraction of a Numerical Model

The main goal of qualitative abstraction is the automated generation of qualitative models from existing (numerical) simulation models. The level of abstraction is task dependent, i.e. the distinctions in the domains of the system variables depends on the requirements of the task. The main task of abstraction is finding variable domains that maintain only those distinctions that are necessary to determine the required distinctions of the target variables. In other words, we want to drop distinctions in the domains of the fine-grained model without losing its inferences concerning the target partitions. Thus, the main task of qualitative abstraction is finding *"the distinctions in the*

domains of the system variables that are both necessary and sufficient to achieve a particular goal in a certain context and under given conditions" [Str02].

Usually, the partitions of the domain can be given by (finite) sets of landmarks that define qualitative values as intervals between adjacent landmarks. The abstract model will then contain a subset of the landmarks of the original model but maintain the predictive power with respect to qualitative values of the target variables.

There are a number of mathematical environments for numerical modeling, most notably Matlab™/Simulink™ and Modelica. A number of papers have been written on abstraction of qualitative models from Simulink™ simulation models. Sachenbacher and Struss presented a theoretical background and implementation of task-dependent model abstraction, called AQUA [Sac01]. Struss presents the mathematical foundations and the implementation of the abstraction process and discussed the various difficulties and problems encountered in practical applications [Str02]. Yan [Yan03] deals with some of the problems discussed in [Str02] and proposes ways of automatic landmark generation for static as well as dynamic simulation systems.

The mathematical principle of generating finite relations from given numerical simulation or ordinary differential equations is as follows [Str02]: Given a numerical (simulation) model that computes one output variable y as a function of n input variables, x_i ; $y = f(x_1, \dots, x_n)$; suppose x_i has a set of landmarks $(l_{i,1}, l_{i,2}, l_{i,3}, \dots, l_{i,n})$. A qualitative value is defined as an interval bounded by a pair of adjacent landmarks; $q_{i,j} := [l_j, l_{j+1}]$. During abstraction, the input of x_i to the function is one of these intervals. The input into the function f is, thus, the tuple $(q_{1,k1}, q_{2,k2}, \dots, q_{i,ki}, \dots, q_{n,kn})$, which is a combinatory set of the qualitative values of the independent variables x_i . But, each input value represents an interval that possibly represents infinite real values between its two bounds. For the n input tuples, the combinations of the end points of the intervals alone results in 2^n possible inputs. If the function is monotonic within the 2^n -dimensional hyper-rectangle, one could evaluate the function at these corners and compute y as the interval bounded by the minimum and maximum of these outputs. This output interval is then translated into a (set of) qualitative value(s) in the abstracted model.

For non-monotonic functions, the landmarks have to be chosen in such a way, that they include the extrema of the function as well. However, determining a set of relevant landmarks from the input-output of numeric quantitative models is a complex task. If the function is differentiable, it is possible to compute landmarks „close enough“ to the extrema based on bounds of the derivatives.

This, however, requires an analysis of the details of the function which requires either user input or some automated approach.

A MATLAB™/Simulink™ model consists of a set of subsystems that have a number of hierarchically interconnected input and output values. These subsystems, at some appropriate level, will be the entities that are subject to the abstraction procedure. However, a MATLAB™/Simulink™ model is usually a dynamic **simulation** model that computes system behavior over time. In particular, it contains integration steps, and, hence, the value of some variables may refer to a later time point than the inputs to the system. To make sure that the qualitative constraints link only values that occur at the same time, any block that involves a delay (such as an integration block) should be eliminated from the respective subsystem, as shown in Figure 8. The elimination of such a delay block will have no effect on the final result of the abstraction [Str02]. The abstraction operation is then performed on the resulting (disconnected) subsystems. The abstracted relation of the entire Simulink™ model is then obtained as the join of the abstract relations of the sub-systems.

Struss outlines some practical obstacles to automated model abstraction, among which computational complexity and difficulty in determining initial landmarks are the major ones.

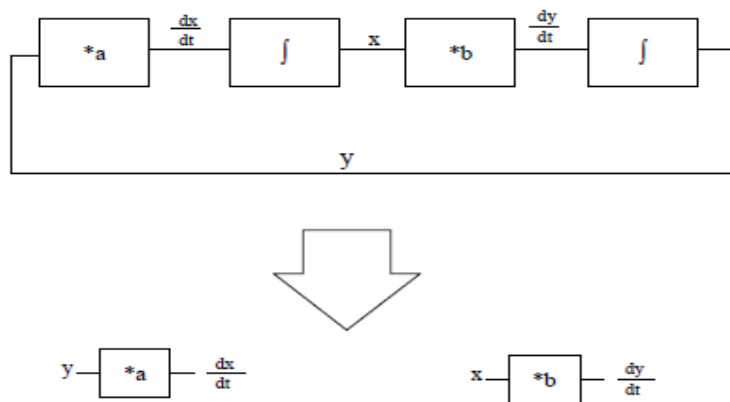


Figure 6 Elimination of an integration block

3.3.2 Generation of Deviation Model

Deviations refers to the difference between the actual value of a variable x and its reference value: $\Delta x = x - x_{\text{ref}}$. The interest in deviation is to know whether the value of a variable is equal, greater or

lower than expected. Qualitative deviation abstracts away the deviation to its sign only, thus its possible values are 0, - and +. Deviation models are used to simulate how deviations in input influences output variables and to describe behavior of a system when fault occurs.

For instance, the model of a valve is given by the constraint:

$$[\Delta q] = [A] * ([\Delta p_1] - [\Delta p_2]) + [\Delta A] * ([p_1] - [p_2]) - [\Delta A] * ([\Delta p_1] - [\Delta p_2])$$

on the signs of the deviations of pressure ($[\Delta p_i]$), flow ($[\Delta q]$), and area ($[\Delta A]$), where $[x]$ means $\text{sign}(x)$. This constraint allows, for instance, to infer that an increase in p_1 ($[\Delta p_1] = +$) will lead to an increase in the flow ($[\Delta q] = +$), if p_2 and the area remain unchanged ($[\Delta p_2] = 0$, $[\Delta A] = 0$) and the valve is not closed ($[A] = +$) [Str04].

The generation of a deviation model involves generating a qualitative model with given input and comparing with a reference model. Struss reviews the concepts and use of deviation models and related techniques and presents their general mathematical formalization [Str04]. Console et al. proposed a semi-automatic derivation of qualitative *deviation* models from MatlabTM/SimulinkTM simulation models [Con03].

Generation of deviation models is one of the important operations in modeling, but it is not implemented in the present work

4. MOM Specification

4.1 Introduction

The present work is based on the requirements that are documented in “Requirements Analysis of MOM” [MOMa], and in the specification of MOM static structure [MOMb]. The aims laid down in the requirements document include developing an environment and toolbox for generating and transforming models of physical systems. The specification document presented a systematic perspective of components required for the building and manipulation of models. These components were organized into classes that represent the model and classes that are means to generate and transform models (model operators) [MOMb]. As depicted in Figure 7, distinction is made between the generic part (abstract classes and interfaces) and classes that are specific to and representing specializations of particular modeling environments and formalisms.

A fundamental decision is to decouple the description of the (physical) system from the description of the respective behavior models. This is motivated by the goal to allow a 1:n relation between physical entities (components) and behavior models. In this way, the stable part of a system description should be represented only once and does not have to be duplicated or changed when the behavior model is modified. Since variables and their associated domains are decoupled, it should be possible to modify the granularity of the description as needed by the intended use of the framework without affecting the underlying model structure.

A layer of interfaces to external modeling environments is intended to serve as a link between MOM and other external modeling environments such as Simulink™, Modelica or CS2®. These classes shall be used for invocation of and communication with numerical or qualitative models in the external environments, for example, for qualitative model abstraction.

This chapter analyses some of the semantic groups shown as the conceptual classification of modules of MOM in Figure 7. The analysis helps to identify key object classes, attributes and methods needed for the software design. A graphical user interface (GUI) integrating the different operators of MOM is mandatory for a serious work, but this will not be covered by the present thesis.

Section 4.2 reiterates the functional requirements of MOM and describes some of the operators identified as major use cases in the requirement document. Section 4.3 analyzes the static structure

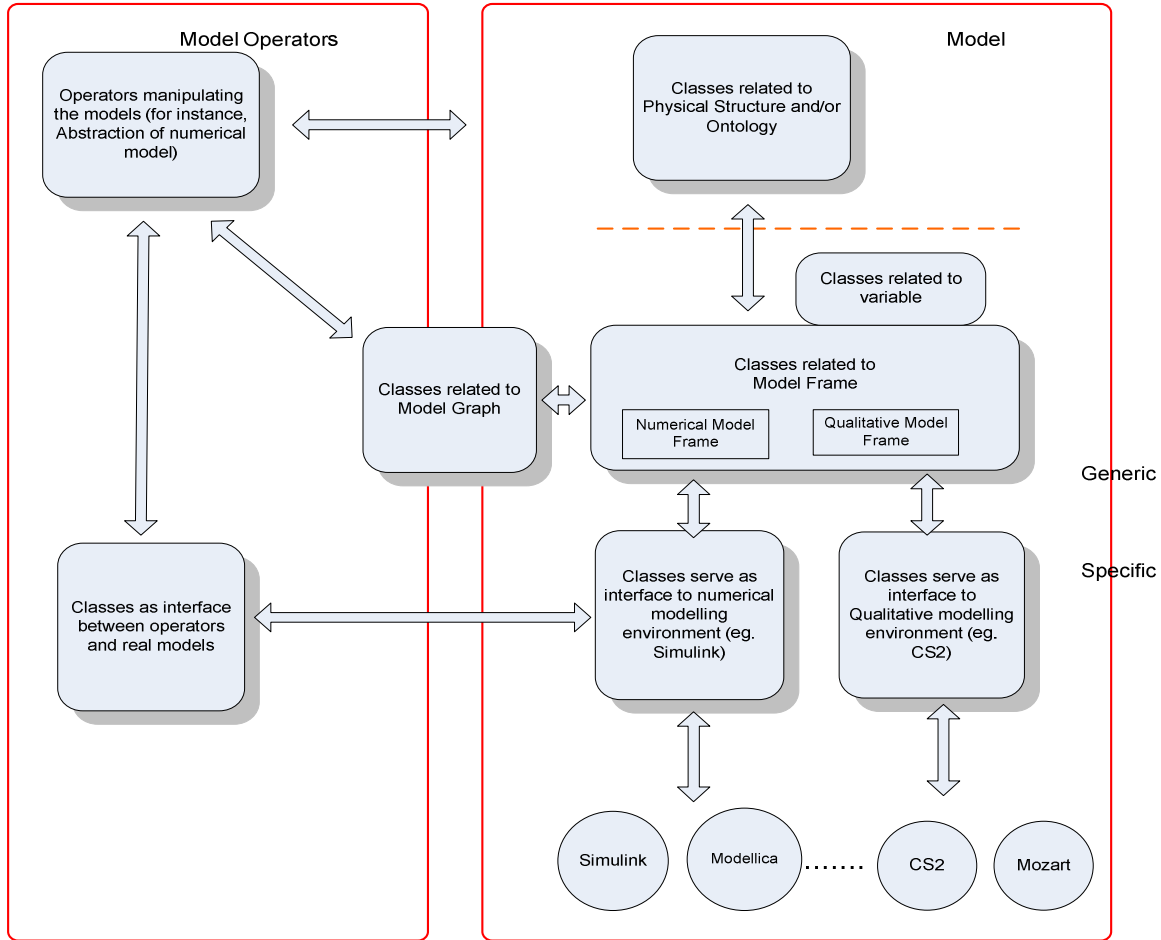


Figure 7 Conceptual organization of MOM modules.

of MOM and tries to identify elements that are the basis for other MOM modules. Section 4.4 deals with variables and related basic elements. Section 4.4 introduces Behavior Description and classes related to the description of entity behaviors. Section 4.5 then deals with Model Frame and Variable Mapping – group of classes that are used to relate different behavior descriptions to the physical structure of a system.

Section 4.6 presents MatlabBehaviorDescription which is an example of classes used as interfaces to external modeling environment. Section 4.7 deals with generic interfaces for operators in MOM. The last section presents some suggestion for a graphic user interface.

4.2 *Functional Requirements of MOM*

The goal of **Model Manipulation Environment (MOM)** is to offer a software tool that supports model building and model transformation in an integrated, modular, flexible, and extensible way. The tool should offer the user a development environment that allows manipulating models in a systematic way by re-using, adapting, or modifying pre-existing models. The toolbox should include a set of operations on numerical models, qualitative models, domain partitions, etc. [MOMa]. The key issue for such an environment is to provide a **generic interface** independent of specific modeling formalisms and systems that can be shared by various model operators. These abstract interfaces can then be used to integrate different existing tools as specializations. More specifically, the framework has to allow for a hierarchical structural representation of a physical system whose elements may have multiple models associated with them.

The functions of MOM should be realized with several related but independent operations. Figure shows the major top-level use cases, including:

1. **Composition of Qualitative Model:** compose or set up a model by modifying existing model pieces taken from libraries or instantiated from model classes.
2. **Qualitative Abstraction of Numerical Model:** produce (abstract) a qualitative model from a numerical one. In principle, the numerical model could be developed in any mathematical environment.
3. **Compression of Qualitative Model:** convert a given qualitative model into a “smaller” model by projecting out some variables that are deemed to be irrelevant. This operation is in effect a manipulation or transformation of qualitative models.

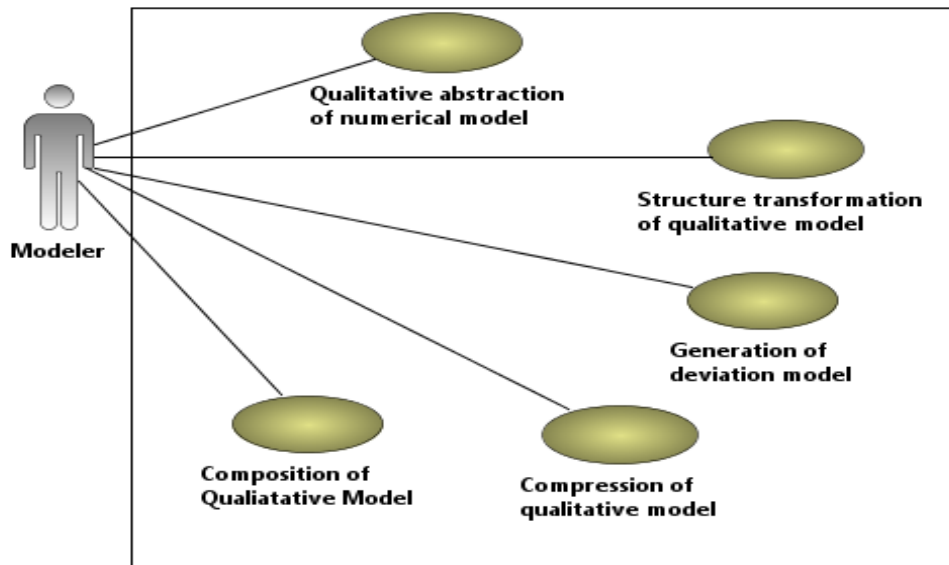


Figure 8 Top-level Use Cases in MOM.

4. **Structure Transformation of Numerical Model:** rearranges the internal structure of a numerical model (such as a Matlab™/Simulink™ block) in order to simplify subsequent operations on them (such as qualitative abstraction).
5. **Generation of Deviation Model:** generate a deviation model from a qualitative or a numerical one. It involves generating a qualitative model with given input and comparing it with a reference model.

In the following section, only the first two will be further analyzed and refined (as the main theme of the current thesis). Short description of the use cases and basic concepts will be presented.

4.2.1 *Composition of Qualitative Model*

Model composition involves putting model pieces together to create a model of desired complexity. The user must be in a position to compose a new model starting with the ones retrieved from a library or a repository. Such operations include editing of components, addition or deletion of connections, checking the integrity of the composition, and packaging, unpacking or deletion of components.

MOM requires a library (a directory in the file system) in which standardized, reusable templates of structural entities, terminals, domains, *model frames* and others could be archived. The library should contain templates for standard elementary entities commonly found in practice. Also, the user must be in a position to construct and save as templates more complex structures (s)he often

uses. A repository, on the other hand, shall be used as a storage for structural entities being manipulated.

Figure 9 depicts relevant use cases of constructing a model structure, which is the use case given in the requirement document with a few modifications. The actions in model composition involve two separate undertakings. The first is the work of one who is responsible for the creation of elementary model blocks necessary for the library, a library administrator so to say. His/her main activities are creating new entities (that include the use cases shown in Figure 10), and associating them with alternative and consistent sets of internal and terminal quantities. Such a consistent list of quantities is termed *QuantityAssociation* (see section 4.4).

The model-builder, on the other hand, picks these elementary building blocks from the library and constructs the complex and possibly hierarchical ordered model of a system out of them. In doing so, he has to take decisions as to which constellation of quantities fits his demands. The activities of model building is conceptually depicted in short in Figure 11. Other decision are also required at other points later in the process.

The following uses cases are in part common to both tasks. The descriptions may not reflect the subtleties of two goals.

Compose new Structure: First level use case representing the operator Model Composition. It opens an empty base structure, which will be built through subsequent addition and modification of sub-structures.

Edit Existing Structure: This loads an existing model structure from the library or repository to be edited. Editing includes modifying the structure, packaging into a new *StructuralEntity* and saving the modified structure.

Modify Sub-structure: A generalization of operations that change the internal structure of a model. It includes adding or deleting a sub-structure, adding new connection or removing existing ones, decomposing the structure into its constituents, and modifying individual sub-structures.

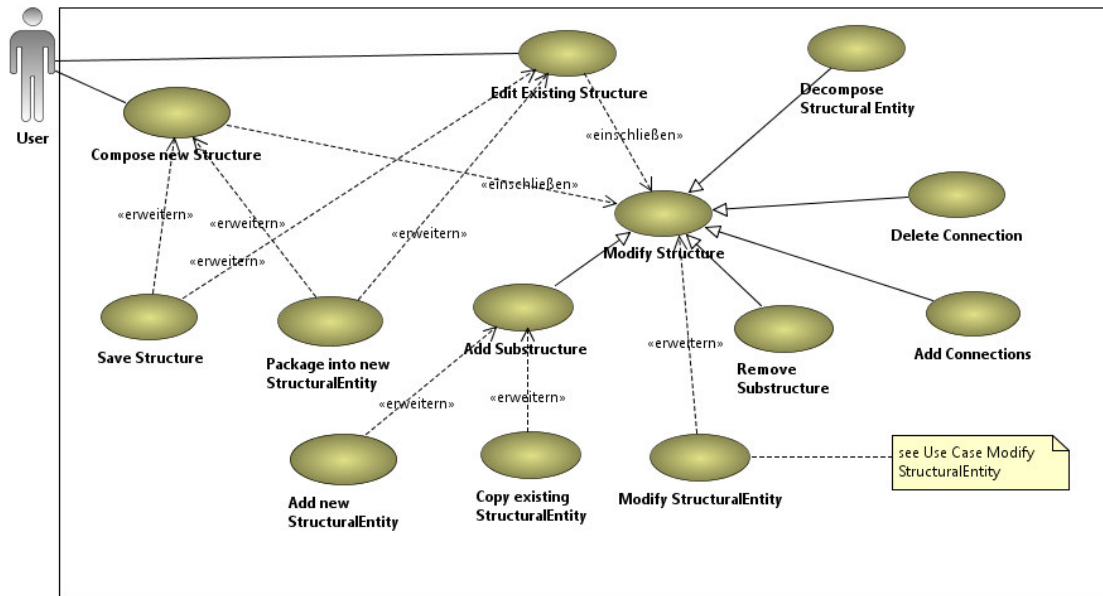


Figure 9 Relevant Use Cases in Model Composition

Add Substructure: Adds a sub-structure to the structure in the current working space. The sub-structure to be added could be a StructuralEntity newly instantiated from the entity class or the template library or a copy of an existing StructuralEntity.

Delete Substructure: Removes a selected sub-structure and optionally removes all the connections attached to it.

Package Substructure into a new Structural Entity: Creates a new structural entity out of sub-entities of a selected sub-structure and adds it to the base structure. All selected structural entities will be moved into the new one, including the connections between them. Existing connections will be replaced by coherent connections to the new entity.

Add Connection: Connects two or more structural entities by connecting selected terminals of the concerned entities.

Delete Connection: Deletes a selected connection node, thus, removing all connection at the node or disconnects a selected terminal from the node.

Decompose Structural Entity: Decompose the selected structural entity into its constituent entities. After decomposition, the structural entity ceases to exist as one entity, but its internal structure remains unchanged.

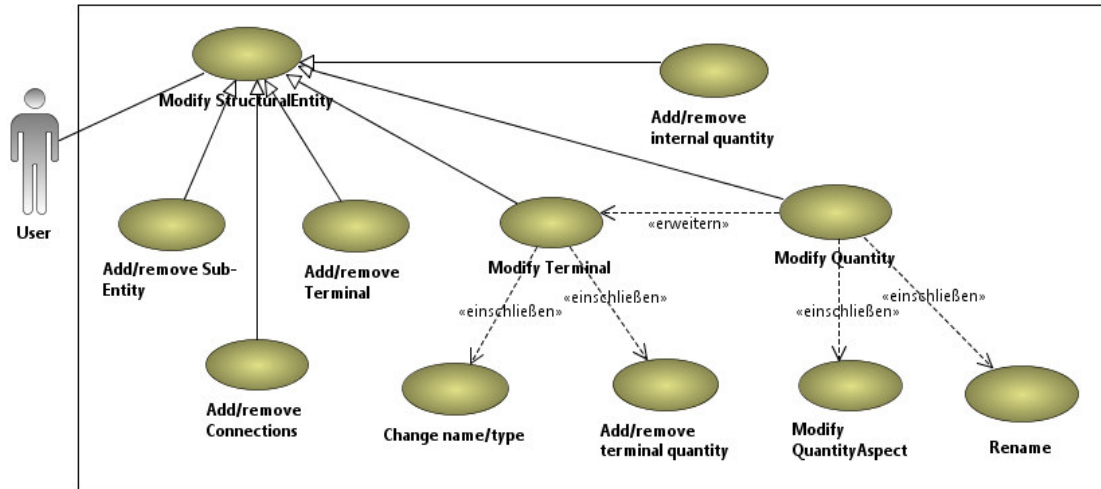


Figure 10 Use Case - Modify StructuralEntity

Modify StructuralEntity: A generalization for a number of modification that could be applied to a StructuralEntity. As shown in Figure 10, these include:

Adding or removing a sub-entity, a terminal, an internal quantity or a connection, as well as modifying a terminal, an internal quantity or a terminal quantity. Modification of a terminal includes changing its name or type. Quantity modification includes renaming or changing the quantity category.

Save Structure: Stores this structure as a structure or a StructuralEntity in the repository, or as a structural entity template in the library.

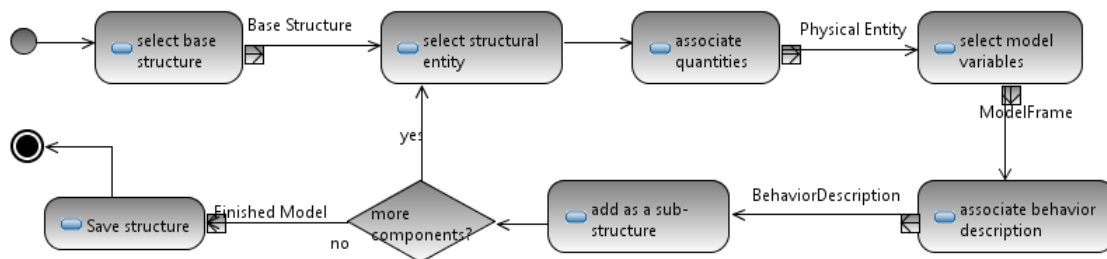


Figure 11 Activities of a model builder

4.2.2 *Qualitative Abstraction of Numerical Models*

As described in the previous chapter, the main goal of this operation is to derive a qualitative model from a numerical one. The numerical model could be one realized in any mathematical environment (Matlab™/Simulink™, Modelica or others). Several sub-operations are involved in this process, which include associating domains with the variables, generating landmarks automatically or semi-automatically, and the possibility to inspect and edit the result. Main actions in this operation are shown in figure below [MOMa].

Using a simplified description, steps involved in qualitative abstraction of a numerical model, typically from a Matlab™/Simulink™ source, include:

1. retrieve (open) the numerical model with the original application.
2. if the application/model allows, select part of the model (a block in Simulink™) to be abstracted. This could also include some manipulation or transformation of the original model structure, such as making a copy of the selected part and classifying the involved variables.
3. get list of the variables involved in the source model, their data types and their domains. This depends on whether the source application provides such a means.
4. create equivalent (i.e. corresponding) variables in the target model. The list of variable deemed relevant in the target model may not include all of the source variables, but ignores those that do not influence the functionality of the target model or that are irrelevant to the current task.
5. set data types and domains for the variables in the target model. This could be effected in the form of a list of intervals or landmarks. The essence of abstraction is to obtain the relational model among the variables in the form of qualitative values. The decision about intervals and landmarks depends on "*the distinctions in the domains of the system variables that are both necessary and sufficient to achieve a particular goal in a certain context and under given conditions?*" [Str02]. The necessary landmarks (or intervals) may be known from the outset (user decision) or they must be generated automatically.
6. map the two sets of variables and their respective domains. Variable mapping is a lookup tabulation of which target variable corresponds to which source variable. Domain mapping relates domain values of a source variable to domain values of the target variable.

7. compute values of the variables in the source model covering all their possible combination and value ranges. Project (translate) the output from the source model to corresponding qualitative values of the mapped variables in the target model. A full tuple of values of the source variables would correspond to a qualitative tuple in the target model.

For an abstraction process, one needs a module that serves as an interface between the application with which the numerical model is developed and the abstraction operator in MOM. Such an interface module wraps the particulars of the external mathematical environment for MOM. Such an interface for Matlab™/Simulink™ (MatlabBehaviorDescription) has been implemented in previous works [Fraracci08]. The present work improves and expands this existing work to include aspects of multi-modeling.

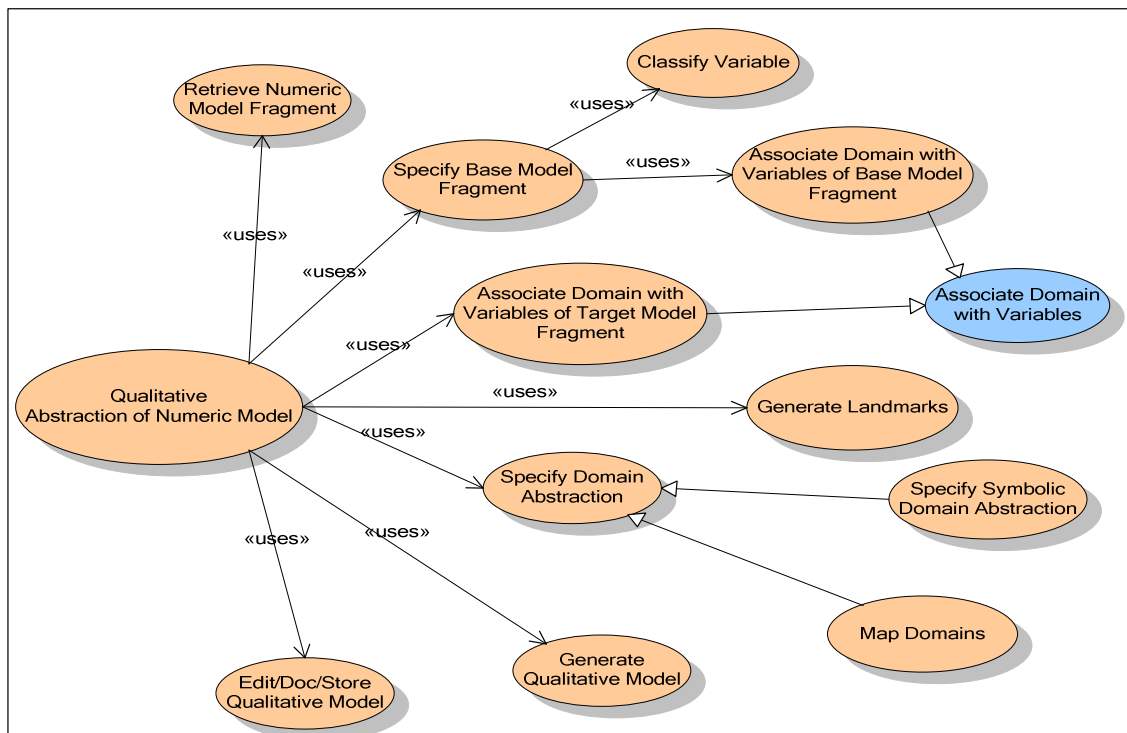


Figure 12 Use Cases in Qualitative Abstraction of Numerical Models

The mathematical background of qualitative abstraction of a numerical model is described in section 3.3.1. Some of the classes needed for qualitative abstraction are also common to other operations, so they would be identified and described together in section 4.3.

4.3 System Architecture

The MOM is currently designed as a desktop software. As stated before, a library is required as a storage for standardized, reusable templates of structural entities, terminals, domains, model frames and others. The library will be used as a storage for standardized and context-free modeling elements that can be instantiated and used in different projects. The user must also be in a position to construct and save as templates structural elements (s)he often uses. A repository is a sort working directory for the modeler, where models under construction and other temporary elements may be saved to. Each modeling project may require its own directory. Figure 13 depicts the system environment.

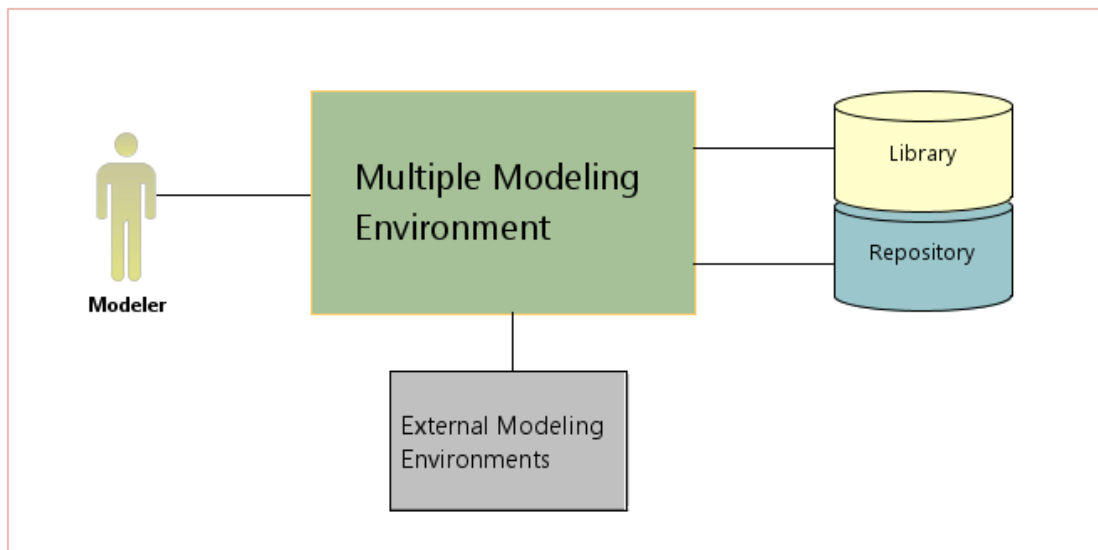


Figure 13 System environment

4.4 The Core Classes and Architecture of MOM

Figure 14 below shows schematically the idea of decision-based model building, consisting four layers. The concept rests upon the assumption that the transition from one layer to the next requires some sort of user input. Precondition for this mode of operation is that context-free and basic elementary models are made available in a library. The work of the model builder then consists of fitting suitable elements together and tweaking alternative decisions at each transition to create the desired modeling unit. A number of such units are then connected according to the physical/logical part-of relation and hierarchy representing the complexity of the task at hand. The concept is described in detail in the next sections.

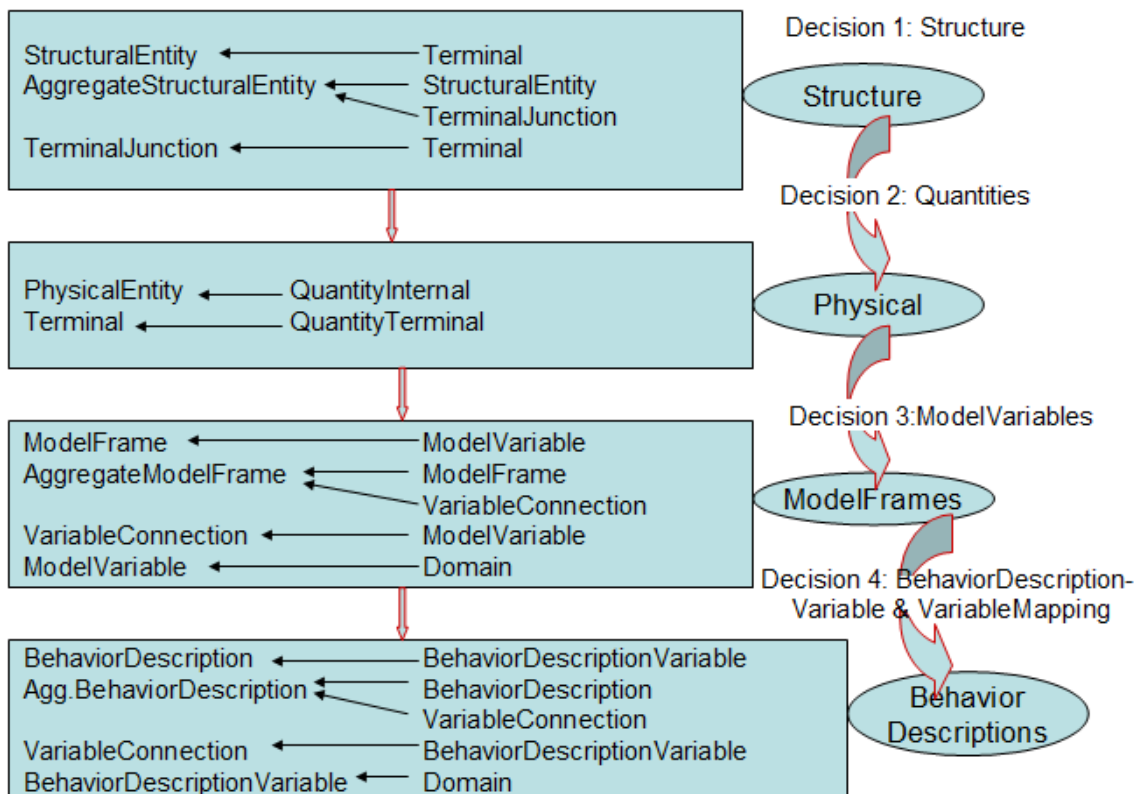


Figure 14 Decision-based Model-building Steps

4.4.1 Describing the Structure of a System

This section enumerates the classes related to the structure of a system. As is introduced in section 3.1, **StructuralEntity** objects are the building blocks of the physical structure of a system. A *StructuralEntity* can be elemental or an aggregated one. *StructuralEntity* is, thus, a generalization of *ElementaryStructuralEntity* and *AggregateStructuralEntity*.

ElementaryStructuralEntity is a specialization of *StructuralEntity* and represents an atomic entity that can/need not be further decomposed into smaller units.

An **AggregateStructuralEntity** is a sub-class of *StructuralEntity* that contains other structural entities connected through their terminals in a composite pattern. It has one or more *TerminalJunction* objects.

As stated before, a *StructuralEntity* (excepting *Structure*) has at least one *Terminal*, i.e. a point of contact with its surroundings. Usually, a *Terminal* is a composite part of *ElementaryStructural-*

Entity. For an *AggregateStructuralEntity*, all free Terminals of its constituents are available for its interaction with its surroundings. A *StructuralEntity* can be associated to a *StructuralEntity-Template*, which is a blueprint from which the structural entity can be derived.

A **Structure** can be seen as an aggregate structural entity containing a complete set of entities and represents a certain system, which is not intended to be used as a building block for a larger system.

Aggregation of structural entities involves addition, removal, and modification of sub-entities, terminals and quantities, as well as connecting and/or disconnecting terminals of sub-entities. These are methods that must be provided by the *StructuralEntity* class in addition to those actions listed as use cases under model composition.

The UML diagram in Figure 15 illustrates the interrelations between structural entity and its terminals.

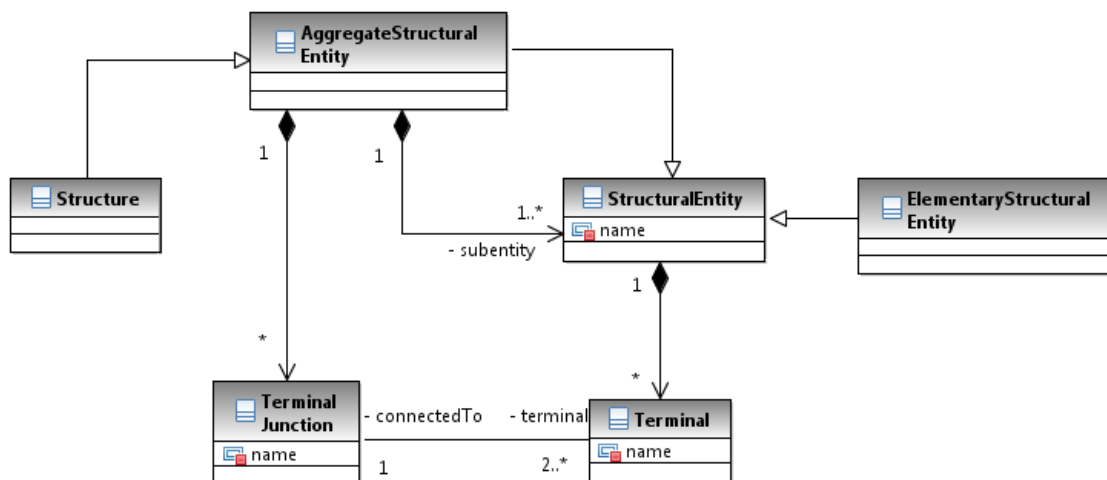


Figure 15: A UML diagram of a structural entity and its elements

As stated in section 3.1.2, a **Terminal** is a real or conceptual interface of a *StructuralEntity* to its surroundings. A *Terminal* has a **TerminalType**, an attribute that indicates the physical compatibility between terminals (such as electrical, hydraulic, etc.). Two terminals must have compatible *TerminalType* for the connection to be possible and meaningful.

TerminalJunction is a node where two or more terminals of differing sub-entities meet. Controlling consistency and coherence of connections can not be adequately regulated if terminals are directly connected to one another. Instead, a *TerminalJunction* is put in place wherever two or

more terminals are to be joined. Two TerminalJunctions can not be connected directly. Instead, at least one StructuralEntity has to be placed between them.

Connection and disconnection of terminals, as well as checking the compatibility and coherence of connections are the methods associated with TerminalJunction. Coupling rules (junction constraints) must be applied to the connections taking place at a node.

A TerminalJunction should have a name as an identification of its role.

These classes allow describing a “flat” structure of a system by connecting StructuralEntity objects via TerminalJunction objects that link their Terminals as well as a hierarchical decomposition by turning such a structure into a StructuralEntity itself.

The unconnected (free) terminals of the sub-entities in an AggregateStructuralEntity serve as terminals of the aggregate. This in turn could be connected to other aggregate entities, so that a highly hierarchical and complex entity can be built.

4.4.2 Associating Quantities

A **Quantity** represents an attribute or a state of a structural entity. It may be associated with a *Terminal (QuantityTerminal)* or it represents an internal state of the entity (*QuantityInternal*). Quantities can be categorized, e.g. according to the type of physical properties or kind of information they represent, such as pressure, force, energy, etc. or command, set point etc., respectively. Such a **category** is essential to identify corresponding quantities at a TerminalJunction. As is discussed in chapter 3, a physical quantity can, for instance, be a flow or an effort type. As is used in most physical system, one can also associate a **unit of measure** (restricted by the category) with a quantity. However, since variables representing different aspects of a Quantity are used for describing behavior of a structure, so that associating a unit with a Quantity would be directly relevant only for certain variables.

Thus, a Quantity object has a name and a category that represents its role.

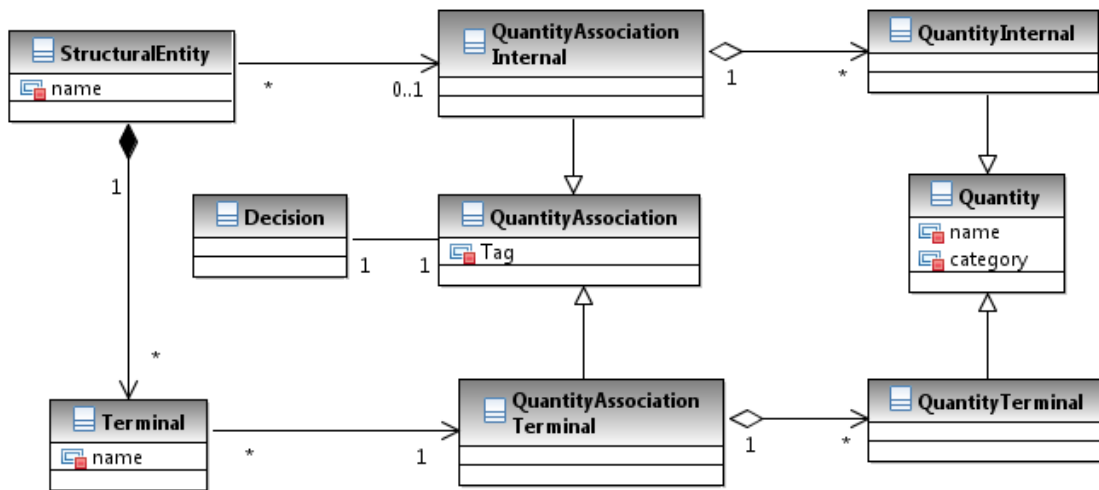


Figure 16 Associating quantity to a physical structure

4.4.3 Decision-based Associations

Decision-based association is concerned with incorporating alternatives, taking decisions at the different levels of the model-building process, and represent them explicitly as shown in Figure 14.

The initial step taken during the model-building process is selecting a base structure from the library. This could be a basic (elementary) structural entity or some composite structure acting as a unit. The next step involves making decisions about the quantities, that represent states and parameters of the base structure. It is concerned with determining which quantities are relevant for the task at hand. To model a valve either as a continuous resistor or as a discrete on/off switch is, at the end, deciding which quantities to consider and how. The decision at this level then determines available choices at lower levels.

The selected base structure is presumed to be associated with a number of alternative quantity constellations. Such a constellation of quantities is called QuantityAssociation, which is a list of quantities that can/must be associated with the structural entity under one particular modeling scenario. One such QuantityAssociation has one set of internal quantities and associated with the set a number of sets of terminal quantities that are consistent with these internal quantities.

Such decisions are made individually for every entity unit picked from the library to build the complex scenario.

The second decision involves model variables. For a particular task, particular aspects of the quantities are needed. Thus, by associating QuantityAspects with quantities selected in the first step, one gets a set of ModelVariable objects that will be used to describe the functionality/behaviors of the entity in the particular scenario. At this stage, however, no particular decision has been taken concerning the domains of the variable (e.g. whether numerical equations are to be used or qualitative descriptions are desired). Thus, number of alternative behavior descriptions can possibly be associated with the model frame so created.

The particular way of describing functionality/behavior is thus the object of the next decision. A number of behavior descriptions may exist in generic and context-free forms, or could be adapted from similar works before, or abstracted from other models. A necessary step is then to adopt the relation for the current situation. This involves mapping the variables in the existing relation to the set of ModelVariable objects selected for the model frame. The user of the library can compose the model by referring to decisions, without necessarily knowing the details of the associations.

4.4.4 *Physical Entity*

As is stated earlier, the first step in decision-based model-building is selecting a base structure from the library. Each basic structural entity has been associated with a number of alternative quantity associations. The structural entity thus selected is assigned the desired type and combination of quantities (internal and interface qty) by selecting one from the alternative QuantityAssociations. A QuantityAssociation is, a constellation of quantities that are (must be) associated with a structural entity under one particular modeling scenario. One such QuantityAssociation has one set of internal quantities and a number of sets of terminal quantities that go (are consistent) with these internal quantities. A model builder can then choose one of these alternative constellations that suit his particular needs. The model unit so created corresponds to a physical entity.

The decision made at this juncture will influence other constellations later in the model building process.

4.4.5 *Choosing Variables and Domains and Creating Model Frames*

Variables are central to modeling, therefore, their proper treatment is key to successful model building and model operation. This section discusses classes that are related to variables, including variable domains and values.

A **Variable** is a name or symbolic representation of a value that may change. For physical systems, a variable is often a real-valued representation of a selected attribute. Variables characterize the system or its constituents and play a major role in model manipulation. A variable has a *Domain*.

A **Domain** is a set that contains values a *Variable* can have. Implicitly, a domain encompasses a range of values of a certain type. Such values can be broadly divided into numeric or non-numeric (i.e. symbolic) types, and they may be discrete or continuous over a bounded range. The extent (size) of the set may be finite or infinite.

As is stated before, quantities represent the state and attributes of the entity without specifics about value and value domains. A variable, on the other hand, represents certain aspects of a Quantity, e.g. its magnitude, its derivative, its deviation from a reference value etc. Hence, variables are associated to Quantities as introduced in section 4.3.2 by QuantityAspects.

A given physical entity may be modeled in different ways. Repeating the previous example, a fluid valve can be modeled either as a discrete, on/off switch or as resistance that can vary continuously. Such alternative perspectives can be taken care of, on the one hand, by associating different quantities with the same structural entity in different modeling scenarios (Quantity Decision). On the other hand, different quantity aspects can be considered in different scenarios involving the same quantity. This means, one can define a different variable by associating a different QuantityAspect with the same Quantity, and even for the same combination of Quantity and QuantityAspect (and thus same Variable), one can define and associate various domains with the Variable (and use to define a different behavior).

A set of variables resulting from such choices spans a certain space for modeling the behavior of a Physical Entity. We represent it as a ModelFrame.

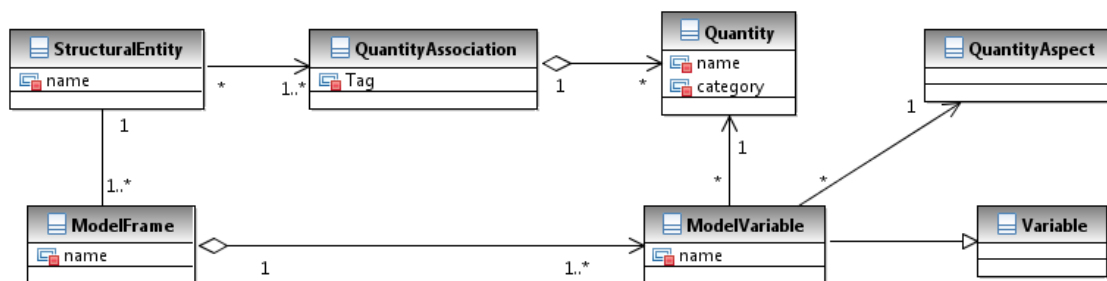


Figure 17 ModelFrame and Modelvariable

4.4.6 *ModelFrame*

A **ModelFrame** is a set up for a particular modeling operation. It could be defined as a definite combination of behavior descriptions that would be adequate to simulate the full behavior of the entity or its interaction with its surroundings under a given context. It relates a StructuralEntity under certain mode of operation and the associated *ModelVariable* objects to a given set of behavior descriptions.

Accordingly, a ModelFrame does not fix a specific description of this behavior, but represents a “container” for such (possibly different) descriptions and is a means for structuring it. Hence, it can be more fine-grained compared to the Physical Entity. The ModelVariable objects of a ModelFrame are very generic and must not presuppose certain specific properties of different ways to describe the behavior in a particular mathematical or computational form. In particular, their domains have to be chosen as generic ones, which can then be mapped to specific representations. For instance, the domain of a ModelVariable Resistance should be specified as R_0^+ , i.e. the non-negative numbers, which, in an implementation, allows for using double, positive double, {0, +} etc.

Also ModelFrames can be structured in a hierarchical way. For instance, a Physical Entity of an electrical resistor has the quantities resistance R as internal quantity, and current & voltage as terminal quantities on each of the two terminals. It could have two associated ModelFrames representing Ohm's law (with the variables V, I, R) and Kirchhoff 's current law (with variables i1, i2).

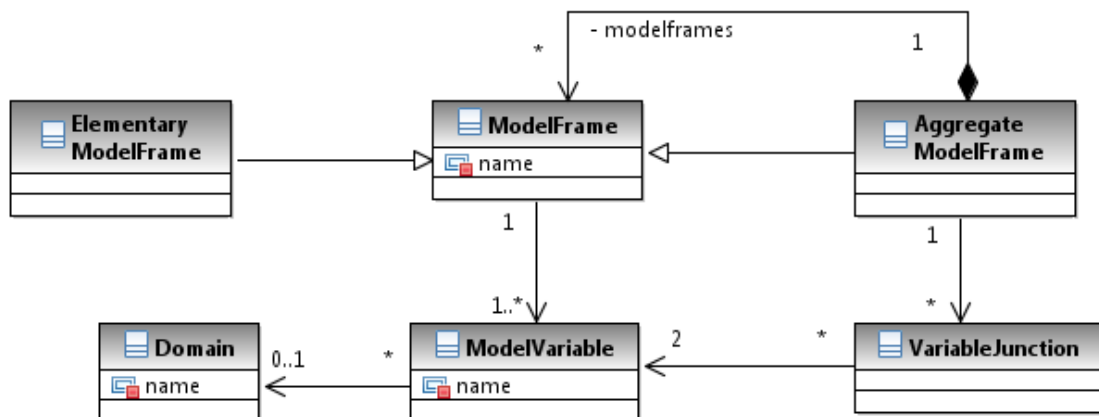


Figure 18 Aggregate ModelFrame

4.4.7 *Associating Behavior Descriptions*

Finally, an actual, implemented BehaviorDescription has to be associated with a ModelFrame. A *BehaviorDescription* is a mathematical piece that describes a portion of the behavior of an entity. It represents a concrete technical realization of the mathematical behavior/functionality in a specific system. As is sketched in section 3.1.5, a *BehaviorDescription* could be some *quantitative* (numerical) model in the form of differential equations or inequalities, or a relational model in terms of a finite set of qualitative values, or some other abstract characterization.

This could be a Matlab™/Simulink™ block, a SPICE model, a constraint over finite domains, etc. Hence, also this association is based on modeling decisions (and, perhaps, the most important branching in the space of decisions). It has to include a description of how the ModelVariables of the ModelFrame are mapped onto the variables used in the BehaviorDescription (with possibly totally different names) and how their domains are related.

VariableMapping and the associated DomainMapping and ValueMapping are used to bind variables in different behavior descriptions related in a ModelFrame. This enables the conversion between values of variables in separate behavior descriptions of an entity.

One can define *VariableMapping* as the correspondence between a pair of variables that represent equivalent values in two different behavior descriptions of the same entity. It is a look up table of variables. Associated with a given VariableMapping is *DomainMapping* which maps the domains of the variables, if both variables possess any. A DomainMapping is in turn associated with a set of *ValueMappings* which map values in one domain to equivalent values in the other domain. Value mapping gives the relation, i.e. equivalence between the values of a variable having one domain to the values of the mapped variable with another domains.

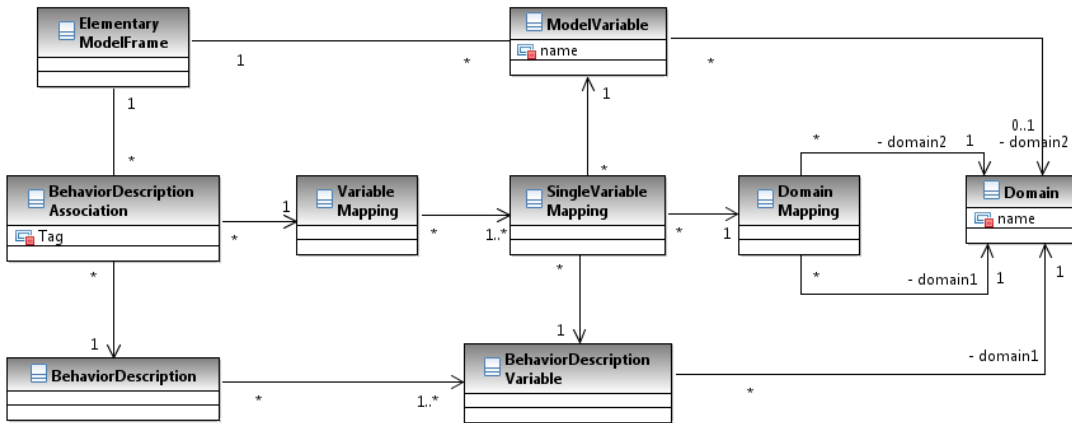


Figure 19 BehaviorDescriptionAssociation

4.4.8 BehaviorDescription

A behavior description for a particular structural entity is then a relation between the different variables representing the different aspects of the quantities associated with the entity and its terminals. Such a relation may involve only a subset of the quantities. Different relations may describe different facets of the behavior of the structural entity.

Also BehaviorDescriptions can have a hierarchical structure and possibly be more fine-grained than the existing ModelFrames.

As a result, we obtain the following SSD:

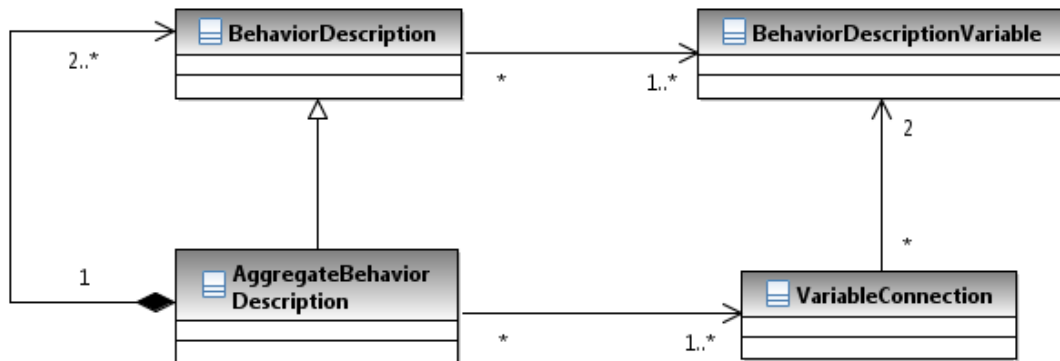


Figure 20 AggregateBehaviorDescription

MOM contains interfaces to various **classes** of BehaviorDescription, which allows the builder of models and model-based systems to use this interface regardless of what kind of computational system is used. For instance, a relational BehaviorDescription over a finite domain could be implemented by various constraint satisfaction systems, OMDDs, a theorem prover, etc..

The main issue here is to provide a generic definition for an interface that can be a common denominator for the different formalisms being used to represent a behavior of an entity. Mathematical formulations (simple or differential equations and inequalities) usually express some dependent variable in terms of a set of other (independent) variables or values. Directionality, i.e. the input-output causality is fixed. In others, such as in tabular representation of relations, such causality is absent. Matlab™/Simulink™ models are directed, whereas Modelica follows an acausal modeling principle.

Behavior descriptions are usually general context-free formulations without a connotation to a particular structural entity or mode of operation. This is also demanded for the purpose of re-usability. As a context-free formulation, a *BehaviorDescription* can be associated to several model frames through different abstraction processes or associations.

Therefore, BehaviorDescriptions could be classified depending on whether the relation is *computational* or *symbolic*, and whether the causality is *directed* or *undirected*. Operations in connection with BehaviorDescription include evaluation of some output for a given set of input, addition or elimination of a variable, etc. In case of tabular relations, operations may include addition or removal of a tuple of such a relation.

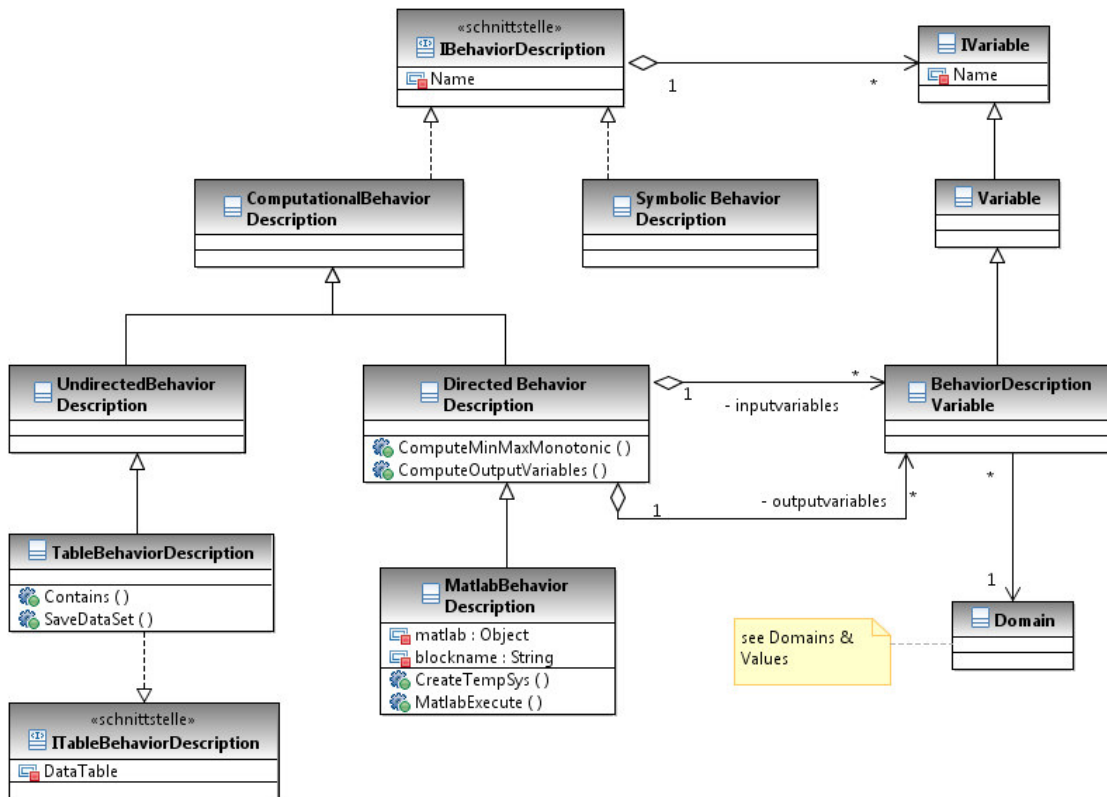


Figure 21: Classes of Behavior Description

As is stated in the previous section, BehaviorDescriptions are conceived as context-free formulations of entity behavior. A **mapping** is thus required between the variables in the behavior description (*BehaviorDescriptionVariable*) and the variables characterizing the quantities of a structural entity (through quantity aspects).

A given structural entity may have many behaviors and, thus, a number of behavior descriptions may be associated with it. Even for an elementary structural entity, a number of behavior descriptions may be needed to describe its various functionalities. Each one of the behavior descriptions may be presented using different formalisms. All of these would then be compounded for an aggregate structural entity.

A given physical entity may be modeled in different ways. Repeating the previous example, a fluid valve can be modeled either as a discrete, on/off switch or as resistance that can vary continuously. Each of these modes are related to different behavior descriptions.

A **ModelFrame** is a set up for a particular modeling operation. It could be defined as a definite combination of behavior descriptions that would be adequate to simulate the full behavior of the

entity or its interaction with its surroundings under a given context. It relates a *StructuralEntity* under certain mode of operation and the associated *ModelVariables* to a given set of behavior descriptions.

4.4.9 Special Domains Built-in

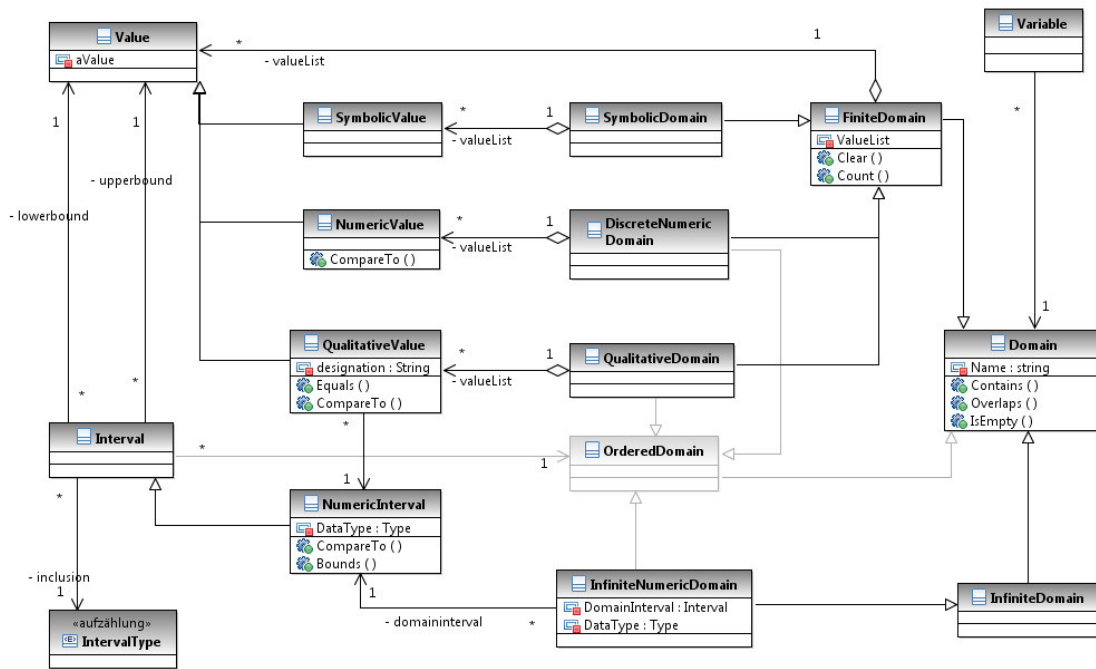


Figure 22: Conceptual class diagram showing the interrelationships between Variables, Domains and Values

A **Value** is actually a single member of a domain, so it could be a symbolic one (an object), a number, or a qualifying designation associated with a background numeric interval. More importantly, seen from the perspective of OOP, a value is a single instance of a certain object. *symbolic values* are, by definition, objects of some form. A *discrete numeric value* represents any discrete number. A *qualitative value* represents a designation for some *numeric interval* that is delimited by two bounds (*landmarks*).

An **Interval** is a general designation of a set of (ordered) values between two boundaries. The boundary itself may or may not belong to the interval. An *Interval* has an *inclusion type*, which indicates whether the interval includes one or both or none of its boundaries. A **Landmark** usually marks the position where a significant (task relevant) change happens in the values of a variable. It

is, thus, used as a boundary between neighboring *Intervals*. A *Landmark*, belongs to either the lower interval or to the upper interval, or to none of them.

A *Value* is, thus, a generalization of values that can be assigned to a *Variable*. As discussed above, *DiscreteNumericValue*, *QualitativeValue* and *SymbolicValue* are specialization of *Value*.

The figure above shows also the interrelationships between Values.

Usually non-numeric (symbolic) types refer to a set of object representations, such as a list of elements, etc., thus are inherently finite. Numeric types may too be finite (such as a fixed array of numbers), or may be, for example, any rational number, and can only be expressed as such. Numeric types may further be divided into integral types and floating point types as in most programming languages (short, long, single, double, etc.).

Depending on the type of the values and their range, domains can be finite (**FiniteDomain**) or infinite (**InfiniteDomain**). As stated above, symbolic values are inherently finite, so *InfiniteDomain* refers normally to numeric value types (**InfiniteNumericDomain**). *FiniteDomain* may contain a finite (discrete) set of numerical values (**DiscreteNumericDomain**), or may simply represent a list of objects without any numerical connotation (**SymbolicDomain**), or it may represent qualified references to an interval of numerical values (**QualitativeDomain**).

From a different point of view, the values in a domain may have an intrinsic total order (as do rational numbers, for example) or not. (**OrderedDomain**) Figure 9 shows the interrelation of value and domain classes.

A **ValueSet** is a special collection (set) containing any value type. It is especially defined for the purpose of exchange of values during domain and value mappings. The concepts of variable, domain and value mappings will be described in conjunction with *behavior description* and *model frames* in the next section. Details concerning ValueSet will be presented in the design stage.

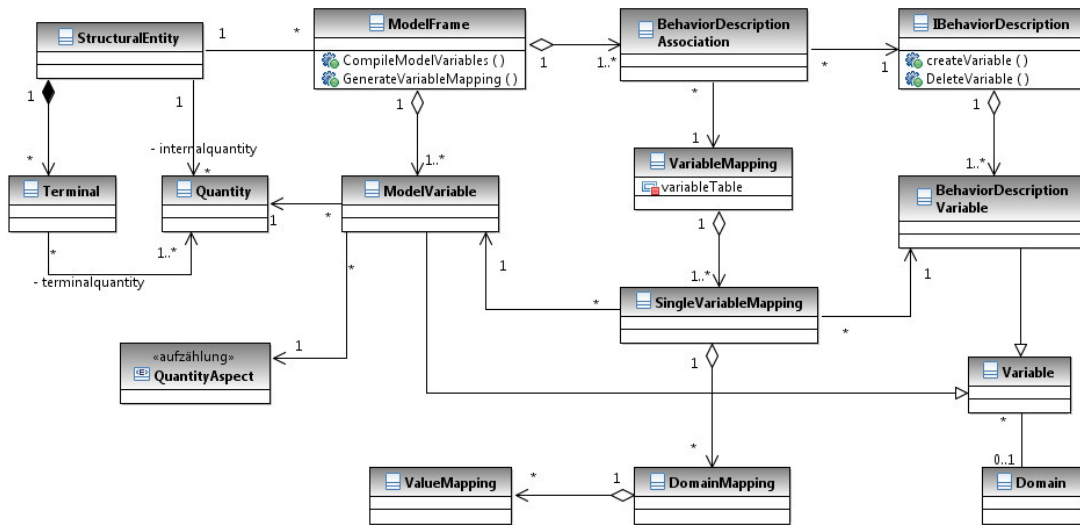


Figure 23: Conceptual class diagram showing the interconnection between StructuralEntity and BehaviorDescription

4.5 MatlabBehaviorDescription

The abstract interfaces of the classes of BehaviorDescriptions have to be implemented for the various modeling systems to be used for the actual computation. Possible classification of these interfaces is as is shown in Figure 21.

MatlabBehaviorDescription is meant as interface mainly to send commands to Matlab™/Simulink™ workspace to modify or run a Simulink™ model and get output of the simulation, and to extract model parameters such as list of sub-blocks, variables, variable data types, etc. Modification of a Simulink™ model includes copying, adding or removing a block, and adding, removing or changing block parameters. Such operations must actually be performed on a copy of the model/block to avoid loss of original model.

The current work examined an existing implementation, and added improvements and necessary changes to adapt it to the needs of MOM.

4.6 MOM Operators

MOM functions should be realized as a number of related but independent operations necessary for the transformation and manipulation of models. Some important operators envisaged to be include are:

- variable elimination in numerical and qualitative models (projection);
- automated generation of significant distinctions (“landmarks” or interval boundaries) from a numerical model;
- automated computation of a qualitative model from a numerical one given a set of significant distinctions;
- automated abstraction of a qualitative model based on a set of significant distinctions induced by a particular task.

Among these, the current work includes the implementation of qualitative abstraction of a numerical model. Other operations may be added later as the need arises.

4.7 A Suggestion for Application GUI

The modeling environment is going to be an editing system. A generic architecture of an editing system consists of a set of interacting objects, which are rather active, and can operate concurrently and autonomously. Essentially screen events are processed and interpreted as commands. This updates a data structure which is then redisplayed on the screen.

In addition to the above, MOM modeling environment shall have a workspace with a main window that graphically displays the interconnection of the Structural entities and their constituent elements.

The project use cases should be categorized into toolbars, menus and context menus. Each of these may be connected to one or more dialogs, which could be arranged hierarchically.

5. Design and Implementation

This chapter presents an overview of the design and implementation of the software components specified in Chapter 4. It discusses key points of the software design and implementation, and outlines which of the generic interfaces and basic software components have been designed and implemented. The software components implemented within the framework of this thesis include classes related to the static structure, variables, domains and values. Implemented are also generic interfaces for behavior description and the service classes for variable and domain mapping. The *NumericDomainAbstractionOperatorDirected* that specializes the *AbstractionOperator* has been implemented and test runs has been conducted with the abstraction of qualitative relation from a Matlab/Simulink model. A GUI was developed for the test run, which could be integrated with future fulfilled graphical user interface for MOM.

5.1 System Architecture

The MOM is currently designed as a desktop software. For the implementation, C# is to be used as the programming language. For operation with external modeling environment, such as Matlab or Spice, local installation of such application is required. Future implementations may consider other (eg. client-server) architectures for interaction with the external environments.

5.2 Software Packages

For the purpose of overview the software modules in MOM could be packaged as shown in Figure 24. Not included in the packages is obviously a GUI, which is indispensable for a development environment like MOM. It is left out for future work.

The following sections discusses which of software classes in each package have been implemented and how.

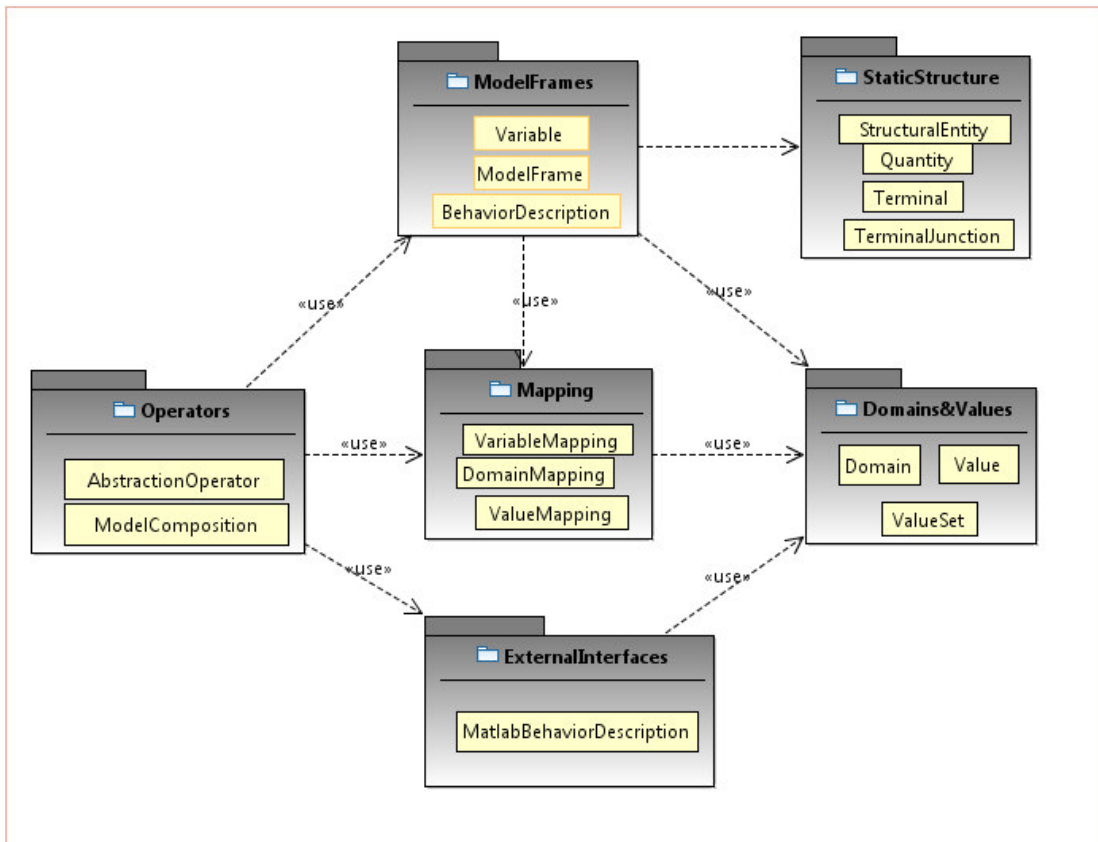


Figure 24 System Packages

5.3 Package Static Structure

The design and implementation of the classes in this package is basically done from the perspective of creating elementary structural entities as would be done for the library. Some of the implemented methods could be used in the model construction phase, such as connection and disconnection of terminals. The design and implementations in this thesis do not include the model-building aspect.

Included in this package are the classes *StructuralEntity* and its sub-classes, as well as the classes *Quantity*, *Terminal*, and *TerminalJunction*.

5.3.1 *StructuralEntity*

StructuralEntity is defined as an abstract class. It has the attributes *name* as an internal identification and also as an indicator of role, and *description*, a textual description of the entity.

It defines standard methods for adding or removing subentities, terminals and quantities, and for connecting and disconnecting terminals. Its sub-classes *ElementaryStructuralEntity* and *AggregateStructuralEntity* override these methods accordingly.

5.3.2 Terminal

The class Terminal has the attribute *name* as internal identification and also as an indicator of role. It defines methods for adding or removing terminal quantities, and methods related to connection and disconnection, as well as for checking compatibility and consistency of terminal connections.

In case of aggregating structural entities, terminals of the sub-entities that have no internal connection will be the external terminals of the aggregate.

5.3.3 TerminalJunction

The *TerminalJunction* class holds list of terminals joined to the junction. It exists as long as it connects at least two terminals. It is a composite part of an *AggregateStructuralEntity*, and defines methods for connection and disconnection of terminals of the sub entities. Connection and disconnection methods take into account that the entity is always left in a consistent state.

5.3.4 Quantity

Quantity class has the attributes *name* and *category*. A boolean attribute can also be associated to a quantity that indicates its type (flow or effort). There is no practical difference between an internal and a terminal quantity. Quantity defines a parent object that indicates whether it is attached to a Terminal or to a StructuralEntity.

5.4 Package Values and Domains

The classes in this package are basic, in that they would be required in almost all other packages. Classes included in this package are *Interval*, *Value*, *ValueSet* and *Domain*, and their respective subclasses.

5.4.1 Interval

The class *Interval* is defined as abstract. Its specialization *NumericInterval* is designed to take two numeric values as its lower and upper bounds. The data type of the interval is determined by the data type of the bounds, both of which must be of the same data type. The lower bound must be less

than or equal to the upper bound. An enumeration flag gives whether the interval is open, includes the lower bound, the upper bound or both.

Methods: in addition to default set and get methods, the class implements:

Contains() Checks if a given value lies within the interval. The value to be check could be a discrete numeric value or another interval.

Equals() checks equality between two intervals. Returns true if the bounds of two intervals are correspondingly equal, and if the intervals have the same inclusion type.

isSingleValued() checks if the lower and upper bounds of the interval are equal.

Overlaps() Checks whether the interval overlaps with another interval given as argument, i.e. if the two intervals have an intersection.

The class also implements methods for set manipulation including *Union()*, *Intersect()*, *Minus()*, and *Merge()*.

5.4.2 Value

The class *Value* (Figure 22) is defined as an abstract one that wraps a value object, i.e. it is defined as a container of a value object. The contained value object will be specialized by the sub-classes.

Class *DiscreteNumericValue* is a wrapper class for a single numeric value. As default, the value object is set to a rational number (double). Other class constructors can set it to any other numeric value type.

Class *QualitativeValue* specifies the wrapped object to be a *NumericInterval* associated to a label or designation. The class implements the methods *Contains()* and *Equals()* that accesses corresponding methods in the associated *NumericInterval* class.

5.4.3 ValueSets

ValueSet is an interface that includes methods common to sets of both Values and Intervals. It is defined as a set, that can be a single *NumericInterval*, the *EmptyInterval*, or a collection of intervals or values (a *MOMCollection*). It is implemented as a mathematical set, i.e. it can not hold duplicates. *ValueSet* defines abstract methods that are needed for set manipulation.

ValueSetElement is an interface, common for both *Value* and *Interval*.

MOMCollection is a collection of *ValueSetElement* objects, a set that can contain both Values and Intervals. It is a specialization of *ValueSet*. In addition to the inherited abstract methods, it implements methods for insertion and removal of various types of *ValueSetElement* objects to or from the collection set.

IntervalCollection is a set containing only *Interval* objects, and *ValueCollection* is a set containing only *Value* objects. Both are specializations of *MOMCollection*.

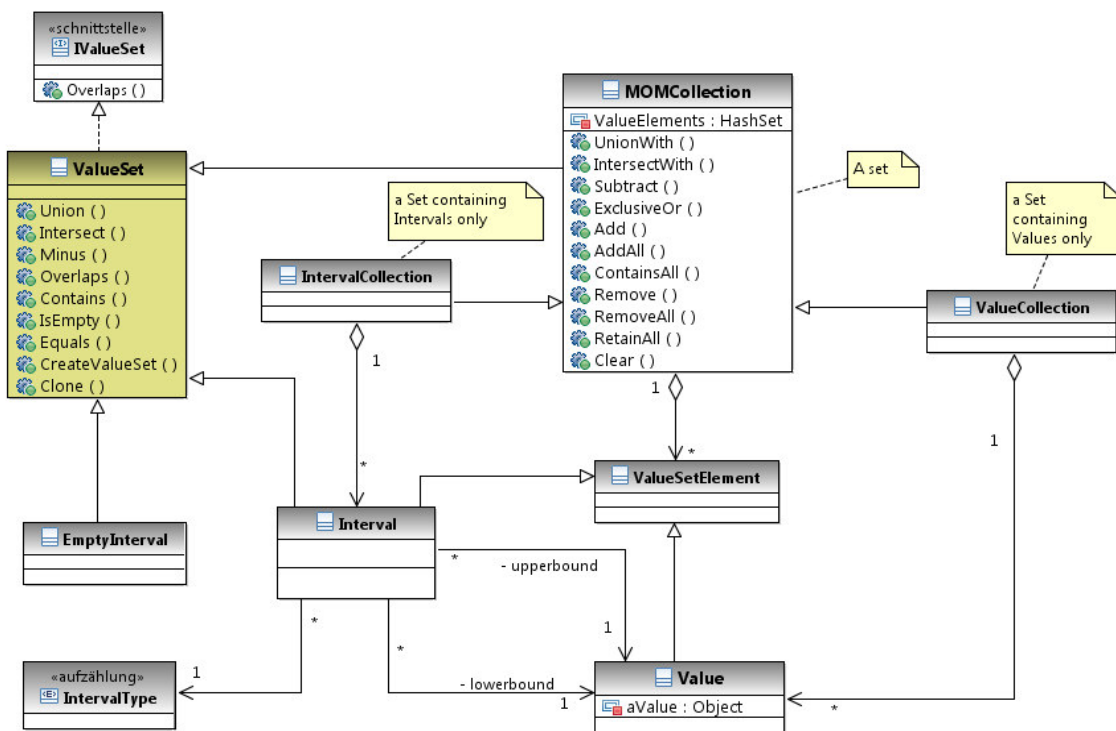


Figure 25: ValueSet and Value Collections

EmptyInterval is a numeric interval devoid of values. It can be seen as an open interval with equal bounds. It could be used to represent the result of the intersection of two non-intersecting intervals. *EmptyInterval* is a specialization of *ValueSet*. It can not be a member of any other set. Due to its special nature, the *EmptyInterval* shall not be mixed up with other non-empty values contained in a *ValueSet*.

The UML diagram in Figure 25 shows *ValueSet* and its subclasses.

5.4.4 Domains

Domain is defined as an abstract class. It has the attributes *name* as internal identification and also as an indicator of role; *datatype* holds data type of the values contained in the domain. Domain is defined as a mathematical set (i.e. it does not contain duplicates) and, thus, defines abstract methods for set manipulation, including *Contains()*, *Union()*, *Intersect()*, *Minus()*, *Overlaps()* and *IsEmpty()*.

FiniteDomain is a sub-class of *Domain*, and is itself abstract. It holds the property *valueList*, a set of objects the domain contains. It defines additional methods for adding and removing objects from the *valueList*. The sub-classes – *DiscreteNumericDomain*, *QualitativeDomain* and *SymbolicDomain* then set their *valueList* to the type of object typical of their class, and implement the inherited methods accordingly.

InfiniteDomain is defined as an abstract class. *InfiniteNumericDomain* is a subclass of *InfiniteDomain*. It has the property *domainInterval*, a numeric interval from which the values of the domain stem. *InfiniteNumericDomain* defines methods that access the methods defined in *NumericInterval*.

5.5 Package ModelFrame

This package includes the classes *Variable*, *ModelFrame*, *BehaviorDescription* and *BehaviorDescriptionAssociation*, and the respective specializations and related classes.

5.5.1 Variables

Variable is defined as an abstract class. It has the attributes *name* as an internal identification and also as an indicator of its role. It has *ValueDomain*, a domain from which it takes its values. It defines the standard methods *Equals()*, *CompareTo()*, and *GetHashCode()*.

ModelVariable is a specialization of *Variable*. It represents a certain aspect of a given Quantity. Thus, it is derived by associating a *QuantityAspect* to a *Quantity*. *ModelVariable* may have a *Domain*

BehaviorDescriptionVariable is a specialization of *Variable*. *BehaviorDescriptionVariables* are specific to the individual *BehaviorDescription* and the underlying environment (Simulink™, etc). *BehaviorDescriptionVariable* has a domain.

5.5.2 Behavior Description

The interface *IBehaviorDescription* is the root interface and defines abstract methods only for creation and deletion of variables (Figure 26). The specializations *ComputationalBehaviorDescription* and *SymbolicBehaviorDescription* are also defined as interfaces. Currently, both specify no methods. Both *DirectedBehaviorDescription* and *UndirectedBehaviorDescription* inherit from *ComputationalBehaviorDescription*. *DirectedBehaviorDescription* is defined as an abstract class. It holds lists of input and output variables, and defines the abstract methods *ComputeOutputVariables()* and *ComputeMinMaxMonotonic()*.

The interface *ITableBehaviorDescription* defines abstract methods for the manipulation of a table. The class *TableBehaviorDescription* inherits from both *UndirectedBehaviorDescription* and *ITableBehaviorDescription*. It implements methods that add or remove tuple(s) to or from a table, and methods to create a table schema or to save a table as a xml file.

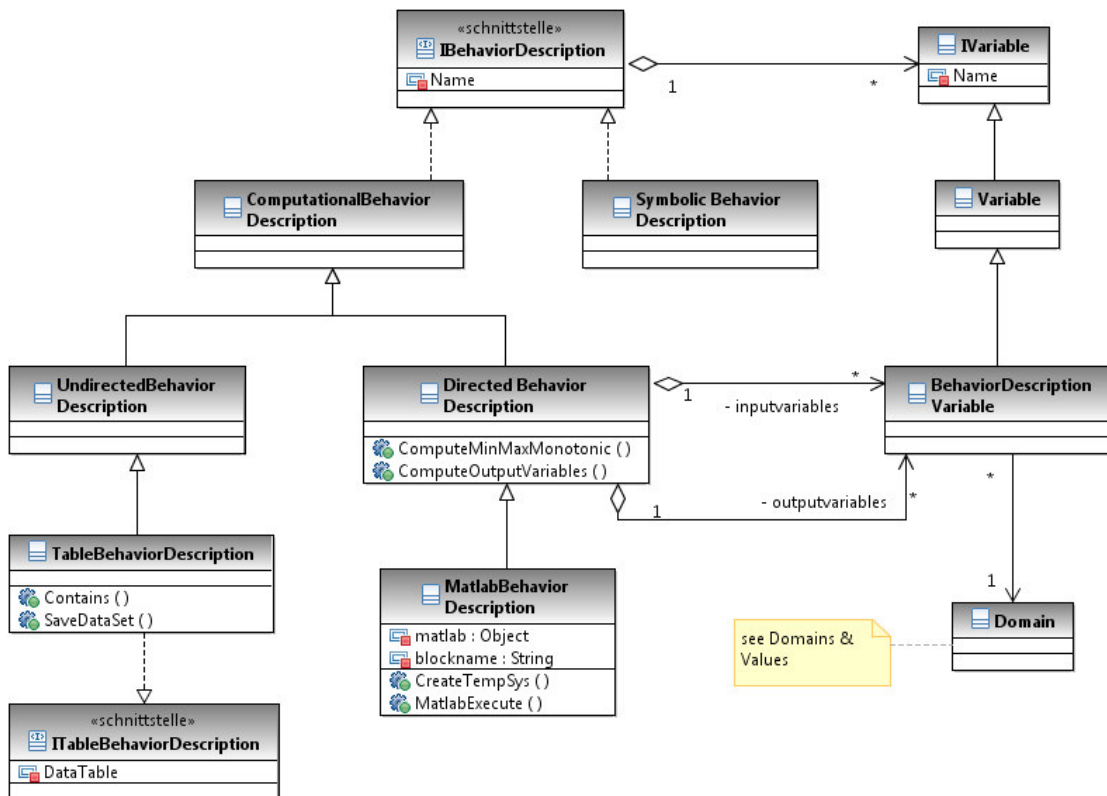


Figure 26: Classes of Behavior Description

5.5.3 ModelFrame

ModelFrame holds references to a list of BehaviorDescriptionAssociations, a StructuralEntity it describes, a list of modelvariables, and a list of VariableConnections.

It defines standard methods for adding or removing a ModelVariable to/from its list, and for compiling ModelVariable for given list of Quantity and QuantityAspect objects.

In addition, it implements the methods:

CombineBehaviorDescriptions() that compiles a combined behavior description given two BehaviorDescriptionAssociations, viz., BehaviorDescriptions as argument. For the present, this is only possible if both arguments are of the TableBehaviorDescription class.

JoinTables() joins two data tables given as argument using the indices of columns representing the table relations. This is equivalent to inner join of tables. It is used by the method *CombineBehaviorDescriptions()*

GenerateVariableMapping() generates a variable mapping for given source and target BehaviorDescriptionAssociations and a list of relevant ModelVariables.

5.5.4 BehaviorDescriptionAssociation

BehaviorDescriptionAssociation simply associates a BehaviorDescription with a ModelFrame via a mapping between the Variables of the BehaviorDescription and the ModelVariables of the ModelFrame. A ModelFrame can have a number of such associations.

5.6 Package Mapping

As is described in section 4.6.2, mapping is used to associate variables (and thereby their corresponding domains and values) from two related behavior descriptions. Here, variable mapping is used to set a relation between a ModelVariable and the corresponding BehaviorDescriptionVariable. At the same time, DomainMapping captures the relationship between the domains of the two mapped variables. But Variable- and DomainMapping could be specified for any two setups and used for different purposes, e.g. during abstraction of a qualitative model from a numerical one. The

object diagram in Figure 27 illustrates `VariableMapping`, `DomainMapping` and `ValueMapping` for two related behavior descriptions.

5.6.1 VariableMapping

`VariableMapping` is defined with a lookup table that maps two variables and the reference to the mapping of their two domains. Each row of the table contains the two mapped variables and the mapping between their domains as a tuple. The column value types are set correspondingly to `Variable` and `DomainMapping`.

The class defines the methods `MapValues()` that takes a variable and a value of that variable, and returns the equivalent value of the mapped variable. The Method `MapVariable()` takes a variable as argument and returns the corresponding mapped variable.

`SingleVariableMapping` corresponds to a row in `VariableMapping`.

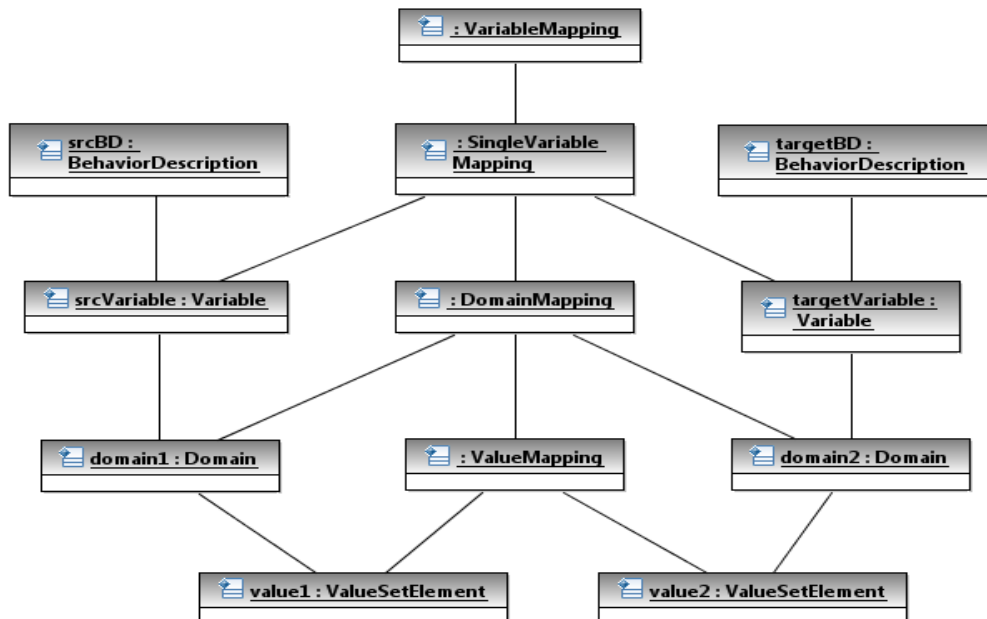


Figure 27: Object Diagram of Variable-, Domain- and Value-Mappings

5.6.2 DomainMapping

Domain mapping captures the relationship between the values of two domains, for instance of the domains of two variables related by a variable mapping. `DomainMapping` is defined as an abstract

class that simply relates two domains. It declares an abstract method *MapValues()* that takes a value of one domain, and returns an equivalent value from the corresponding (mapped) domain.

Basically, there are two ways of mapping domain values: tabular and using functions. *FiniteDomains* usually contain a finite set of objects, so a direct mapping between two finite domains is possible only using a tabular relation. On the other hand, any mapping involving an infinite domain requires a sort of a conversion function. Therefore, mapping between any two finite domains (*Qualitative*, *DiscreteNumeric* and *Symbolic*) can be performed using *TableDomainMapping*, a sub-class of *DomainMapping*.

Then, there are four combinations involving infinite numeric domain: Infinite to infinite, infinite to qualitative, infinite to discrete numeric and infinite to symbolic. A mapping between an infinite numeric domain and a symbolic one makes no sense, unless the symbolic value elements have any connotation with real numbers (which in effect makes them qualitative values).

Actually a mapping between *InfiniteDomain* and *QualitativeDomain* is intrinsic in the definition of a *QualitativeDomain*, since a *QualitativeDomain* contains *QualitativeValue*, which in turn is defined as a designation representing a numeric interval.

Infinite to discrete domain mapping can be conceived as a sort of discretization. This means, each discrete value element of the *DiscreteNumericDomain* must be related to some numeric interval (just like a qualitative value). The relation could be based on, for example, some modulo operation, randomization, etc.

The last type of mapping is infinite to infinite, which actually implies identical domains, but if the data types are different (e.g. double to integer) the mapping involves conversion of the data type.

CompositeDomainMapping represents the composition of two domain mappings which share one domain.

5.6.3 ValueMapping

ValueMapping is mapping between two *ValueSets* stemming from different domains. It represents the equivalence of the two values.

5.7 Interfaces to External Environments

Currently only *MatlabBehaviorDescription* is implemented as an interfaces to Matlab™/Simulink™. It is implemented as a specialization of *DirectedBehaviorDescription* itself a specialization of *BehaviorDescription* (section 4.4.8). A module *MatlabBehaviorDescription.dll* had been implemented in a previous work [Fraracci08] and a draft description is available². The current work examined the existing implementation, and added improvements and necessary changes to adapt it to the needs of MOM.

MatlabBehaviorDescription requires that a pre-installed version of Matlab™/Simulink™ is available on the machine, preferably version 7.1 or later. It takes a the name of the simulation (*.mdl) file and the name of the block therein as arguments, and opens an ActiveX session with the Matlab automation server. One can then send a series of commands over the session to manipulate the selected block. A number of methods have been defined in *MatlabBehaviorDescription* for such purposes.

The modus of operation of *MatlabBehaviorDescription* is opening an ActiveX session with the Matlab™/Simulink™ application, where the application acts as an automation (COM) server. However, the ActiveX communication is difficult to utilize, because there is no defined interface for exchange of arrays and matrices between the Matlab workspace and Windows. Therefore, the type of data that can be exchanged is limited to strings only. Recently, Matlab™/MathWorks Inc. has issued an add-on called MATLAB Builder for .NET that creates a .NET assembly out of Matlab program files that can then be linked to C#. But still, this is not the solution for the mode of operation intended for MOM.

5.8 MOM Operators

The class *Operator* is defined as an interface that serves as the root for all possible operators. Currently, no specific methods are declared. The abstract class *AbstractionOperator* inherits from the interface *Operator*, and is intended as a general class for all abstraction operations. The class *NumericDomainAbstractionOperator* is a sub-class of *AbstractionOperator* and serves as a super class for abstraction from computational behavior descriptions. It defines an abstract method *GenerateRelation()*.

² "Matlab Behavior Description", a draft document describind the module, by A. Fraracci, 20.04.2008

The class *NumericDomainAbstractionOperatorDirected* specializes *NumericDomainAbstractionOperator* and is used for the abstraction of qualitative models from directed behavior descriptions such as Matlab™/Simulink™. The method *GenerateRelation()* takes a variable mapping, a *DirectedBehaviorDescription* as the source and a *TableBehaviorDescription* (initially containing only the table scheme) as the target behavior descriptions and generates the target.

Currently, it is implemented and tested in conjunction with *MatlabBehaviorDescription* to abstract a qualitative relation from a given Simulink™ simulation block.

The Abstraction operator has been described in detail in [Fraracci08].

6. Evaluation and Discussion

The MOM project is large, and it was obvious that time does not allow to realize the project in full within the framework of the current thesis. This work, thus, basically aimed at laying the ground work for the overall project, including specification of the major components, and design and implementation of some of the basic software elements on which the rest lie. The previous chapter discussed the different software modules that have been implemented within the current framework.

As the full extent of the project is not yet realized, the implemented software components could only be tested individually, and only in certain constellations. The present evaluation is thus limited mostly to the unit tests. Only the module *NumericDomainAbstractionOperatorDirected* has been implemented that utilizes several other classes/modules.

Obviously, a suitable graphics user interface (GUI) is indispensable for such an editing system as MOM, but this is not yet developed. For demonstration of some of the components implemented within the current work, specifically for the AbstractionOperator, a test GUI has been implemented as shown below.

6.1 The Abstraction Operator

In this section, qualitative abstraction of some components of the Landing Gear System modeled in Simulink™ (described in chapter 3) will be demonstrated using the test GUI. The implementation of the CheckValve-function in Simulink™ is shown in Figure 28, which also includes different failure modes of the check valve.

As stated earlier, the first step towards qualitative abstraction is choosing the variables relevant in the resulting qualitative model. In the present example, the quantities pressure at inlet (Pinlet), pressure at the outlet (Poutlet) and the inflow (Q) are considered adequate for modeling. The corresponding variable in the target qualitative relation are then named *Pin*, *Pout* and *Qin*, respectively. The next step is setting the domains for these target variables. As pressure can not be negative, the qualitative value "zero" is to represent zero pressure (closed interval $[0, 0]$), and the qualitative value "pos" is to represent some positive pressure (open interval $(0, \infty)$). Thus the domain for pressure will be {"zero", "pos"}. Following common convention, flow(rate) is considered positive if it is into, and negative if it is out of the component. Thus, the qualitative value "neg" will represent some negative flow (open interval $(-\infty, 0)$). Zero and positive flow will be

represented by "zero" and "pos" with the intervals as defined above. Thus, Q_{in} will have the domain {"neg", "zero", "pos"}.

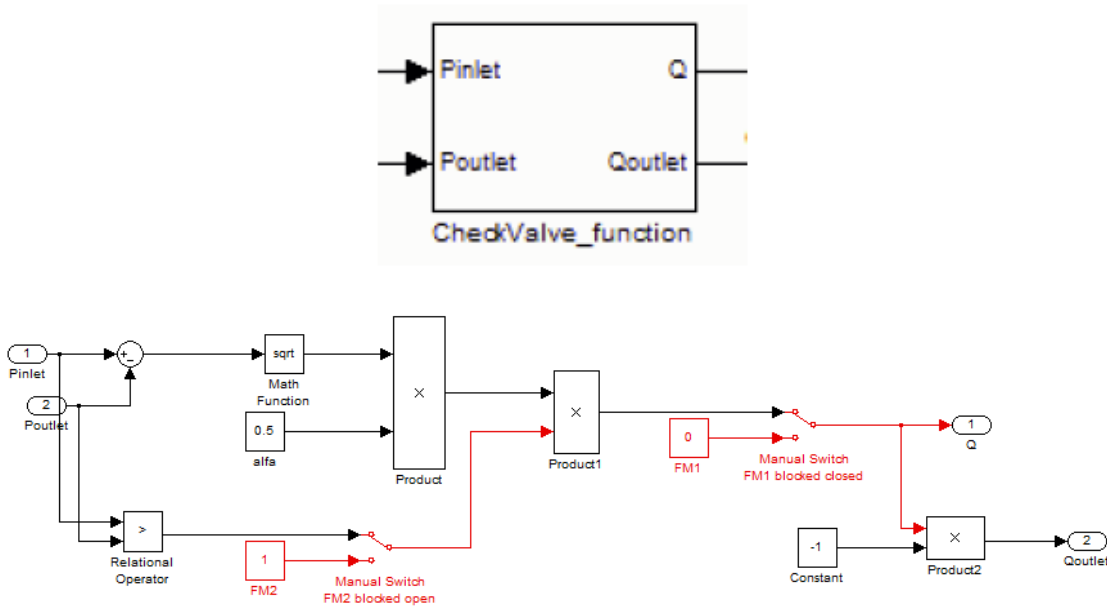


Figure 28 The CheckValve-function modeled in Simulink™ including different failure modes [Fraracci08]

The next step in the abstraction process is to map the two sets of variables. In the current case, Pin will be mapped to $Pinlet$, $Pout$ to $Poutlet$ and Q_{in} to Q . Q_{outlet} remains unmapped.

These steps in the Abstraction GUI are as follows (Figure 29): With the button Load File, one opens the directory (usually in the directory 'work' under the Matlab installation directory) in which the simulation (*.mdl) file is saved. This opens the simulation file and a Matlab command window. One has then to type the particular Simulink™ block of interest. The selected block must not include any integration or similar delaying step(s) (see discussion in chapter 3). Clicking the button "Retrieve Variables" calls the function *GetVariables()* in *MatlabBehaviorDescription* and displays the *input* and *output* signals of the block with their default domains as shown. The table in the GUI also includes columns for the target variables, including suggested variable names and buttons for setting target domains of each. Before that one can deselect the source variable which does not need to be included in the qualitative abstraction. This deselection will, however, be overridden if the concerned variable is an input signal in the source simulation.

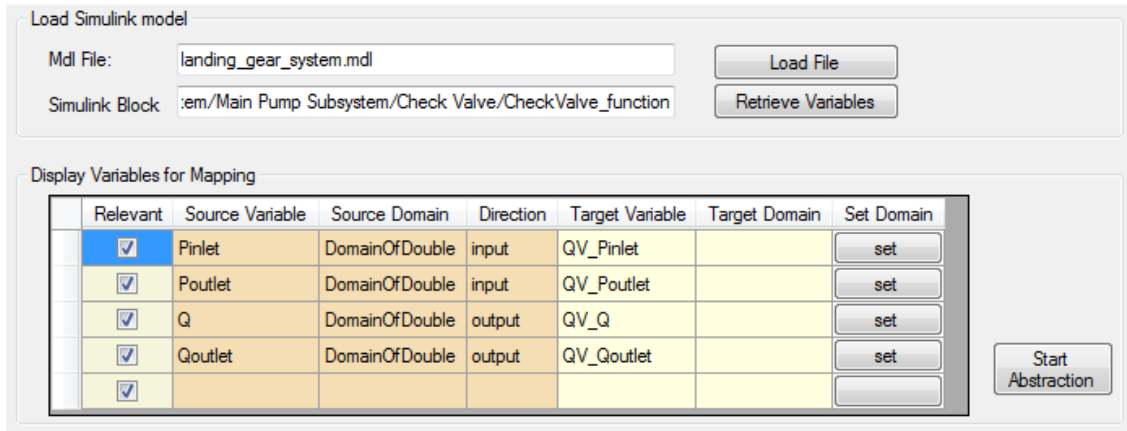


Figure 29 Loading a simulation block and retrieving the variables

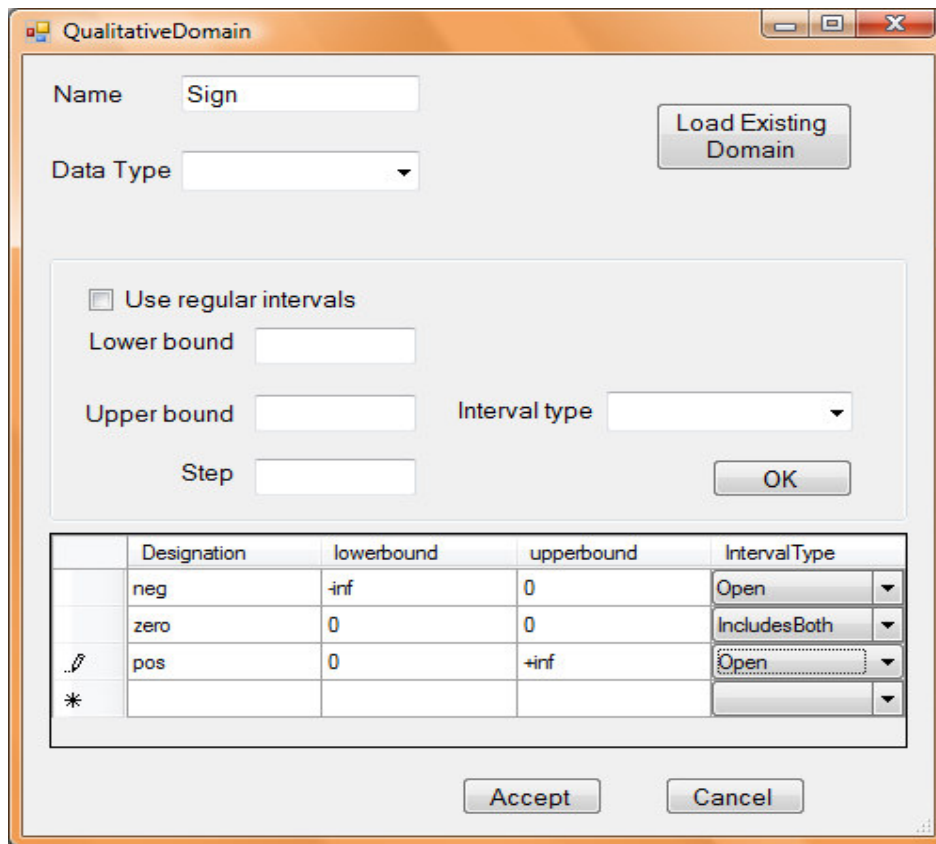


Figure 30 A dialog for editing or loading a qualitative domain

Clicking the button "set" opens the dialog for setting the domain for that target variable (Figure 30). With the dialog, one can load an existing qualitative domain (saved as a xml file) from the file system, or create a new one. One can, hereby, generate a set of qualitative domains by specifying regular intervals within a larger one by setting the limits of the larger interval and an increment step.

The resulting set of qualitative values will be listed in the table as shown in Figure 31. One can then tweak one or the other parameters in the list. One can also fill the table manually (Figure 30). The button "*accept*" creates a qualitative domain out of the table of values and associates it with target variable in the concerned row.

The button "*Start Abstraction*" starts the actual abstraction process (Figure 32). The result of the example run is shown in Figure 33. One can edit the resulting table and save it as an xml file

Evaluation of the result:

As is also tested by [Fraracci08], the abstraction process runs, for the cases treated above, excluding the time necessary to open Matlab™/Simulink™ and to load the numerical model, in less than a second on a relatively recent machine (CPU 2x2GHz, 2GB RAM) under Windows Vista.

Of course, the number of internal iteration are exponential in the number of variables and the size of the variable domains. If n variables are involved in the abstraction operation, and if domain of the i^{th} variable has m_i value elements (intervals), the number of input tuples that result from the combination of these values equals the product $m_1 \times m_2 \times \dots \times m_n$. Each of these tuples will be an input into Matlab per cycle of iteration. Within *MatlabBehaviorDescription*, another combinatoric is formed out of the two endpoints of each of the intervals in an input tuple, that practically results in 2^n possible combinations to be computed. *MatlabBehaviorDescription* then computes the min/max of these 2^n computation results and returns the two as values of the output variable(s). Essentially, this means $2^n \times \prod (m_i)$ computations would result. One could avoid repeated inclusion of the intermediate interval points, in which case the combinatorial size would reduce to $\prod (m_i + 1)$. However, one needs to cache the results of the intermediate computations, which could be prohibitive. This combinatorial problem and other complexities has been pinpointed by [Struss02].

The process of designing and debugging qualitative models by hand may take days, so that in spite of these complexities, if numerical models are already available, automated abstraction can significantly reduce the required time and effort.

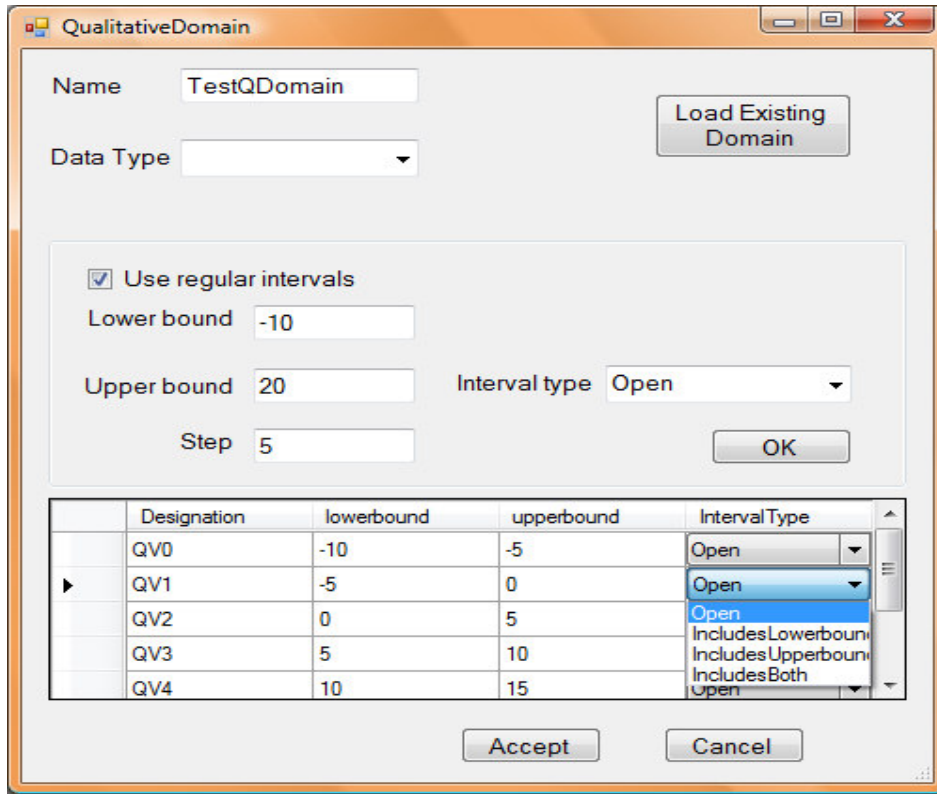


Figure 31 Editing a qualitative domain

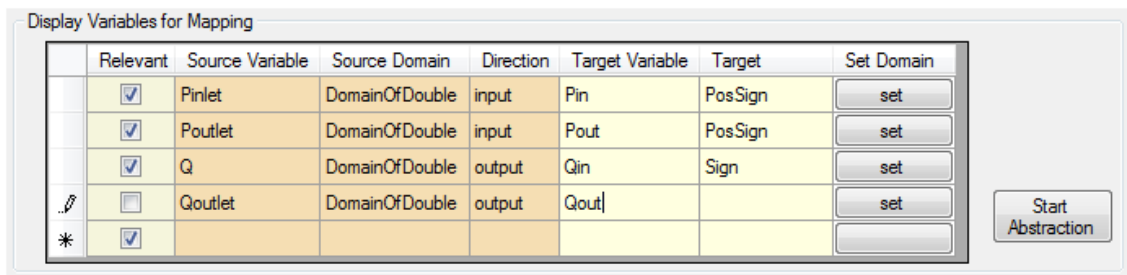
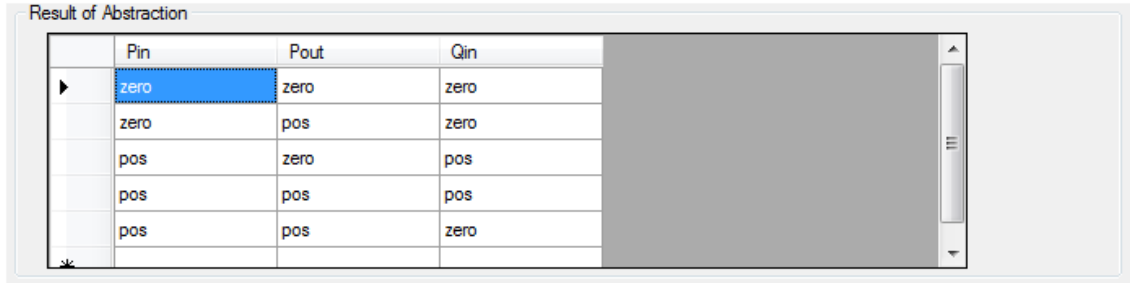


Figure 32 Setting the domains of the target variables



	Pin	Pout	Qin
▶	zero	zero	zero
	zero	pos	zero
	pos	zero	pos
	pos	pos	pos
	pos	pos	zero
*			

Figure 33 Result of the abstraction process

6.2 Other Built-in Modules

During the current work, a number of software units that serve as the basis for MOM have been designed and implemented. These units include the packages described in chapter 5. Classes related to the static structure have been implemented to the extent that they allow local operations on entities, such as aggregation, connection/disconnection of terminals/terminal junctions and adding or removing of sub-entities, terminals and quantities. Only unit tests could be conducted on these classes.

Classes related to Domains and Values have also been unit tested, and also within the operation of domain and value mappings in conjunction with the abstraction operator discussed above. Classes related to mapping have also been used in the abstraction operation, and seem to function well. However, intensive testing and evaluation are needed to point out weaknesses and missing functionalities.

As is illustrated in section 4.5, MOM's interaction with Matlab™/Simulink™ (through MatlabBehaviorDescription) is over ActiveX (COM interface), which is limited to exchange of command strings. The retrieval of simulation and other command results is based on parsing the result strings. It is not guaranteed if future version of Matlab™/Simulink™ could be used in the same manner. It is currently tested with Version 7.1 of the application.

6.3 Model Composition

The Model building aspect of the project has not been designed or implemented. It requires a consistent hierarchy of decision classes and availability of a model library. Elementary classes for StructuralEntity, Quantity, Terminal and TerminalJunction have been implemented. These could be

used to create models for common structural entities. In conjunction with the ModelFrame and other packages, these could be used to start a model library.

The screen shot in Figure 34 shows the different elements of a sample GUI. It depicts suggestions for some of the elements a future graphics user interface for MOM may need to contain. It includes a table for editing and saving tabular data such as table behavior descriptions or qualitative values of a domain.

It also shows a library and other directories in a tree structure and a details window below it. Beside it is another window which can be used to show a model in a tree structure. The tree can then be manipulated by providing context sensitive menus. One can also implement drag-and-drop between the elements of the tree, for example to transfer elements or to initiate terminal connections.

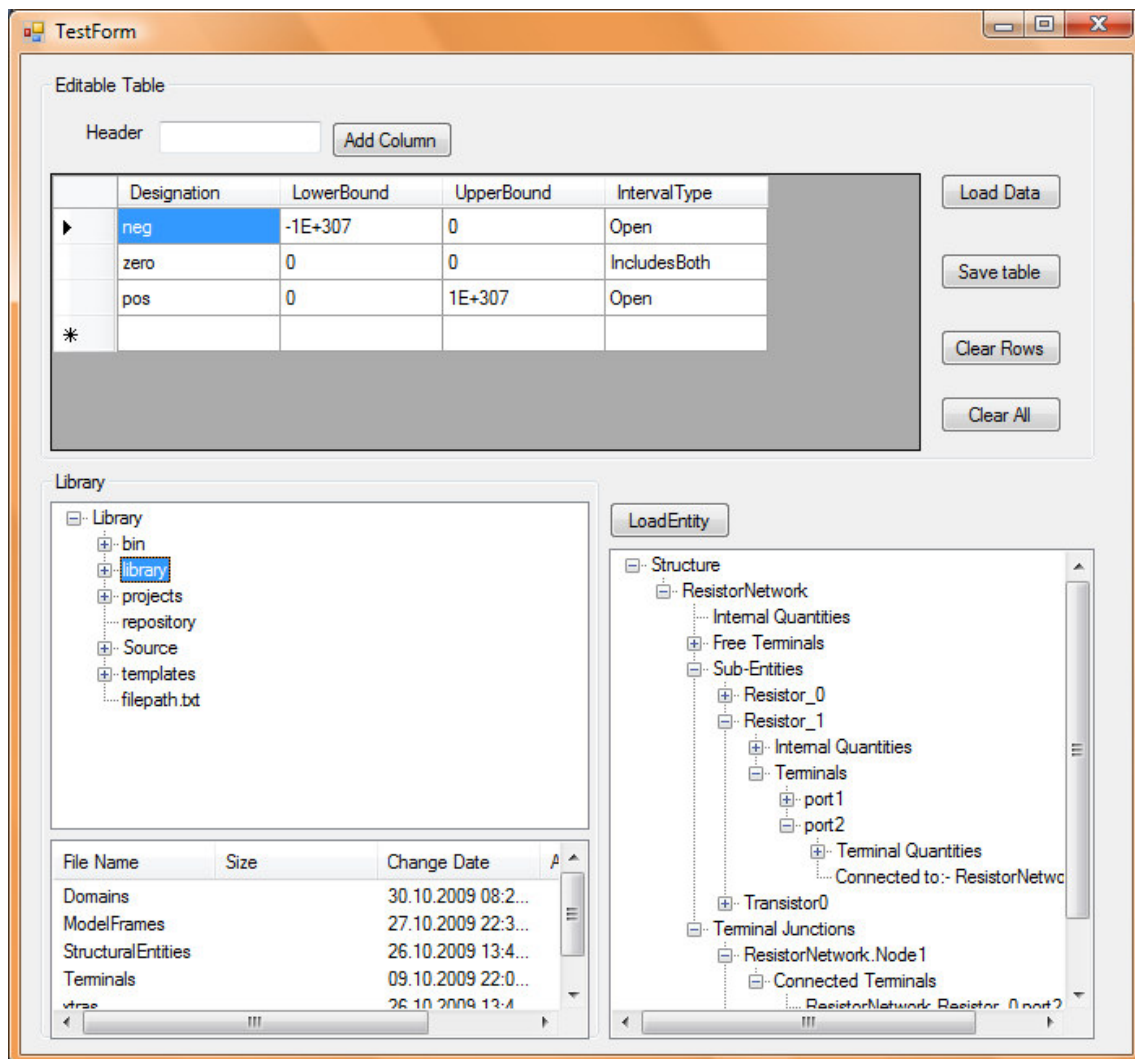


Figure 34: Sample Window showing some of the components a future MOM GUI should provide

7. Conclusion and Suggestions

7.1 Review

Modeling a physical system (or a complex theoretical issue) is a laborious and time consuming task. It involves taking various facts and different aspects of issues into consideration, and making assumptions and taking decisions on multiple levels in arriving at an acceptable model that represents the situation at hand.

Multiple modeling is an approach to produce complex models by piecing together a number of simpler subsystems. It aims at building systems which operate robustly over a wide range of operating conditions by decomposing them into a number of simpler linear modeling problems, even for nonlinear ones. Such a modular approach has computational advantages, in that it lends itself to adaptation, and allows direct incorporation of high-level and qualitative domain-specific knowledge into the model.

Not only modularization, but also making the basic modules generic and re-usable is the key to reduce the time and energy required to build model-based solutions for the increasingly complex engineering and scientific problems.

The preceding chapters presented the analysis of and solution approaches to modeling and model building. Chapter 1 introduced the concept of Qualitative Reasoning, the need of multiple modeling, and the aims of the current work. Chapter 2 dealt with the different types of models, model-based systems and the different approaches of the scientific community to modeling, and specifically to qualitative modeling.

Chapter 3 was concerned with the model building process. It discussed in depth what makes up a system and how these interact in determining the system's behavior. It identified basic constituting elements of a system and introduced fundamental concepts concerning structural entity, terminal, quantity and variable. Compositionality of models and the multi-level decisions associated with selecting a particular mode of operation and formalism of describing the behaviour of a given entity were elaborated. It briefly introduced the concept of decision-based model building.

Chapter 4 broadly discussed the aim of the Model Manipulation Environment (MOM) project and the functionalities it has to provide. It illustrated the concept of decision-based model building and presented a solution framework. It identified object classes and software components to be designed

and implemented. Chapter 5 then presented design solutions and implementation of the basic software components. Chapter 6 presented a brief evaluation and discussion of few of the implementations.

7.2 Conclusions

The present work is part of a project aimed at realizing a development environment that provides a platform for manipulation and transformation of according to the needs of specific tasks. The current thesis laid the groundwork for the project in designing the general framework and implementing some of the basic software components.

Generic interfaces have been designed and in part implemented that enable interaction with external modeling environments as to utilize computational capabilities of such numerical applications as Matlab/Simulink or Spice. Implemented software components also include those that enable hierarchical representation of entities and the association of multiple behavior description to them.

A number of modeling paradigm have been presented by a number of researchers, but none of the approaches are generic to be applicable for a wide range of domains. The current work tried to present yet another approach. The put the idea forward that rests upon not only providing a library of generic, context-free and re-usable model elements, but also on basing the selection of the building units on consistent and hierarchical sets of decisions.

Although the concept of decision-based model building is presented as a working idea, we are sure that it can be realized with relative ease. The current thesis has put necessary basic components and concepts in place.

7.3 Outlook

Obviously, the one major handicap is that a suitable and all encompassing Graphical User Interface is not yet implemented. It is going to be one of the next major works in MOM. Design and implementation of a GUI that meets the demands of all the anticipated operations in MOM itself is a major undertaking and should be carefully planned and executed.

Current implementations of software components are not trimmed towards efficiency, because in most cases, compatibility with older implementations and other projects have to be assured. In some instances, alternative methods have been devised to bypass the restrictions. Future works must consider other solutions to the compatibility problem, and device efficient and easily maintainable code.

Appendix

Glossary

AggregateModelFrame is a subclass of *ModelFrame* that comprises two or more model frames. Each ModelFrame could in turn be elementary or aggregate.

AggregateStructuralEntity is a subclass of *StructuralEntity*. It is composed of more than one structural entity.

BehaviorDescription represents a concrete technical realization of the mathematical behavior/functionality of an entity in a specific system (e.g. MatLab/SimuLink).

BehaviorDescriptionVariable is a Variable in a *BehaviorDescription*, and is specific to the underlying environment (SimuLink, etc).

CompositeDomainMapping represents the composition of two domain mappings sharing one domain.

ComputationalBehaviorDescription is a BehaviorDescription in which values of the variables can be computed.

DirectedBehaviorDescription - a BehaviorDescription in which one distinguishes between input and output variables, according to the direction of the computation. The values of the input variables have to be given in order to be able to compute the values of the output variables.

DiscreteNumericDomain represents a finite set of single numeric values, i.e. a finite set of discrete integers or real numbers.

Domain is a set of values a variable could take. There exist 2 different types of domains – Finite and Infinite. A finite domain could further be divided into: Qualitative, Symbolic and Discrete Numerical Domains

DomainMapping captures the relationship between the values of two domains, for instance of the domains of two variables related by a variable mapping. But it could be specified for any two domains independently of this context (and used for different purposes)

Elementary Structural Entity is a subclass of *StructuralEntity*. It represents a structural entity that can not be further decomposed into smaller structural units.

ElementaryModel Frame is a subclass of *ModelFrame*. It represents a model frame which is not a composition of any other model frame..

EmptyInterval: a numeric interval excluded of values. It is the case where the the two bounds are equal and but both are excluded.

FiniteDomain represents a domain whose values stem from a finite set, such as a list of names.

FiniteDomain is a specialization of Domain.

InfiniteDomain represents a domain whose values stem from an infinite set such as natural or real numbers. InfiniteDomain a specialization of Domain.

InfiniteNumericDomain is a subclass of Infinite Domain, which is predefined and is represented by an identifier of a typical class of numbers, possibly with a specified range. e.g. Real Numbers,

Interval A general designation of a set of (ordered) values between two boundaries. An Intervals could be represented by a pair of landmarks (bounds) plus additional information about whether it is closed or not

IntervalCollection Is a specialization of MOMCollection. It contains *NumericIntervals* only.

IntervalType specifies the type of interval. The types could be enumerated as open (i.e. both bounds excluded, closed (i.e. includes both bounds), includes lower bound only, or includes upper bound only.

Landmark Landmarks are boundaries of intervals. In a numerical function, landmarks may be used to designate the positions where significant change take place.

Library a directory dedicated to containing standardized elementary model units.

ModelVariable is a variable that represents a given aspect of a given Quantity. Thus, it is derived by associating a *QuantityAspect* to the Quantity.

NumericInterval represents a (infinite) set of numeric values of a given type between two bounds (Landmarks).

OrderedDomain A Domain with an intrinsic total order of values.

QualitativeDomain represents a finite set of qualitative values and is a subclass of FiniteDomain. Each QualitativeValue is associated to an underlying numeric interval.

QualitativeValue is a subclass of *Value* and represents a name or designation of an underlying numeric interval.

Quantity represents aparameter or a state variable of a structural entity. It may be associated with an interface or it represents the internal state of the entity.

QuantityAspect refers to the different facets used to describe the same quantity. It may be the magnitude, deviation, derivative, derivative deviation, etc. of the quantity.

QuantityCategory quantities can be categorized according to the type of physical properties they represent, eg. pressure, force, energy, etc.

StructuralEntity represents one of the units that are the building blocks of a (physical) system to be modeled. It can be an aggregated one (*AggregateStructuralEntity*), i.e. it is composed of several other structural entities or it is an elementary one (*ElementaryStructuralEntity*) that does not contain other structural units.

SymbolicDomain represents a finite set of symbolic values. In contrast to qualitative values, no intervals are associated with symbolic values.

SymbolicValue is a subset of *Value* that may be any object.

Terminal is a point of communication of a structural entity with its surroundings.

TerminalConnection represents a passive canal which connects the structural entities and through which exchange of information takes place. It connects two *Terminals* and it is part of a *Structure*.

TerminalJunction is a point of connection of two or more *Terminals*, i.e. a node where connected terminals meet.

UndirectedBehavior Description a Behavior Description in which the computations have no direction (no distinction between input/output).

Value could be a magnitude assigned to a variable. It could be symbolic (an object), numeric (discrete), or a qualitative designation of a numeric interval.

Variable is a name or symbolic representation of a value that may change. Variables are selected attributes of a system. They characterize the system or its components and play a major role in model manipulation.

VariableConnection represents a mapping between two variables in different behavior description that stand for the same quantity.

VariableMapping represents the association of two sets of variables. An important special use of this class is the mapping of the corresponding variables in a *ModelFrame* and in an associated *BehaviorDescription* and the mappings between their domains.

References

- [Addanki et. al., 1989] S. Addanki, R. Cremonini, J. S. Penberthy; *Reasoning About Assumptions in Graphs of Models*, in Readings in Qualitative Reasoning, 1989, p.
- [Addanki et. al., 1991] S. Addanki, R. Cremonini und J.S. Penberthy; *Graphs of models*; Artificial Intelligence, 51(1-3), October 1991.
- [Bassa, et al.] *Vânia Bessa Machado and Bert Bredeweg; Towards Interactive Tools for Constructing Articulate Simulations*; <http://staff.science.uva.nl/~bredeweg/pdf/qr2006b.pdf>
- [Bredeweg et al; 2006] Bert Bredeweg, Paulo Salles, Anders Bouwer and Jochem Liem; *Towards a Structured Approach to Qualitative Modelling*; <http://staff.science.uva.nl/~bredeweg/pdf/qr2006b.pdf>
- [Con03] L Console, G Correndo, C Picardi; *Deriving Qualitative Deviations from Matlab™ Models*, 14th International workshop on Principles of Diagnosis; 2003
- [Falkenhainer and Kenneth, 1990] Brian Falkenhainer and Kenneth D. Forbus; *Compositional Modeling of Physical Systems*; Readings in Qualitative Reasoning, 1990,
- [For84] K.D. Forbus, Qualitative process theory, *Artificial Intelligence* **24** (1984), 85–168.
- [For86] K.D. Forbus, *The qualitative process engine*, Technical report, University of Illinois, Department of Computer Science, 1986.
- [Fraracci & Struss 2006] *Common Example: a landing gear system - Qualitative model*, Internal Document Nr. AUTAS-4-TUM-3-2, 2006
- [Fraracci 2008] Alessandro Fraracci, *Model-based Failure-modes-and-effects Analysis and its Application to Aircraft Subsystems*; A Dissertation at the Technische Universität München (TUM), 2008
- [Fritzson 2006] Peter Fritzson: *Introduction to Object-Oriented Modeling and Simulation with OpenModelica, Tutorial*; Linköping University, Dept. of Comp. & Inform. Science, <http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/releases/1.5.0/doc>
- [Fritzson, et al. 2009] Peter Fritzson, et al.; *OpenModelica System Documentation*; Version, 2008-01-27 for OpenModelica 1.4.5, January 2009; Linköpings Universitet, Department of Computer and Information Science, Linköping, Sweden

- [Hel01] Ulrich Heller and Peter Struss; G+DE - The Generalized Diagnosis Engine; Proceedings of the 12th international workshop on principles of diagnosis, 2001.
- [Hin03] Hinkkanen A., Lang K. R. , Whinston A. B. ; *A Set-Theoretical Foundation of Qualitative Reasoning and its Application to the Modeling of Economics and Business Management Problems*; Information Systems Frontiers 5:4, 379–399, Kluwer Academic Publishers 2003
- [Karnopp, et al., 1990] Karnopp D., Margolis D., and Rosenberg R.; *System Dynamics: A Unified Approach*, John Wiley and Sons, 1990.
- [Kle84] J. de Kleer and J.S. Brown, *A qualitative physics based on confluences*, Artificial Intelligence **24** (1984), 169–203.
- [Kui86] B. Kuipers, *Qualitative simulation*, Artificial Intelligence 29 (1986), 289–388.
- [Kui94] Kuipers, Benjamin; *Qualitative Reasoning – Modeling and Simulation with Incomplete Knowledge*, The MIT Press, 1994
- [MOMa] Oskar Dressler, Michael Esser, Alessandro Fraracci, Tesfaye Regassa, Peter Struss, Tao Yu, *Requirements Analysis of MOM*, MQM (Model-Based Systems & Qualitative Reasoning, TUM) internal document, 2005-2009
- [MOMb] Oskar Dressler, Michael Esser, Alessandro Fraracci, Tesfaye Regassa, Peter Struss, Tao Yu, *Analysis and Static Structure of MOM*, MQM (Model-Based Systems & Qualitative Reasoning, TUM) internal document, 2005-2009
- [Paritosh 2003] Praveen K. Paritosh; *A Sketch of a Theory of Quantity*; 17th International Workshop on Qualitative Reasoning, Brasilia, August 2003
- [Pic07] Picardi C., Salles P., and Wotawa F.; *An introduction to model-based systems*, AI Communications 20 (2007) 1–6
- [Razr] OCC'M Software GmbH, www.occm.de.
- [Rodon] UpTime Solutions AB, Sweden, <http://www.uptimeworld.com/Rodon.aspx>
- [Sac01] M. Sachenbacher, and P. Struss; AQUA: A Framework for Automated Qualitative Abstraction; in Proceedings of the 15th International Workshop on Qualitative Reasoning, 2001.
- [Sac05] M. Sachenbacher and P. Struss. Task-dependent qualitative domain abstraction. *Artificial Intelligence*, 162, 2005.
- [Spice] Simulation Program with Integrated Circuit Emphasis, <http://zone.ni.com/devzone/cda/tut/p/id/5413>

- [Str97] Struss, Peter; *Model-based and qualitative reasoning: An introduction*; Annals of Mathematics and Artificial Intelligence 19 (1997) 355–381
- [Str02] Struss, Peter; *Automated Abstraction of Numerical Simulation Models - Theory and Practical Experience*, Workshop on Qualitative Reasoning, Sitges, Catalonia, 2002.
- [Str04] Struss, P., Deviation models revisited. In Working Papers of the 18th International Workshop on Qualitative Reasoning 2004.
- [Str08] Struss, Peter; *Model-based Problem Solving*, in Handbook of Knowledge Representation, Edited by F. van Harmelen, V. Lifschitz and B. Porter; Elsevier B.V. , 2008
- [Tra03] Travé-Massuyès L., Ironi L., and Dague P.; *Mathematical Foundations of Qualitative Reasoning*; AI Magazine Volume 24 Number 4 (2003)
- [Yan03] Yan Yuhong, *Qualitative Model Abstraction for Diagnosis*; Proceedings of the 17th International Workshop on Qualitative Reasoning, Aug 2003, Brasilia, Brasil
- [SAC01] M. Sachenbacher, *Automated qualitative abstraction and its application to automotive systems*, TUM PhD thesis, 2001