ModelML: a Markup Language for Automatic Model Synthesis

Cong Zhang Amol Bakshi Viktor K. Prasanna

Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, California
Email: {congzhan, amol, prasanna}@usc.edu

Abstract

Domain-specific modeling has become a popular way of designing and developing systems. It generally involves a systematic use of a set of object-oriented models to represent various facets of a domain. However, manually creating instances of these models is time-consuming and error-prone when a system in the domain is complex. Automatic model synthesis tools are thus usually developed to free users from the model creation process. In practice, most of these tools would hard code knowledge about the domain specific models in the program. A biggest problem with tools is that their source code needs to be changed whenever the knowledge changes. In this paper, we define a model markup language (ModelML) to facilitate the development of automatic model synthesis tools. The language provides a complete self-describing representation of object-oriented models to be synthesized. Unlike other XML-based representations of models, ModelML reflects the structure of the models directly in the nesting of elements in the XML-based syntax. This feature allows the knowledge about the domain specific models to be decoupled from model synthesis tools. To demonstrate the usefulness of the markup language, we have developed a generic automatic model synthesis tool which is based on ModelML inputs.

1 Introduction

Domain-specific modeling is being widely discussed and explored in various domains. In software engineering domain, it is considered as a way of increasing the quality and efficiency of large-scale software development [4]. In embedded system design, domain-specific models that capture the structural and behavioral aspects of embedded systems are used to drive various simulators in a unified simulation framework [1]. Generally, the modeling constructs for describing various facets of a system are defined using object-oriented class modeling approach. Most research on domain-specific modeling is based on the instances of these modeling constructs, which are the focus of this work and referred to as *models* or *model elements* throughout this paper. Specifically, these models are converted into other formats like executable code, inputs to some analysis tool, or configuration files for simulators. The conversion of models to a more useful form is called model interpretation and is performed by model interpreters [12].

Modeling paradigm is a concept used in domain-specific modeling. A modeling paradigm contains all the syntactic, semantic, and presentation information about a domain, e.g., which concepts will be used to construct models, what relationships may exist among those concepts, how the concepts may be organized and viewed by the modeler, and rules governing the construction of models [7]. A modeling paradigm defines the family of models that can be created using a domain-specific modeling environment. Generally, modeling paradigms are developed based on an exhaustive characterization of the underlying domain, and are used by the designer to instantiate domain-specific models.

The work described in this paper is part of the Integrated Asset Management (IAM) project at the Chevron-funded Center for Interactive Smart Oilfield Technologies at the University of Southern California, Los Angeles [3]. The current focus of the IAM project is on enabling model-driven reservoir management. In model driven reservoir management, the reservoir engineer relies on simulations (and hence simulation models) to make key operational decisions pertaining to the reservoir on a day-to-day basis.

In [15], we designed and implemented a prototype toolkit and the modeling paradigm for IAM. One major step in using our toolkit is asset modeling. At the type level, the model elements in an oilfield asset model are classified into physical and non-physical components. Physical components include wells, reservoir volume elements, separators, compressors, etc. Non-physical components include production controls, field constraints, drilling schedules, among others. At the instance level, the asset modeling involves instantiation of the model elements that represent the structure and properties of an asset. For small assets—with, say, tens of wells—this model instantiation can be performed manually, although the effort involved is not insignificant. For larger assets, manual model instantiation is time-consuming and error-prone. Also, most of the information about the asset model exists in legacy data stored in MS Excel files, text files, databases, etc., obviating the need for manual entry. Therefore, we developed a prototype automatic model synthesis tool that reads legacy data (in a specific format) and automatically creates the suitable entities in the modeling environment. Because the tool is customized for a modeling language created specifically for our oilfield applications, it has to be modified and recompiled whenever there is a change in the modeling language. In this paper, we discuss the design and implementation of a generic, domain-independent approach for automatic model synthesis based on an XML-based model description language, which is strong enough in terms of expressiveness for solving the model synthesis problem.

The paper is organized as follows. In Section 2, we provide an high level view of object-oriented model synthesis process. Challenges in developing a generic automatic model synthesis tool are also identified. Sections 3 provides details about the markup language. A generic automatic model synthesis tool which is based on ModelML is described in Section 4. Section 5 discusses related efforts and we conclude in Section 6.

2 Synthesis of Object-Oriented Model

In this work and in [15], we use the Generic Modeling Environment (GME) [7] to synthesize a domain-specific modeling interface for the models defined in the previous section. GME provides a graphical meta-modeling language, which is called MetaGME and based on the UML class diagram notation and OCL constraints, to formally define the modeling paradigm. GME then automatically generate a domain-specific modeling environment based on the modeling paradigm. The generated domain-specific modeling environment, which has similar look and feel as GME, is used by end users to build domain models. In our work, the domain is a generic oilfied domain. The modeling paradigm consists of a set of building blocks and composition rules, which are used to describe all the physical and non-physical model information acting as input to workflows in Integrated Asset Management.

There are two major operations in creating an object-oriented model in GME: creating the new model and setting up its model attributes. For illustration purpose, we adapt the APIs for the two operations in GME into functions of the following form: `CreateNewModel` and `SetModelAttribute`. Function `CreateNewModel` takes at least two arguments: the parent model where the new model is created, and the kind of model to be created. It returns a handle of newly created model. If the model to be created is the root in the model hierarchy, a `Null` handle is used. `SetModelAttribute` is to set a model's attribute with some value. It takes three parameters: the model whose attribute is set, name of a model attribute, and value used to set the model attribute.

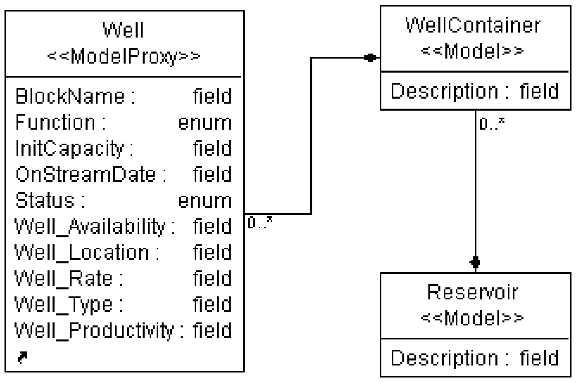


Figure 1. Model Paradigm Fragment

Figure 1 shows a fragment of a modeling paradigm used in Integrated Asset Management. There are three kinds of model: *Reservoir*, *WellContainer*, and *Well*. Suppose a well whose *InitCapacity* is 5000 is to be created. The following illustrates how to programmatically synthesize the models using the two functions:

```
rModel = CreateNewModel(Null, "Reservoir")
cModel = CreateNewModel(rModel, "WellContainer")
wModel = CreateNewModel(cModel, "Well")
SetModelAttribute(wModel, "InitCapacity", 5000)
```

In practice, however, the code for synthesizing object-oriented models is more complex. First of all, the modeling paradigm contains many more modeling concepts. To create a new model at the right "place", in terms of model hierarchy, the model's ancestors have to be created. More modeling concepts in the hierarchy result in more code. Secondly, the data used to set model attributes would not be hard coded in the source code. It usually comes from some data source, such as a relational database, or data files on a local disk. Data access code is required by the synthesis tools.

To make valid calls to the two functions, we have to be aware of the underlying modeling paradigm. Because the models being created and attributes being set need to conform to the modeling paradigm. Specifically, for the foregoing example, we need to know: (1) names of various model kinds, such as *Reservoir*, *WellContainer*, and *Well*; (2) relationships among various models, e.g., a *Reservoir* contains

WellContainer; (3) names of model attributes, such as *Init-Capacity*.

A straightforward approach to implementing a model synthesis tool is to hard code the modeling paradigm in the tool, as we did in the foregoing example. The knowledge about modeling paradigm is completely coupled with the logic of a model synthesis tool, scattered over the source code where `CreateNewModel` and `SetModelAttribute` are used.

However, in real life, the modeling paradigm for a domain might evolve over time. There are several reasons for this. One of the most important ones is the demand for continuous system improvement, which leads to a continuous adaptation of the corresponding domain modeling. Another reason is caused by the customization of a model paradigm to the needs of a specific case, e.g., to support a new workflow . Therefore, a model kind could be added, removed, or changed as the modeling paradigm evolves, even though major model kinds and relationships may have become stable.

Modeling paradigm evolution causes problems for application development in the domain. One of the biggest problems is that any change to the modeling paradigm breaks the tools that are tightly coupled with it, e.g., the automatic model synthesis tool implemented in the straightforward way. Therefore, **a challenge is to decouple the knowledge about the modeling paradigm from the tools that belong to the domain**. In this paper, we only concentrate on automatic model synthesis tools.

3 ModelML

To meet the challenge described in Section 2, we propose a model description language, ModelML. An automatic model synthesis program can be divided into two parts: (1) the model description, and (2) the code that implements the synthesis logic. The objective of ModelML is to create separation of the two parts. There are two reasons for drawing a clear line to distinguish the two parts. First, the model description is modeling paradigm dependent. It contains the data used by the synthesis logic. It should be made an input to model synthesis program, instead of an integral part of the program. The second reason is to allow a many-to-one relationship between the two. One may want multiple model descriptions to the same program logic. Or one may want a model description to control multiple synthesis tools, each with distinct internal program logic. For example, one model description might be used in two different synthesis tools.

As we define ModelML, we are aware of other languages or approaches that can also address the challenge. For example, Meta-Object Facility (MOF) [10] is a set of standard interfaces that can be used to define and manipulate a set of interoperable meta-models and their corresponding models. The MOF model can be used as a model for defining metamodels such as the UML and metamodels used by GME. It is possible to write a generic MOF code to manipulate these metamodels. This generic code is not statically developed for each metamodel. All APIs generated from metamodels inherit from a fixed set of interfaces called the MOF Reflective Interfaces. The Reflective Interfaces have all the functionality that the generated, metamodel-specific interfaces have, although they are less convenient to use since their signatures are not tailored to the specific metamodels. Therefore, automatic model synthesis tools could be developed similarly using MOF Reflective Interface. We are open to this approach. As of now, we use ModelML because it is simple and works well for our purpose.

3.1 Overview of ModelML

ModelML is an XML-based model markup language. It is designed for specifying hierarchical models for facilitating application development of automatic model synthesis. Because ModelML is XML-based, it is easy to parse, and all existing tools that work with XML can be applied to models in its ModelML representation.

The key features of ModelML include:

- *High level data representation* ModelML is working at semantic level, not syntax level. Part of the modeling paradigm, which will be used by automatic model synthesis tools, is embedded in the model description in ModelML. This way, information about the modeling paradigm is decoupled from the model synthesis process.

- *Support for visual rendering* ModelML allows specifications of rendering information for a visual rendering tool. For example, ModelML elements can specify a location and can reference an external configuration file that defines a visual rendition, such as an icon.

- *Extensibility* Names of elements, attributes, or keywords are either from a modeling paradigm or designed in such a way that they do not imply any underlying platfrom.

3.2 Specification of Object-Oriented Models

ModelML describe models with two sections: *structure*, and *style*. The structure section is required and should not be empty, while the style section is optional.

A typical model description in ModelML has the following skeleton:

```
<?xml version="1.0" standalone="yes"?>
<ModelML
 xmlns="http://cisoft.usc.edu/2007/IAM">
```

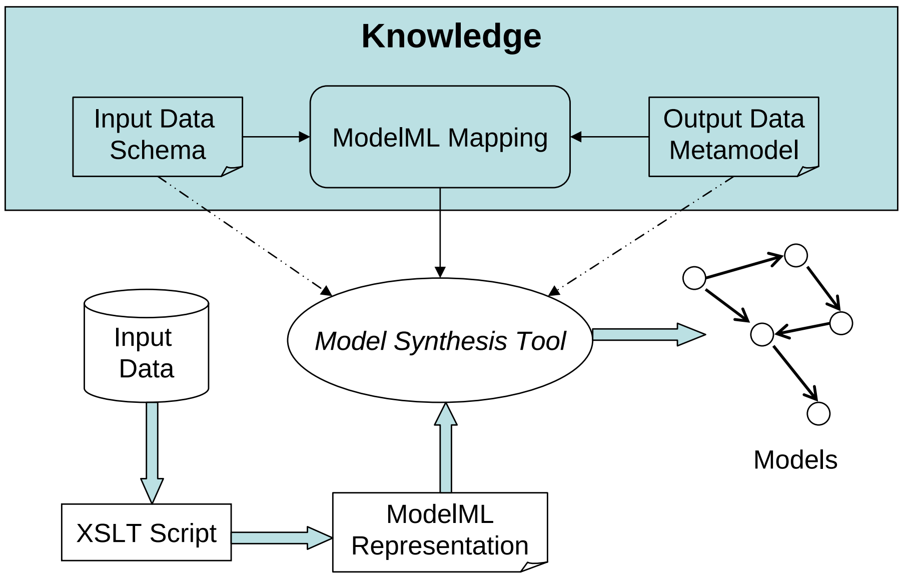


Figure 2. ModelML based model synthesis

```
  <structure>...</structure>
  <style>...</style>
</ModelML>
```

The <structure> section lists all models to be created, specifying model attribute values as well as how models are organized. Each element in the structure section has a name and one or more attributes. Attributes correspond directly to model properties, and the name of the ModelML element exactly matches the model kind name. The ModelML representation reflects the structure of the models to be synthesized in the nesting of the elements. For example, *Well* is a child of *WellContainer*, thus *Well* element is nested inside the *WellContainer* element. This nesting is even more apparent when presented visually as in the following.

```
<structure>
 <Reservoir
  <WellContainer>
   <Well name="ABC1" BlockName="Asset1"
     Function="W" InitCapacity="3000"
     OnStreamDate="2004-12-07"/>
   <Well name="ABC2" BlockName="Asset1"
     Function="W" InitCapacity="3000"
     OnStreamDate="2004-12-27"/>
   <Well name="ABC3" BlockName="Asset1"
     Function="O" InitCapacity="4500"
     OnStreamDate="2005-11-07"/>
  </WellContainer>
 </Reservoir>
</structure>
```

The <style> section contains visual rendering information for models in the structure section. This section is optional and platform-specific. Different modeling platforms need different style specification. Each line in the style section describes a set of model elements that are specified with style attributes. Various style attributes can be specified, such as size, layout, etc. Elements in the style section, which are selected from the list in the structure section, are identified by the name attribute. If the name attribute is missing for an element in the style section, the style will be applied to all the model elements of the same kind.

```
<style>
 <Reservoir name="" type="Size" value="100, 100"/>
 <Well name="ABC1"  type="Position" value="100,10"/>
</style>
```

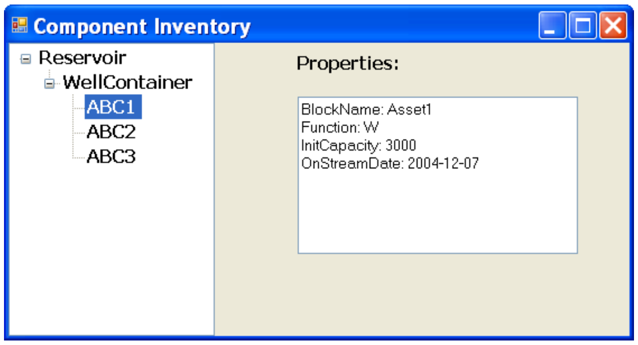
4 ModelML-based Automatic Model Synthesis Tool

In Section 1, we briefly described an automatic model synthesis tool which was specially developed as a part of IAM toolkit. Having designed ModelML, we completely re-implemented the tool. The new automatic model synthesis tool takes as an input a model description in ModelML. In terms of how a model description is created from some legacy data, it is not a trivial problem. By design, a model description in ModelML can be specified manually or programmatically, given data for the models to be synthesized. When the data is very large in size, it is not realistic to read the data and create every ModelML manually. In our current solution to the problem, we assume that data for model synthesis is stored in an XML file. At the end of this section, we will show an XSLT script that transforms legacy data into a model description in ModelML. Figure 2 shows our design of ModelML based model synthesis.

The new automatic model synthesis tool is paradigm-independent. It essentially implements several rules that guide the model synthesis process. The rules, summarized below, are specific to the syntax of ModelML.

- Each element in the structure section of a ModelML is transformed into a model.
- Each attribute of a model element listed in the structure section is transformed into a model attribute.
- The *name* attribute of a model element in the structure section is used to set the name of a model. If the attribute is empty, the name of the model element is used.
- Each element in the style section of a ModelML is applied to the model(s) that it implies.

To show the advantage of decouping model description by using ModelML, we have created a Windows Form application which displays information on models described in ModelML. Data in the style section is ignored by the Windows application. In this Figure 3 is a screenshot of the application. With the advent of XAML and Windows Presentation Foundation (WPF) [13], ModelML could become much easier to drive a Windows Form application. For example, through an XSLT script, a ModelML description can be transformed into an XAML specification.



BlockName: Asset1
Function: W
InitCapacity: 3000
OnStreamDate: 2004-12-07

Figure 3. Windows Form rendition of ModelML data

In our prototype toolkit for IAM [15], the data set that describes an oilfield asset is stored in XML files. Therefore, we use XSL to transform XML data sources into ModelML representation. The following is a snippet of one XML file which is used as the data source for the transformation.

```
<DataSet>
 <Well>
  <Name>ABC1</Name>
   <Function>W</Function>
   <InitCapacity>3000</InitCapacity>
   <OnstreamDate>2004-12-07</OnstreamDate>
   <BlockName>Asset1</BlockName>
 </Well>
...
</DataSet>
```

Below is the XSLT script which extracts the *Well* elements in the XML file, converts them into the model elements in ModelML, and places them in the right place of the model hierarchy.

```
<xsl:output method="xml" version="1.0"
          encoding="UTF-8" indent="yes" />

<xsl:template match="/DataSet">
<structure>
 <Reservoir>
  <WellContainer>
  <xsl:for-each select="Well">
   <xsl:element name="Well">
    <xsl:attribute name="name">
     <xsl:value-of select="Name"/>
    </xsl:attribute>
    <xsl:attribute name="BlockName">
     <xsl:value-of select="BlockName"/>
    </xsl:attribute>
    <xsl:attribute name="Function">
     <xsl:value-of select="Function"/>
    </xsl:attribute>
    <xsl:attribute name="InitCapacity">
     <xsl:value-of select="InitCapacity"/>
    </xsl:attribute>
    <xsl:attribute name="OnStreamDate">
     <xsl:value-of select="OnstreamDate"/>
    </xsl:attribute>
   </xsl:element>
  </xsl:for-each>
  </WellContainer>
 </Reservoir>
</structure>
<style>
...
</style>
</xsl:template>
```

5 Related Work

To the best of our knowledge, our work is the first effort to address the challenges in facilitating application development for automatic model synthesis for a domain specific modeling environment. However, from a broader perspective, automatice model synthesis can be considered as a special case of model transformation, where there has been significant research effort. The Institute for Software Integrated Systems (ISIS) at Vanderbilt University [8] has developed a UML-based approach for specifying model transformation [9]. UML class diagrams are used to represent the graph grammars of the input and the output of the transformations. The input is a set of models created in a Domain Specific Design Environment (DSDE). The output is models in other formats like executable code, inputs to some analysis tool, or configuration files for simulators. Eclipse Modeling Project (EMP) [6] is a huge research effort in the field of model engineering. One of the major componets of the project is Eclipse Epsilon [5], Extensible Platform for

Specification of Integrated Languages for mOdel maNagement. The purpose of Epsilon is to provide an integrated set of languages for the most common model management tasks such as model transformation, comparison, merging, validation etc. However, our work is different from these research efforts. The objective of our work is to convert legacy data into large amount of domain specific models, while the other two research projects are studying issues beyond that, with the assumption that models have been created in some way.

From another perspective, the model description language proposed in this work is similar to many other user interface markup languages, e.g., MXML [2] by Macromedia, Extensible Application Markup Language (XAML) [14] by Microsoft, Scalable Vector Graphics(SVG) [11] by World Wide Web Consortium, etc. Take XAML as an example. It is used in Windows Presentation Foundation (WPF), as a markup language to define user interface elements, data binding, events, etc. XAML elements are mapped to Common Language Runtime (CLR) object instances and XAML attributes are mapped to CLR properties and events on those objects. In our work, each ModelML element in the structure section is mapped to a model kind in a domain specific modeling language, while attributes of the ModelML element mapped to model attributes. The major different between ModelML and other user interface markup languages is that ModelML works with a variety of domains, each has different objects and properties, while a user interface markup is specific to a UI technology and its language elements are well defined and stable.

6 Concluding Remarks

ModelML is a simple but powerful way of building trees of domain specific model objects. Because it is based on XML, it is straightforward to create ModelML-based markup. This not only makes it easy to create model descriptions by hand, it also makes it straightforward for tools to generate ModelML. We have shown that it is easy to use technologies such as XSLT to transform XML data sources into ModelML documents. ModelML enables a clean separation of knowledge about underlying modeling paradigm from source code of automatic model synthesis. We have also shown that ModelML can be further utilized as data source for other rendering tools, such as Windows Form.

Acknowledgment

This research was funded by CiSoft, (Center for Interactive Smart Oilfield Technologies), a Center of Research Excellence & Academic Training and a joint venture between the University of Southern California & Chevron. We are grateful to the management of CiSoft and Chevron for permission to present this work.

References

[1] A. Agrawal, A. Bakshi, J. Davis, B. Eames, A. Ledeczi, S. Mohanty, V. Mathur, S. Neema, G. Nordstrom, V. Prasanna, C. Raghavendra, and M. Singh. Milan: A model based integrated simulation framework for design of embedded systems. In *Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2001)*, Snowbird, Utah, June 2001.

[2] An overview of MXML: The Flex markup language. `http://www.adobe.com/devnet/flex/articles/paradigm.html`.

[3] CiSoft: Center for Interactive Smart Oilfield Technologies. `http://cisoft.usc.edu`.

[4] Domain Specific Language Tools. `http://msdn.microsoft.com/vstudio/DSLTools`.

[5] Eclipse Epsilon. `http://www.eclipse.org/gmt/epsilon`.

[6] Eclipse Modeling Project. `http://www.eclipse.org/modeling`.

[7] GME: Generic Modeling Environment. `http://www.isis.vanderbilt.edu/Projects/gme`.

[8] Institute for Software Integrated Systems, Vanderbilt University. `http://www.isis.vanderbilt.edu`.

[9] G. Karsai, A. Agrawal, and F. Shi. On the use of graph transformations for the formal specification of model interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, November 2003.

[10] Meta-Object Facility (MOF). `http://www.omg.org/mof`.

[11] SVG: Scalable Vector Graphics. `http://www.w3.org/Graphics/SVG/`.

[12] J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4):110–112, April 1997.

[13] Windows Presentation Foundation. `http://wpf.netfx3.com`.

[14] XAML: Extensible Application Markup Language. `http://www.xaml.net`.

[15] C. Zhang, A. Orangi, A. Bakshi, W. D. Sie, and V. K. Prasanna. Model-based framework for oil production forecasting and optimization: A case study in integrated asset management. In *SPE Intelligent Energy Conference and Exhibition*, April 2006.