SIMPLIFIED INFRASTRUCTURE FOR THE TRANSFORMATION OF XML MODELS

N. N. Glibovets and V. M. Fedorchenko

An approach is proposed to the construction of a lightweight infrastructure for the model driven development of complex software systems. It is based on the use of a domain-dependent XML format for primary models. A model has a compact representation, allows using advanced tools for changing and extending it, and makes it possible to easily determine transformations for converting domain-dependent XML models into any other models. The use of XLS to describe transformations allows organizing both vertical and horizontal transformations (with any number of abstraction layers).

Keywords: *model driven development, domain-dependent model, model transformation, IoC-container.*

INTRODUCTION

Programming, as a method to formalize the description of various processes, depends on specific technical constraints and on the infrastructure within which the formalization is performed. Due to the progress in this field, for many types of problems, the priority in programming was shifted from the most efficient (in the context of technical resources involved) coding to the fastest (in the context of human resources) and at the same time qualitative development [1–4].

MODEL DRIVEN DEVELOPMENT

The key concepts of model driven development (MDD) are the domain-dependent language of modeling and model transformation [5, 6]. In the context of MDD, a model is an abstraction of a software system or/and environment over its software implementation. For example, according to this definition, a program code is also a model since it is an abstraction over the machine code generated by a compiler. A metamodel describes the syntax and semantics of a DSM language in terms of the definition of concepts and their relations in a specific subject domain (the process of creating a DSML is called metamodeling).

Model transformation plays a key role in MDD and is responsible for the transformation of models defined by DSML into other program artefacts and forms of representation. For example, a model can be transformed into a combination of original texts, resources, and XML configurations (since, in the context of MDA, a program in any form is actually a model). The OMG (Object Management Group) consortium has proposed its own modification of such a technology called model driven architecture (MDA) as a series of specifications and standards [7].

The currently available approaches to the transformation of input models are conditionally divided into (i) those generating the original model as a text (usually being a program code in a certain general-purpose programming language, for example, Java, C++) and (ii) those generating the original model in a more structured form (for example, XML), which corresponds to a metamodel. Since a program code is also a model, the approaches differ only in how the original metamodel is used. If a text is generated, the transformer does not necessarily operate with the structure of the metamodel of the programming language (a by-effect of such a simplification is that an incorrect code may be generated).

The practice of determining transformers that combine several approaches is rather popular. A typical example is the QVT standard, which can be used to determine transformations in MDA [7]. Another popular and powerful transformer is the implementation of the XSLT standard since any model can be represented as XML using XMI format (XML metadata interchange [8]). However, there is a significant shortcoming: the XMI format is rather unwieldy (for the transformation developer), which results in XSL-rules that are complex to be developed and supported .

For MDD to be successfully introduced in software development, an infrastructure is necessary that would support the model-driven development and meet the following requirements: possible flexible manipulation in life cycle parameters, determination and expansion of models by demand, possible simultaneous use of models of different levels of abstraction, integration with available program systems, minimized expenses on the integration and support of the infrastructure for MDD.

SIMPLIFIED INFRASTRUCTURE

Let us consider a simplified infrastructure for a model driven development that meets the above requirements. The essence is to return to the original representation of the model as a domain-dependent XML format, which will allow getting rid of the shortcomings of the XMI+XSL version. The model will have the maximum compact representation and can be read and modified by programmers probably without using any specialized tools (editors, visual analyzers). This property becomes of primary importance in the cases where alterations are introduced in the domain-dependent metamodel during the development. It remains possible to rapidly change and expand the metamodel, and if the domain-dependent metamodel is stabilized, to determine transformations of the created domain-dependent models into any other (standardized) models (for example, UML).

In practice, it is admissible to reduce the definition of a metamodel to creating the XML schema (XSD) of the domain-dependent model. This language is well-known to programmers and has a tool support in any modern platform. A practically comprehensible level of model validation with minimum time expenses for the metamodel expansion can be provided. Naturally, XSD has restricted capabilities to describe a metamodel; however, if the infrastructure is simplified as much as possible, this case is a comprehensible compromise since own metadata can be added to the XML schema if necessary. If this is also insufficient, metamodels can be supplemented by using the MOF (MetaObject Facility [7]) standard since this specification allows defining any metamodel.

In this case, using the XSL to describe transformations seems to be the most acceptable since this language is a powerful tool (contains a Turing-complete set of functions), especially if the target model can also be represented in the XML format. This will allow organizing both vertical transformations with an arbitrary number of abstraction layers and horizontal ones. As this language is widely popular and commonly known, necessary conditions for a teamwork are created, where a metamodel is defined and simultaneously models based on this metamodel are created by one and the same group of developers. This factor is extremely important for the modern development where the time interval between the beginning of development and the first results should be minimized.

The last stage in forming the simplified infrastructure for model driven development is determining the lowest-level model format accessible within the framework of the infrastructure. As indicated above, it is expedient for this model to have natural representation in the XML format, to be sufficiently simple for a trivial transformation into a machine code, and at the same time to be convenient enough to express concepts from models of higher levels of abstraction (though these two conditions are somewhat contradictory).

Recently, IoC-containers whose configuration usually has XML-representation (SpringFramework [9], Winter.NET [10]) are widely used. IoC-containers implement the “inversion of control” pattern of programming also known as the “dependency injection,” which has become especially popular due to the paper by Martin Fowler [11]. The essence of this pattern is abstracting the creation and initialization of one objects by other ones by delegating certain actions to special types of objects (IoC-containers). All the other objects just declare the dependences minimum necessary for functioning (through interfaces). IoC-containers also create the graph of objects and, respectively, determine the dependences as specific copies of objects that implement the necessary interfaces. In the context of MDD, this specificity provides the unique possibility of satisfying both conditions for the lowest-level model since the way the objects are implemented is fixed (and extremely simple), and the objects can implement rather complex concepts. Creating such a graph based on an XML configuration is a fast and trivial problem [9, 10].

In terms of MDA, an IoC configuration is a PSM (Platform Specific Model) dependent on the lowest-level platform (since it includes references to classes and properties of a quite specific platform), and domain-dependent XML models (if

have no binding to features of the platform) pertain to PIM. Naturally, this is a rather conventional division. The proposed alternate infrastructure can adjust the PSM abstraction degree depending on the situation.

EXAMPLE OF CREATING A SIMPLIFIED MDD INFRASTRUCTURE

To illustrate the described approach, let us consider creating a simplified MDD infrastructure to transform models in the automation of business processes.

We may borrow the definitions of key concepts for such a model from numerous enterprise ontologies, for example, CEO (Core Enterprise Ontology [12]) as one of the most simple ones. The transformation results in the Winter configuration of an IoC-container [10] to be applied on the Microsoft .NET platform.

The CEO is intended to describe business processes and to model as an information system (IS). In a simplified form (minimum necessary to construct a metamodel), this ontology consists of the following basic concepts [12].

- Passive entities represent business objects, passive elements of a corporate medium that are created, varied, and revised in order to obtain any information. An important property of passive entities is identifiability, which is a necessary condition to include passive entities to determine the data schemes of information systems.
- Active entities are active elements from the corporate medium that can make decisions and initiate certain actions. They may be both people and organizations (office, department, section). It is with respect to active entities duties, access rights, etc. are specified.
- Transformations are any actions, operations, and processes that exist in a corporate environment and are reflected in the IS. Usually, both passive and active entities take part in transformations.
- Conditions are predicates that can be calculated in order to determine their validity. This basic concept underlies such concepts as a purpose, a rule, a constraint, and a state.

A detailed analysis of CEOs goes beyond the scope of the paper. A metamodel based on these concepts allows describing various business processes. Note that even the most simple description of such concepts is rather bulky; therefore, to illustrate a simplified approach to model transformation, we will detail only the definition of an entity.

Let us describe a metamodel that can be represented as an XML scheme within the framework of a simplified infrastructure:

```
<xsd:element  name="entity">
  <xsd:complexType>
    <xsd:sequence  maxOccurs="unbounded">
      <xsd:element  name="storage">
        <xsd:complexType>
          <xsd:sequence  maxOccurs="unbounded">
            <xsd:element  name="sourcename"  type="xsd:string"  minOccurs="1"/>
          </xsd:sequence>
          <xsd:attribute  name="versions"  type="xsd:boolean"  use="optional"
default="false"  />
        </xsd:complexType>
      </xsd:element>
      <xsd:element  name="schema">
        <xsd:complexType>
          <xsd:sequence  maxOccurs="unbounded">
            <xsd:element  name="field"  minOccurs="1">
              <xsd:complexType>
                <xsd:attribute  name="uid"  type="xsd:boolean"  use="optional"
default="false"  />
                <xsd:attribute  name="name"  type="xsd:string"  use="required"/>
                <xsd:attribute  name="type"  type="xsd:string"  use="required"/>
                <xsd:attribute  name="nullable"  type="xsd:boolean"  use="optional"
default="false"  />
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute  name="name"  type="xsd:string"use="required"/>
  </xsd:complexType>
</xsd:element>
```

Based on this metamodel, the model of a certain entity, for example, “order,” will have the following form:

```
<entity  name="purchaseOrder">
  <storage  versions="true">
```

```
    <sourcename>purchase_orders</sourcename>
  </storage>
  <schema>
    <field  uid="true"  name="id"  type="int"/>
    <field  name="contact_id"  type="int"/>
    <field  name="sum"  type="decimal"/>
    <field  name="delta"  type="decimal"/>
    <field  name="expired_date"  type="dateTime"  nullable="true"/>
  </schema>
</entity>
```

Thus, the information system is supplemented with one more passive entity “purchaseOrder” with the characteristics “id” (a unique identifier), “contact_id,” “sum,” “delta,” and “expired_date.” These characteristics should be stored in a data storehouse (for example, a database) as “purchase_orders.” This entity, by definition, should also have the possibility to be stored, modified, and loaded from the data storehouse. Moreover, the marking versions=“true” obliges the IS to store all the versions of this entity that were stored in the data storehouse.

To determine the transformation of this model to the configuration of components of the IoC-container, it is necessary to have the possibility to represent the concept of a passive entity with the above set of properties using one or a composition of several objects. Let there exist a class where, according to the “Active Record” pattern [13], the properties of storage, loading, and modification of the record in data storehouse are implemented (C#):

```
public  interface  IActiveRecord  {
       string  XmlSchema  {get;  set;}
       string  SourceName  {get;  set;}
       bool  VersionsEnabled  {get;  set;}
       bool  IsPersisted  {get;}
       object[]  Uid  {get;  set;}
       object  this[string  fieldName]  {get;  set;}
       void  Save();
       void  Load(object[]  uid);
       void  Delete();
       void  Disconnect();
}
```

Let there also be a DbActiveRecord class (implementing the IActiveRecord interface) such as its functioning needs XmlSchema, SourceName, and VersionsEnabled properties to be initialized, and the XmlSchema property should be represented in a format that is necessary for the ReadXmlSchema method of the System.Data.DataSet class. Then the transformation of the model of the entity to the IoC-configuration of the ActiveRecord class will have the form (XSL)

```
<xsl:template  match='entity'>
  <component  name="{@name}"
        type="LightweightTransformSample.DbActiveRecord"
        singleton="false">
    <property  name="XmlSchema">
      <xml>
        <xsl:apply-templates  select="schema"/>
      </xml>
    </property>
    <property  name="SourceName">
        <value>
          <xsl:value-of  select="storage/sourcename"/>
        </value>
    </property>
    <property  name="VersionsEnabled">
        <value>
          <xsl:value-of  select="storage/@versions"/>
        </value>
    </property>
  </component>
</xsl:template>
```

The last stage is to determine the configuration of the IoC-container so that to carry out the transformation automatically when the configuration of the container is loaded (the models of entities are stored in the entityModel.xml.config file, and the description of the XSL transformation in the entityModelToIoCTransform.xsl file):

```
<components>
  <import
        file="config/entityModel.xml.config"
        xsl-file="config/xslt/entityModelToIoCTransform.xsl"/>
</components>
```

As a result, the definition “order” is accessible to be linked with other components. This example illustrates the elementary case of transforming a PIM model (which describes the abstract passive entity “order”) into a PSM as a Winter configuration of the IoC-container.

More complex transformations can be developed similarly. For example, if the DbActiveRecord class has no VersionsEnabled property, this aspect should be represented by the definition of one more entity to save the versions and to generate a configuration for the corresponding trigger that would create a new version for each modification in the purchaseOrder entity. In other words, the general strategy consists in the configuration of complex concepts from a PIM metamodel as a composition of more primitive concepts available in PSM.

CONCLUSIONS

The simplified infrastructure for MDD considered here meets modern requirements to creating large program systems. Note that the original representation of models as a domain-dependent XML format has no shortcomings of the XMI+XSL version. The model has the maximum compact representation, and programmers can look it through and modify without using specialized tools. It remains possible to quickly change and expand the metamodel. After stabilizing the domain-dependent metamodel, it is possible to determine the transformations to transform the domain-dependent models into any other models (for example, UML).

It is acceptable to use the XSL to describe transformations since this language is a powerful tool (contains a Turing-complete set of functions). Such an approach is especially efficient if the output model can be represented in the XML format. This allows organizing both vertical transformations with an arbitrary number of layers of abstraction, and horizontal ones.

In modern development, where it is necessary to minimize the time interval between the beginning of the development and first results (feedback), using a simplified infrastructure (based on XML standards) creates optimal conditions for efficient team work of programmers.

The approach proposed to introducing MDD was tested by the NewtonIdeas company (http://www.newtonideas.com), which focuses on creating many similar web projects in business process management systems (BPMS). A positive effect was obtained in two months of introduction, during which basic metamodels and transformations were formed (at present, about 900 concepts reused in various projects are defined; a library of more than 1000 classes is used to implement these concepts). In less than two years after the beginning of introduction, the approach described above showed to be completely efficient: without any significant expenses for the introduction, the degree of the code (and transformation) reuse increased by an order of magnitude.

REFERENCES

1. E. W. Dijkstra, “On the role of scientific thought,” Selected Writings on Computing: A Personal Perspective, Springer-Verlag, New York (1982).
2. O. L. Perevozchikova, Fundamentals of the System Analysis of Objects and Processes of Computerization [in Ukrainian], Vydavn. Dim KM Akademiya, Kyiv (2003).
3. A. Hunt and D. Thomas, Pragmatic Programmer: From Journeyman to Master, Addison Wesley (1999)
4. ISO/IEC 12207:1995 Information technology — Software life cycle processes (DSTU 3918-99 — Information technology — Software life cycle processes).
5. K. Czarnecki, T. Stahl, and M. Volter, Model-Driven Software Development: Technology, Engineering, Management, John Wiley&Sons (2006).
6. D. Gasevic, D. Djuric, and V. Devedzic, Model Driven Architecture and Ontology Development, Springer, New York (2006).
7. A. Kleppe, J. Warmer, and W. Bast, MDA Explained: The Model Driven Architecture™: Practice and Promise, Addison-Wesley Professional (2003).
8. ISO/IEC 19503:2005 Information technology — XML Metadata Interchange (XMI) (standard).
9. C. Walls and R. Breidenbach, Spring in Action, 2 ed., Manning Publications, (2007).
10. Lightweight .NET. Inversion of Control Container, Winter4Net (http://www.winter4.net)
11. M. Fowler, Inversion of Control Containers and the Dependency Injection Pattern (2004) (http://martinfowler.com/articles/injection.html)
12. P. Bertolazzi, C. Krusich, and M. Missikoff, “An approach to the definition of a core enterprise ontology: CEO,” Intern. Workshop on Open Enterprise Solutions: Systems, Experiences, and Organizations (OES-SEO 2001), Luiss Publications (2001).
13. M. Fowler et al., Patterns of Enterprise Application Architecture, Addison-Wesley Professional (2002).