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
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], [Rensencourt 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 account for the fact that every numerical model is a function of continuous functions, the output variable is given as a function of the input variables, \( y = f(x_1, ..., x_n) \).

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: \( q_{ij} := (l_{ij}, l_{ij+1}) \).

For each tuple of qualitative input values, \( (q_{ij}, ..., q_{nk}) \), 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:

\[
\text{Corners}(\text{cell}) = \text{Corners}(q_{ij}, ..., q_{nk}) := \{ (l_{ij}, ..., l_{nk}) | q_{ij} := (l_{ij}, l_{ij+1}) \land k \in \{j, j+1\} \}.
\]

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:

- \( f^{-\varepsilon, \varepsilon} \) , \( f + \varepsilon \) continuous for each \( x_i \),
- \( \{q_{ij}\} \) the qualitative values for \( x_i \),
- \( \{q_{ij}\} \) the qualitative values for \( y \),
- \( \{q_{ij}\} \) the qualitative values for \( x_i \),
- \( \{q_{ij}\} \) the qualitative values for \( y \),
- \( \{q_{ij}\} \) the qualitative values for \( x_i \).

For each input cell, \( cell \in Cells \), the range of the output \( y \) is

\[
\text{int}(cell) := \{ \text{Cells} | x \in cell \},
\]

\[
\max \{|f^{\varepsilon,\varepsilon}(x_i)| | x \in cell \}
\]

Then

\[
R_{\text{abst}}(f, \varepsilon, \varepsilon) := \{(q_{ij}, ..., q_{nk}) \mid \text{int}(cell) \neq \emptyset \}
\]

is the qualitative abstraction of \( R_d(f, \varepsilon, \varepsilon) \).

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.

\[
R_d(f, \varepsilon, \varepsilon) \subseteq R_{\text{abst}}(f, \varepsilon, \varepsilon).
\]
Lemma 2.2 (Minimality)
The qualitative abstraction is minimal, i.e. any proper subset of $R_{abstr}(f, e, e')$ is not an abstraction of $R_d(f, e, e')$.

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 at the corners in order to obtain the abstraction

Lemma 2.3 (Abstraction of Monotonic Functions)
If $f(ad, e'), f(a+e')$ are monotonic in each $x_i$, then
\[
\int_{v(e', i)} \left( \{ f(x) \mid \text{cor} \in \text{Corners}(cell) \} \right) = \max \left( \{ f(x) \mid \text{cor} \in \text{Corners}(cell) \} \right)
\]

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 \( y = (y_1, y_2) = f(x_1, \ldots, x_n) \),
\[ y_1 = f_1(x_1, \ldots, x_n), \]
\[ y_2 = f_2(x_1, \ldots, x_n). \]

As the abstraction of $f$, we compute
\[
R_{abstr}(f, e, e') = R_{abstr}(f_1, e, e') \otimes R_{abstr}(f_2, e, e')
\]
where $R_{abstr}(f, e, e')$ 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](image)

- **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 reused. 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).

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 **UndirectedNumericalBehaviorDescription**, 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 \( \in \{\text{input, output}\} \),
- **OpenSession** which in case of Matlab/Simulink establishes the connection to a particular Simulink block in a particular file
- **CloseSession**.

![Figure 3 - Association between ModelFrame and two Behavior Descriptions and the related VariableMappings and DomainMappings](image-url)
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 DirectedNumericalBehaviorDescription, the class FiniteConstraintBehaviorDescription (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 DirectedNumericalBehaviorDescription, 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.
• 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 FiniteConstraint-
BehaviorDescription, 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 preparatory steps are then the following:
• Generate the targetBehaviorDescription, a Finite-
  ConstraintBehaviorDescription 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;
  • 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 Generate-
RelationRecursively, whose pseudo code is shown in Fig. 7.

```
GenerateRelation (ModelFrame, sourceBD Association, targetBD Association, modelVariables)
{
  variableMapping =
  modelFrame.GenerateVariableMapping
  (sourceBD Association, targetBD Association, modelVariables)
}

sourceBD = sourceBD Association.BehaviorDescription
targetBD = targetBD Association.BehaviorDescription
sourceInputCell = new ValueSet{input variable count}

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

Figure 6 - GenerateRelation pseudo code

1 BD: BehaviorDescription
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


