An Automated Model Abstraction Operator Implemented in the Multiple Modeling Environment MOM

P. Struss, A. Fraracci, D. Nyga
Technische Universität Munich, Germany
struss@in.tum.de, fraracci@in.tum.de, nyga@in.tum.de

Abstract

Configuring system models based on a model library is the crucial means to make modeling cheap. However, different tasks may require models with modeling assumptions, phenomena captured, and, especially, different levels of granularity. The only solution to this dilemma is generating the required abstractions of the most fine-grained model available automatically. This paper focuses on domain abstraction and describes the implementation of an algorithm for producing such abstract models, i.e. finite constraints, from a numerical model with directed computation, such as Matlab/Simulink. The context for this implementation is MOM, an environment for multiple modeling. Instead of requiring a specialized implementation of model abstraction for a particular numerical simulation system and a particular constraint system, MOM provides abstract interfaces to classes of such modeling systems and, thus, enables us to implement a generic abstraction operator for arbitrary instances of numerical modeling and constraint satisfaction algorithms. We describe these abstract interfaces in MOM and how the abstraction operator uses them.

1 Introduction

The set of models used in science and engineering is vast and extremely heterogeneous. Even when the same system is modeled, and even if this is done for similar purposes, and even if this happens in one company or department – there probably exist different variants and types of models. Actually, this limits the utility of models severely, since it increases the efforts spent on model building, prevents existing models from being re-used and integrated, and blocks potential synergy effects which could be achieved by combining complementary models.

There are several reasons for and dimensions of such a variety of models; besides ignorance, individual preferences, and biased education of modelers, there exist undeniable justifications for using different models:

- the **problem** to be solved may require models with different properties and power. For instance, the engine control unit in a car must, of course, calculate numerical values for fuel injection, while diagnosis or failure-modes-and-effects analysis (FMEA) may have to work with information about qualitative behavior deviations; also a model for control purposes usually includes nominal behavior only (which some people consider a big mistake), while a model for test generation or FMEA cannot do without fault models.
- The **system** to be modeled may demand for different kinds of modeling. For many artifacts, the component structure and, hence, the model structure can be assumed as fixed, while a model of an ecological system may have to be more open and allow for dynamic extensions.
- Different modeling and simulation **tools** reflect particularities of the respective system models and, thus, are able to provide efficient computation.

Among the most important resulting variations, we can identify different

- modeling **ontologies** ((differential) equations, constraints, difference equations, finite element analysis, … and at the conceptual level components, processes, causal graphs, …),
- **temporal granularity**: continuous vs. discretized time. What can be considered instantaneous?
- **structural granularity**, both related to
  - the conceptual (physical) layer: what are the elementary units to be modeled? Which types of quantities are relevant?
  - the behavior modeling: which variables have to be involved (magnitude, derivative, deviation, …)?
- **granularity** of variable **domains**: are numerical values need (and available), intervals, or simply signs?
- **tools** used to represent the model and implement the calculations.

Our work on MOM (Model manipulation system) aims at providing an environment for the **integration** of different models along these dimensions, at the time being focusing on the last three issues. The second goal of MOM is to facilitate the **automated transformation** of models in order to achieve solutions tailored to particular requirements. In [Struss and Regassa, 2010] we presented the basics of MOM, especially the structuring of different layers of

models. In this paper, we focus on the last two dimensions and describe MOM's facilities for

- the automated generation of qualitative models from numerical ones (more specifically, directed ones), i.e. we focus exclusively on domain abstraction (as opposed, for instance, to functional or structural abstraction)
- the integration of different tools, namely the ones needed for domain abstraction: directed numerical models and constraint systems.

The task of organizing and generating multiple models has been subject to research in qualitative modeling (see, as a small selection, [Addanki *et al.*, 1989], [Falkenhainer and Forbus, 1991], [Struss, 1992], [Weld, 1992], [Nayak, 1995], [Ressencourt *et al.*, 2006], [Sachenbacher and Struss, 2005], [Torta and Torasso, 2009]), although most of the work takes a rather academic perspective. The motivation for the work presented here arose from our industrial projects, for instance in diagnosis, fault analysis, and testing, we were often confronted with existing numerical system models built by engineers, usually in order to validate control. Models supporting diagnostic tasks are usually qualitative, but still only valid if they are abstractions of proper engineering models. Hence, the task of automatically generating them from the numerical models, rather than constructing them by hand from scratch, is quite important from a practical perspective and can even be crucial to the feasibility of an industrial application.

The following section discusses the mathematical background for qualitative abstraction of (monotonic) real-valued functions, while section 3 summarizes the part of MOM that is relevant in the context of this paper. Section 4 presents the abstract interfaces for the source (*Directed-NumericalBehaviorDescription*) and target (*Finite-ConstraintBehaviorDescription*) of the abstraction operator, which is then discussed in section 5.

2   Automated model Abstraction Background

The section describes qualitative abstraction, i.e. discretization w.r.t. a finite set of discrete values (called landmarks) of a numerical function in mathematical terms (extending [Struss, 2002]).

We account for the fact that every numerical model is only approximate and that the abstraction step has to reflect the limited precision. Since a function with n dependent variables can be replaced by n single-output functions, the input to the abstraction is given by:

- a numerical model that computes one output variable y as a function of n input variables, x_i:
  $$y = f(x_1, \ldots, x_n)$$
- a set of landmark values for all input variables x_i and the output variable, i.e. $\{l_{ij}\} \subset \boldsymbol{R}$
- two continuous functions, $\varepsilon^-(x_1, \ldots, x_n)$, $\varepsilon^+(x_1, \ldots, x_n)$, that characterize the precision of the model, i.e. the base model is given by the envelope of f:
  $$R_0(f, \varepsilon^-, \varepsilon^+) = \{(x_1, \ldots, x_n, y) \mid f(x_1, \ldots, x_n) - \varepsilon^-(x_1, \ldots, x_n) < y < f(x_1, \ldots, x_n) + \varepsilon^+(x_1, \ldots, x_n)\}.$$

For monotonic (sections of) functions, it is straightforward to define and compute the model abstraction for a given sets of landmarks.

We define a qualitative value as an interval between two adjacent landmarks: $q_{i,j} := (l_{i,j}, l_{i,j+1})$.

For each tuple of qualitative input values, $(q_{1,j1}, \ldots, q_{n,jn})$, we have to compute the qualitative values of y that are consistent with this tuple. Then a tuple of qualitative input values is the cross product of such intervals, i.e. an n-dimensional rectangle, which we call an input **cell**. The corners of such a cell are given by the tuples that combine the bounding landmarks:

$Corners(cell_i) = Corners(q_{1,j1}, \ldots, q_{n,jn}) :=$
   $\{ (l_{1,k1}, \ldots, l_{n,kn}) \mid q_{i,j} := (l_{i,j}, l_{i,j+1}) \land k_i \in \{j, j+1\} \}.$

If f is a continuous function, the consistent qualitative values of y are those that have a non-empty intersection with the interval between the minimal and the maximal value that, $f - \varepsilon^-, f + \varepsilon^+$ take on in the rectangle.

Definition (Qualitative Abstraction of a Function)
Let be:
- $f - \varepsilon^-, f + \varepsilon^+$ continuous for each x_i,
- $\{q_{i,j}\}$ the qualitative values for x_i,
- *Cells* the set of cells defined by $\{\{q_{i,j}\}\}$,
- $\{q_{y,j}\}$ the qualitative values for y.

For each input cell, $cell_i \in Cells$, the range of the output y is
   $int_y(cell_i) :=$
   $[\min \{(f - \varepsilon^-)(\underline{x}) \mid \underline{x} \in cell_i \},$
   $\max \{(f + \varepsilon^+)(\underline{x}) \mid \underline{x} \in cell_i\}]$

Then
   $R_{abstr}(f, \varepsilon^-, \varepsilon^+) :=$
   $\{(q_{1,j1}, \ldots, q_{n,jn}, q_{y,j}) \mid q_{y,j} \cap int_y(cell_i) \neq \varnothing\}$
is the *qualitative abstraction* of $R_0(f, \varepsilon^-, \varepsilon^+)$.

Figure 1 illustrates this and conveys the intuition that a coverage of the envelope is constructed.

It is easy to see that the resulting qualitative model does not lose anything included in the original model and is the "tightest" abstraction w.r.t. the given landmarks:

Lemma 2.1 (Abstraction Property)
Qualitative abstraction is an abstraction, i.e.
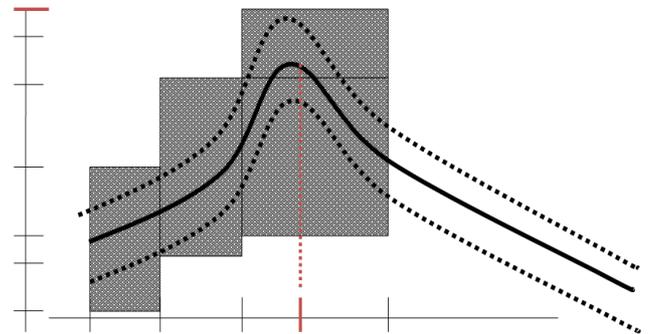 $$R_0(f, \varepsilon^-, \varepsilon^+) \subseteq R_{abstr}(f, \varepsilon^-, \varepsilon^+).$$

Figure 1 - Coverage of the envelope

Lemma 2.2 (Minimality)
The qualitative abstraction is minimal, i.e. any proper subset of $R_{abstr}(f, \varepsilon^-, \varepsilon^+)$ is not an abstraction of $R_0(f, \varepsilon^-, \varepsilon^+)$.

If the envelope functions are also monotonic, then extremal points are taken at some corners of the cell, and function values need to be computed only the at the corners in order to obtain the abstraction

Lemma 2.3 (Abstraction of Monotonic Functions)
If $f-\varepsilon^-$, $f+\varepsilon^+$ are monotonic in each x_i, then
$$\text{int}_y(cell_i) :=$$
$$[\min\{(f-\varepsilon^-)(cor) \mid cor \in Corners(cell_i)\},$$
$$\max\{(f+\varepsilon^+)(cor) \mid cor \in Corners(cell_i)\}]$$

If several variables depend on the same set of input variables (or, more generally, on sets with a non-empty intersection), the abstractions of the respective functions can be computed separately and then combined via a join.

Let $\underline{y} = (y_1, y_2) = f(x_1, \ldots, x_n)$,
$y_1 = f_1(x_1, \ldots, x_n)$,
$y_2 = f_2(x_1, \ldots, x_n)$.
As the abstraction of f, we compute
$$R_{abstr}(f, \varepsilon^-, \varepsilon^+) = R_{abstr}(f_1, \varepsilon^-, \varepsilon^+) \bowtie R_{abstr}(f_2, \varepsilon^-, \varepsilon^+)$$
where $R_{abstr}(f_j, \varepsilon^-, \varepsilon^+)$ denotes the abstraction of the function with one dependent variable as described above.

In case of multiple outputs, we may have no other way to compute the abstraction of the functions because the numerical model has been implemented this way. In this case, the result of the abstraction may produce spurious tuples if the qualitative output values are not unique, because we have to consider the cross product of the outputs. An example to illustrate this case can be found in [Fraracci, 2009].

The solution can also be applied if the functions to be abstracted do not represent algebraic, but differential equations in simply treating derivatives as separate variables. If the numerical model includes integration steps, they have to be eliminated before applying abstraction (see [Struss 2002] for a discussion). In general, all model elements relating values of a variable at different time points need to be deleted or variables be duplicated in order to avoid inconsistencies.

3   MOM Background

This section summarizes basic ideas underlying our multiple modeling environment MOM and classes that are relevant in the context of this paper (for more coverage, see [Struss and Regassa, 2010]).

MOM organizes elements of models in 4 layers (see Figure 2):

- **structural** layer, which allows to introduce "building blocks" and aggregate them, e.g. components connected to form a device,
- **physical** layer, which results from a decision, which physical quantities to associate with entities, e.g. resistance, current, voltage to a resistor,
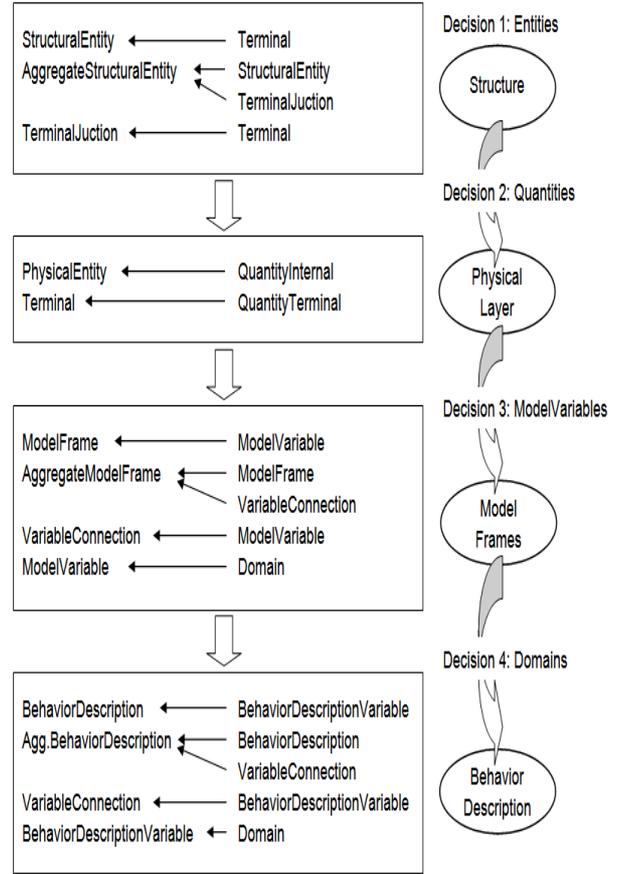

Figure 2 - Decision-based model building steps

- **model frames,** which introduce variables representing certain aspects of the quantities (such as magnitude, derivative, …) and represent "containers" for fragments of behavioral descriptions, e.g. "Ohm's Law",
- **behavior descriptions,** which capture the actual description of a behavior (aspect), involve a decision on the variable domain, and implement computations, e.g. a Matlab model of Ohm's Law or a constraint as a qualitative model of the same.

This separation allows us to make the decisions explicit that underlie the associations between objects at adjacent layers and, hence, can represent alternative associations and, thus, multiple models. An important practical impact is that for different models, all objects and structures that depend on modeling decisions that are common to them can be re-used. For instance, all elements of the upper three layers may be shared between a Matlab model and a qualitative model of a particular component or device.

This means that in MOM, model abstraction through domain abstraction is located in the lowest layer and independent of the others. Only a model frame is involved providing the hook for the original and the abstracted behavior description as alternative choices.

Variables are different objects for model frames and related behavior descriptions and also for different behavior descriptions. Therefore, mappings between the respective variable sets have to be specified and recorded. And since their domains may be different, also a mapping between their values is created. Figure 3 illustrates this for variables of model frame and different behavior descriptions. For instance, the model frame may represent the domain of the variable "position" of a valve as {closed, open}, this may be mapped to a variable "state" with the domain {0, 1} for a behavior description in Matlab.

Through these mappings, MOM provides the means for relating and even gluing together behavior descriptions of different modeling environments: structuring and executing a system model (and also transforming it), can be done without fixing or knowing how the various behavior descriptions are implemented.

Obviously, in MOM, we capture the domain abstraction underlying model abstraction as described in section 2, by domain mappings. The major methods of variable mapping are

- *MapVariable* (variable)
  returning the counterpart of the argument variable,
- *MapValues* (valueOrSetOfValues, variable)
  returning the values of the counterpart of variable that are consistent with a given value or a set thereof (for instance, specified as an interval).
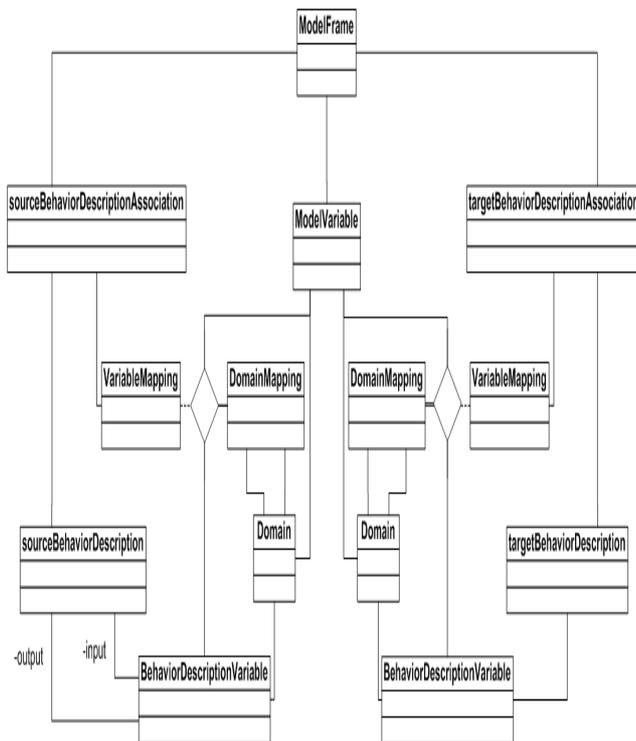

Figure 3 - Association between ModelFrame and two Behavior-Descriptions and the related VariableMappings and DomainMappings

In MOM, the principle of re-use and integration is not only applied to the objects of a represented model, but also to operators that transform models. For instance, the algorithmic structure of an implementation of the model abstraction defined in section 2 is independent of the system that performs the numerical computations (determining maximum and minimum of a function for a cell) and also independent of the system that represents the resulting finite constraints. Hence, the operator should be implemented independently of these choices. Therefore, MOM offers abstract interfaces to different types of behavior descriptions. Implementing this abstract interface for your favorite modeling environment suffices to exploit the entire functionality of MOM and, especially, the model operators.

Figure 4 shows the abstract behavior description classes in MOM with some instances for illustrations (where CS3 is the constraint system used in Raz'r [OCC'M, 2011]).

MOM contains an abstraction operator written against the interfaces *DirectedNumericalBehaviorDescription* and *FiniteConstraintBehaviorDescription*. The following section presents the parts that are relevant to this operator.

4 Numerical and Qualitative Models in MOM

4.1 Directed Numerical Models

In the version of an abstraction operator presented in this paper, the source of abstraction is some system that computes (vectors of) output values for a vector of input values, where these values can be continuous (reals) or discrete. Many standard modeling and simulation tools, such as Matlab, belong to this class, which is called *DirectedNumericalBehaviorDescription* in MOM. In contrast, Modelica is one of the instances of *Undirected-NumericalBehaviorDescription*, because it does not fix what is input and output for a computation.

Instead of implementing an abstraction operator tailored to Matlab/Simulink models, we did so using the abstract interface and then implementing it for Matlab/Simulink. For the task at hand, the key part of it is

- *ComputeMinMaxMonotonic* (cell)
  returning the interval of minimal and maximal values at the corners of the cell (i.e. assuming monotonic functions) and using
- *ComputeOutputVariables* (inputvector)
  returning the vector of output values for the input.

The latter one is the only method that needs to be implemented for a specific computation or simulation system, apart from methods organizing the access to the tool and extracting the variables:

- *GetVariables* (direction)
  returns the list of variables for direction∈{input, output},
- *OpenSession*
  which in case of Matlab/Simulink establishes the connection to a particular Simulink block in a particular file
- CloseSession.

If a tool is able to compute the minimum and maximum values of the output variable for an input cell, then the abstraction operator can properly handle functions that are not monotonic.

4.2 Finite Constraints

For representing and further processing abstract qualitative models that are generated from the *DirectedNumerical-BehaviorDescription*, the class *FiniteConstraintBehavior-Description* (see Fig. 4) is used in MOM. A finite constraint is given by a set of variables, each with a **finite** domain assigned (i.e. a set of qualitative values), and a finite set of tuples, each of which represents one particular binding of the variables to respective values. The set of variables together with their domains is called the *signature* of the constraint.

Again, *FiniteConstraintBehaviorDescription* has been implemented as an abstract interface for defining and representing generic finite constraints in a particular constraint solver. Like the *DirectedNumericalBehavior-Description*, the interface is independent of the underlying constraint satisfaction algorithm, such that the respective system can be chosen at runtime without modifying the implementation. In the following, we describe this interface and demonstrate its usage by means of two particular constraint solvers, namely CS3 ([OCC'M, 2011]) and Toulbar2 ([Sanchez *et al.*, 2008], [Schiex, 2011]).
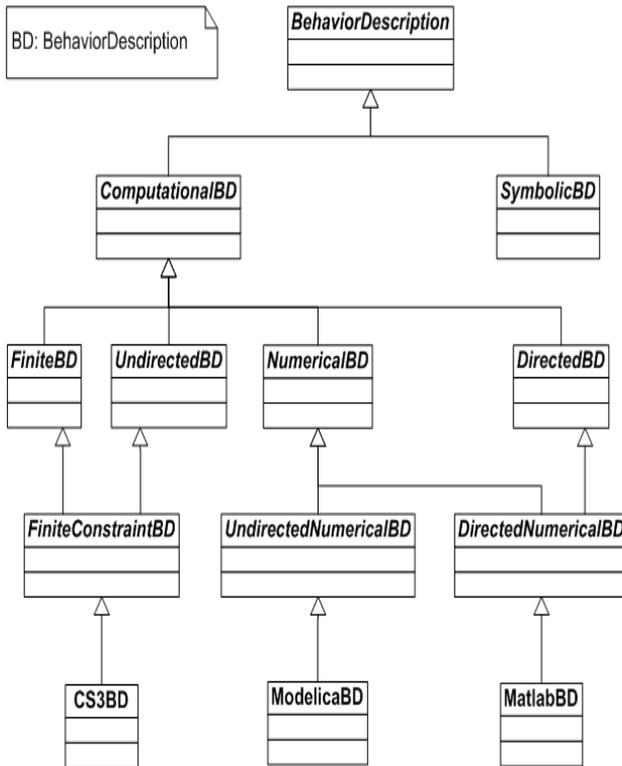
Since each constraint solving system uses its own representation of finite constraints, i.e. variables, domains and relations, an abstract interface must implement some mechanism for mapping between the distinct representations. In Toulbar2, for instance, the WCSP format ([Graphmod, 2011]) is used for encoding a constraint, which uses integer values for representing variables and domains, whereas in CS3, arbitrary strings can be used for variable names and domain values. Therefore, we employ the variable mapping algorithms that are already implemented in MOM in order to transform variables and their respective values into representations specific to the solvers.

Additionally, the procedures of how to define a constraint in the systems significantly differ and must be considered in designing the interface. CS3, for example, provides a programming interface for interactively defining variables and their domains, as well as the tuples of values, whereas Toulbar2 can only handle these definitions in batch mode. Therefore, in a Toulbar2 implementation of the interface, the tuples, the variables and the domains must be cached and handed over to Toulbar at once. Figure 5 shows an overview of the interface definition.

FiniteConstraintBehaviorDescription, as a subclass of *UndirectedBehaviorDescription* is implemented as an abstract class that defines interface methods for defining a generic constraint. Among these are

- *CreateDomain* (domain)
  creates and returns the solver-specific representation of domain (e.g. the domain {-, 0, +} becomes {1,2,3} in Toulbar2
- *CreateVariable* (variable)
  transforming the MOMvariable into the solver-specific representation of a variable (e.g. an integer in Toulbar2) and returns it.

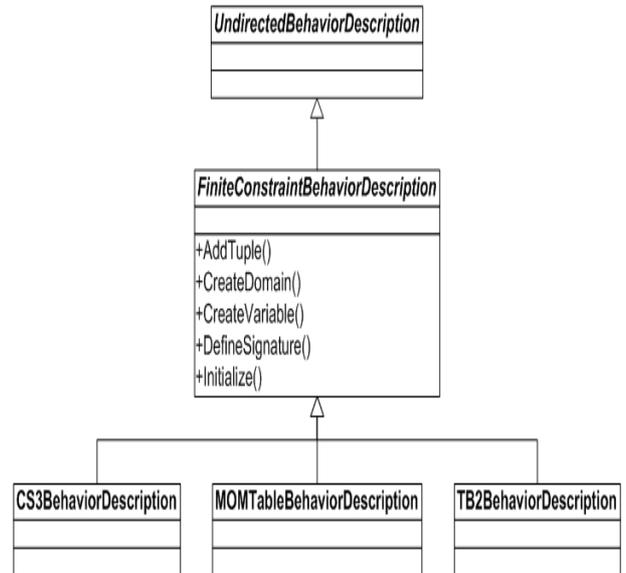Figure 4 – Various behavior description classes as interfaces

Figure 5 - Finite Constraint Behavior Description

- *DefineSignature* (variables)
  takes a set of variables as an argument and initializes a constraint's signature in the respective constraint solver. It uses the previous two methods and the variable/domain mapping facilities described in section 3.
- *AddTuples* (tuples)
  Incrementally generates the relation specifying the variable-value assignments.

We implemented three subclasses of *FiniteConstraintBehaviorDescription*, in particular

- *CS3BehaviorDescription*, which implements the interface to the CS3 constraint solver
- *TB2BehaviorDescription*, which implements the interface to the Toulbar2 solver
- *MOMTableBehaviorDescription*, which only represents the relation and can be used as an intermediate representation when the relation should be exported to various specific constraint solvers or if constraint satisfaction is not needed.

Since the *DefineSignature* and *AddTuple* methods are defined over MOM variables and domains, the three implementations are interchangeable and can be used transparently. Additionally, the *Initialize* method allows to specify a solver-specific parameter string, e.g. for specifying a path to where the solver is installed in the file system.

5   Implementing Model Abstraction in MOM

Operators in MOM are any kind of algorithms that transform or generate objects at any level from other objects. They may transform structures, such as adding or removing structural entities in an aggregate entity (e.g. dependent on whether or not wires connecting components need to be considered or not. They can generate new behavior descriptions, such as compressing a *FiniteConstraintBehaviorDescription* by projecting out irrelevant variables or eliminating elements with integration or delay from a *DirectedNumericalBehaviorDescription* which represents ordinary differential equations.

The latter is indeed the operation that has to be performed before applying the abstraction as stated at the end of section 2.

The operator *NumericDomainAbstractionDirected*, which implements the computation described in section 2, takes an existing *DirectedNumericalBehaviorDescription* as a source and generates a new *FiniteConstraintBehaviorDescription* as the target. More precisely, usually, initially there is a model frame with the numerical behavior description associated to it including the variable mapping, i.e. the left-hand side of the diagram in Figure 3. The involved domain mappings will be identities on the *RealNumberDomain* (or double) and potentially mappings of discrete domains. If one only has, say, a Matlab/Simulink model as the starting point, the respective model frame and mappings have to be constructed first, if needed.

The preparative steps are then the following:
- **Generate the targetBehaviorDescription**, a *FiniteConstraintBehaviorDescription* along with the

respective mappings, i.e. the right-hand side of Fig. 3. The domain mappings are between real numbers and qualitative domains given by the landmarks of the various variables or (identity) mappings of discrete domains.
- **DefineSignature** for this *targetBehaviorDescription*, whose defining relation has to be constructed now by the abstraction operator.

This is done by the method *GenerateRelation* (see Fig. 6). It needs the following parameters:
- **modelFrame**: as an instance of *ModelFrame*
- **sourceBehaviorDescriptionAssociation**: as an instance of *BehaviorDescriptionAssociation* to associate *modelFrame* to *sourceBehaviorDescription*; in our context, an instance of (a subclass of) *DirectedNumericalBehaviorDescription*
- **targetBehaviorDescriptionAssociation** to link *modelFrame* with *targetBehaviorDescription*;
- **modelVariables**: the list of Variables to occur in the relation as a result of the abstraction process; it can be a proper subset of the respective input/output variables of the *sourceBehaviorDescription*. If some input variables are excluded, they need to be used in the computation with their complete range, but are not included in the *targetBehaviorDescription*.

GenerateVariableMapping creates the mapping between the variables of source and target and their domains by concatenating the mappings from source to model frame and from model frame to target. Note, that such a concatenation establishes also a variable mapping and domain mappings. It can be compiled into a new one, but does not have to. *sourceInputCell* will be built recursively by *GenerateRelationRecursively,* whose pseudo code is shown in Fig. 7.

```
GenerateRelation (ModelFrame,
                  sourceBD¹Association,
                  targetBDAssociation,
                  modelVariables)
{
  variableMapping =
        modelFrame.GenerateVariableMapping
                      (sourceBDAssociation,
                      targetBDAssociation,
                      modelVariables)

  sourceBD = sourceBDAssociation.BehaviorDescription
  targetBD = targetBDAssociation.BehaviorDescription
  sourceInputCell = new ValueSet[input variable count]

  GenerateRelationRecursively (0,
                  sourceBD,
                  targetBD,
                  variableMapping,
                  sourceInputCell)
}
```

Figure 6 - GenerateRelation pseudo code

---

[1] BD: BehaviorDescription

```
GenerateRelationRecursively
                    (sourceInputVariableIndex,
                    sourceBD,
                    targetBD,
                    variableMapping,
                    sourceInputCell)
{
IF sourceInputVariableIndex == sourceInputCell.Length
           //every input value was chosen

THEN   //compute the numerical output interval, the
       respective qualitative values and return them
       together with the input values

   sourceValues = CONCAT(sourceInputCell,
         sourceBD.ComputeMinMaxMonotonic
                       (sourceInputCell) )
   tuples = GenerateTuples (sourceValues,
                       variableMapping)
   targetBD.AddTuples (tuples)

ELSE   //extends sourceInputCell

   sourceInputVariable = sourceBD.InputVariables
                    [sourceInputVariableIndex]

   //retrieve the mapping for the sourceInputVariable
   targetInputVariable = variableMapping.MapVariable
                    (sourceInputVariable)

   FOR EACH targetValue IN
               targetInputVariable.Domain.Values

     sourceInputCell[sourceInputVariableIndex] =
       variableMapping.MapValues (targetInputVariable,
                            targetValue)

     GenerateRelationRecursively
                    (sourceInputVariableIndex + 1,
                    sourceBD,
                    targetBD,
                    variableMapping,
                    sourceInputCell)
   END FOR

END IF
}
```

Figure 7 - GenerateRelationRecursively pseudo code

It iteratively maps all target values to the source domain to extend the source input cell until it is complete.

Then *ComputeMinMaxMonotonic* is invoked, and in *GenerateTuples*, the resulting intervals for the source output variables are mapped to the respective set of qualitative values. If there are several output values, the Cartesian product of these sets is constructed, and each of its tuples prefixed by the input tuple.

AddTuples extends the constraint by this set.

6   Discussion

The presentation in this paper attempted to illustrate both the facilities in MOM for representing variants of models while re-using the shared part and their utility for implementing automated model abstraction. The re-implementation of our model abstraction algorithm in this environment provides the benefit that it can be applied to whatever numerical modeling system has the source and that it can generate finite constraints in your favorite constraint solving system – provided the abstract interfaces (which are quite narrow for this task) have been implemented.

Our work on task-dependent model abstraction ([Sachenbacher and Struss, 2005]) assumed the existence of a fine-grained, but finite model relation. The abstraction operator presented here starts from a numerical model and could be used to generate a finite model to apply task-dependent abstraction.

The previous implementation of the operator has been used to generate a qualitative model for failure-modes-and-effects analysis of a simplified landing gear of an aircraft from a library of hydraulic Matlab models ([Fraracci, 2009]).

Due to the inherent limitations of model abstraction from a directed numerical computation in case of multiple qualitative output values, implementing an abstraction operator for *UndirectedNumericalBehaviorDescription* and Modelica as an instance will be challenging.

7   References

[Addanki *et al.* 1989] Addanki, S., Roberto Cremonini, J. Scott Penberthy. *Graphs of models*. Artificial Intelligence (51) 1-3, 1991, p.145-177

[Falkenhainer and Forbus, 1991] Falkenhainer, B., and Forbus, K. *Compositional modeling: Finding the right model for the job*. Artificial Intelligence, 51, 1991, p. 95-143

[Fraracci, 2009] Fraracci, A. *Model-based Failure-modes-and-effects Analysis and its Application to Aircraft Subsystems*. Dissertationen zur Künstlichen Intelligenz DISKI 326, AKA Verlag, ISBN 978-3-89838-326-4, IOS Press, ISBN 978-1-60750-081-0

[Graphmod, 2011] http://graphmod.ics.uci.edu/group/WCSP_file_format

[Nayak, 1995] Nayak, P. *Causal approximations*. Artificial Intelligence (70), Issue 1-2, 1994, p. 277 – 334

[OCC'M, 2011] OCC'M Software GmbH. *CS3 Constraint Solver*, Germany, 2011. http://www.occm.de/

[Ressencourt *et al.*, 2006] H. Ressencourt, L. Travé-Massuyès, J. Thomas, *Hierarchical modelling and diagnosis for embedded systems*, 17th International Workshop on Principles of Diagnosis DX'06, Aranda de Duero (Spain), June 26-28, 2006, pp. 235-242

[Sachenbacher and Struss, 2005] Sachenbacher, M. , Struss, P. *Task-dependent qualitative domain abstraction*, Artificial Intelligence, (162)1-2, 2005, p. 121-143

[Sanchez *et al.*, 2008] M. Sanchez, S. Bouveret, S. de Givry, F. Heras, P. Jégou, J. Larrosa, S. Ndiaye, E. Rollon, T. Schiex, C. Terrioux, G. Verfaillie, and M. Zytnicki. *Max-csp competition 2008: toulbar2 solver description*. In Proceedings of the Third International CSP Solver Competition, 2008

[Schiex, 2011] Schiex, T. *ToulBar2 an open source weighted constraint satisfaction solver*, France, 2011. http://mulcyber.toulouse.inra.fr/projects/toulbar2/

[Struss, 1992] Struss, P., *What's in SD? Towards a Theory of Modeling for Diagnosis*. In Hamscher et al. (eds.), Readings in Model-based Diagnosis. Morgan Kaufmann Publishers, 1992, p. 419-450

[Struss, 2002] Struss, P., *Automated abstraction of numerical simulation models – theory and practical experience*. In Sixteenth International Workshop on Qualitative Reasoning, Sitges, Catalonia, Spain, 2002

[Struss and Regassa, 2010] Struss, P., Regassa, T. *MOM – An Environment for Multiple Modeling*. In 24th International Workshop on Qualitative Reasoning, Portland, USA, 2010

[Torta and Torasso, 2009] Torta, G., Torasso, P. *Parametric abstraction of behavioral modes for model-based diagnosis*. AI Communications (22) 2, 2009, p.73-96

[Weld, 1992] Weld, D. *Reasoning about model accuracy*, Artificial Intelligence (56) 2-3, 1992, p. 255-300