Mapping-Based Exchange of Models Between Meta-Modeling Tools

Heiko Kern, Fred Stefan
University of Leipzig
Business Information Systems
Augustusplatz 10, 04109 Leipzig, Germany
{kern, stefan}@informatik.uni-leipzig.de

Vladimir Dimitriški, Milan Čelikovič
University of Novi Sad
Faculty of Technical Sciences
Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia
dimitrieski, milancel}@uns.ac.rs

Abstract

The exchange of models between meta-modeling tools is an important requirement. Tools often only cover a certain task in the development process. The exchange of models between different tools is necessary for covering a complete development process. Besides the aspect of cooperation, exchange of models also enables the replacement of old tools by new tools that better fit the customer’s needs. In order to avoid the vendor lock-in effect, model exchange allows the reuse of existing models. In this paper, we focus on the problem of model exchange and present an mapping-based approach between different meta-modeling tools. The approach is centered around a declarative mapping language with a graphical notation and a solution for connecting different metamodelling tools. We apply our approach to exchange models between MetaEdit+ and Microsoft Visio.

Categories and Subject Descriptors D.2.12 [Interoperability]: Data mapping; D.2.6 [Programming Environments]: Graphical environments

General Terms Design, Languages, Experimentation

Keywords migration, interoperability, mapping, code generation, model transformation, meta-modelling

1. Introduction

Models play an important role in Domain-Specific Modeling [6] and other related development disciplines. Generally, models represent a system in an abstract way, improve the understanding of a system, and facilitate the communication between different stakeholders. The creation of models is the result of a modeling process which is supported by a modeling tool. A special class of these modeling tools are meta-modelling tools. In addition to providing a user with a set of pre-defined modeling languages, meta-modelling tools provide a mechanism for the specification of new modeling languages. Modeling concepts of these languages are specified in a form of meta-models. Examples of meta-modelling tools are: MetaEdit+ [6], Generic Modeling Environment [12], and Microsoft Visio [2].

An important requirement for modeling tools, including meta-modelling tools, is the interoperability with other tools. In the context of this paper, interoperability is defined as the ability of two or more tools to exchange models or meta-models. Additionally, these exchanged models and meta-models must be usable in these tools. Often, tools support a specific task in the development process. Therefore, a successful application of the whole development process depends heavily on the degree of interoperability between the tools used. Besides the cooperation of tools, evolution of a tool landscape is an important aspect. As the software industry constantly evolves, modeling tools also evolve and old ones are being replaced by new tools that better fit the customer’s needs. In order to avoid the vendor lock-in effect, interoperability between tools is necessary and enables the reuse of existing models between tools from different vendors.

The paper is structured as follows. In Section 2, we introduce the problem of model exchange between meta-modelling tools. In the same section we derive requirements for our approach. In Section 3, we present our solution in detail. Afterward, we present a use case in Section 4 and a evaluation of the approach in Section 5. We discuss related work in Section 6 and conclude this paper in Section 7 with a summary and suggestions for future work.

2. Purpose and Requirements

In this paper we address the exchange of models between meta-modelling tools. In contrast to simple modeling tools, the exchange of models between meta-modelling tools is a challenging task because the exchange must consider the heterogeneity of the meta-models. A study about interoperability between meta-modelling tools [8] shows that currently there are no suitable approaches allowing this kind of model exchange. The majority of meta-modelling tools provides an export of meta-models and models in a specific serialization format, e.g. Extensible Markup Language (XML) with a predefined schema, XML Metadata Interchange (XMI), or Graph eXchange Language (GXL). These solutions are unsatisfactory because a serialization format cannot solve the problem of having different meta-model elements.

The tool approach that we want to present should exchange models in dependency of their meta-models between different tools. We determine that the models of the source and target tool implement the same language by using meta-models. These meta-models can have heterogeneity stemming from the tool-specific meta-modelling language and different ways to express a language. Although an automatic creation of the language in the target tool is possible, it is unsatisfactory as some of the particularities of the target meta-modelling tool are lost. Hence, we assume that the language is already manually specified by a language engineer in both tools. Besides this assumption, the source and target models must be
in an isomorphic relationship after the exchange. The approach we propose uses model transformation concepts from Model-Driven Engineering. Beside the defined purpose, the approach and the appropriate tooling must satisfy the following requirements:

(i) **Graphical and declarative language.** The mapping approach must comprise a declarative language with a graphical notation for specification of correspondences between source and target meta-models. The primary target audience of the mapping approach are users of modeling tools. These users are familiar with graphical modeling usually used in these kind of tools. Hence, we assume that a graphical mapping language fits this group’s needs better than a textual transformation language. Furthermore, we assume that the correspondence specification using a graphical language suits the exchange process better than the manual implementation of transformation code. The declarative nature of the language allows the abstraction from execution details. We want to hide these execution details from the user as the focus should be solely on the specification of correspondences between language concepts.

(ii) **Structural independence.** The model exchange approach must be independent from the structure of meta-models and models that have to be exchanged. The structural independence allows a broad application of the exchange approach and enables the connection of different meta-modeling tools. To support this requirement, a generic algebraic structure has to be utilized in the approach. All of the exchanged meta-models should be transformed to this generic structure before the specification of correspondences.

(iii) **Operational independence.** The mapping approach itself must be independent of the transformation execution environments. Using such a language, a user would be able to use the approach on two meta-models from different meta-modeling tools. Based on the correspondences, an executable transformation is created through an automatic generation process.

### 3. Mapping-Based Model Exchange Approach

Our approach for model exchange can be divided into several steps. In Figure 1 we present an overview of this approach. The first step is the import of meta-models from meta-modeling tools. For this task, we use M3-Level-based Bridges (M3B) [7, 9, 10]. A M3B transforms meta-models into an intermediate meta-model environment. After that, a binding component creates a generic tree structure for representing meta-models. Although the imported meta-models implement the same language, there is still some heterogeneity between their elements. For instance, elements with the same meaning may have different names or relationships can be specified in a different way. In order to overcome this heterogeneity, the user defines a mapping containing correspondences between different elements. In the next step, a generator iterates over specified mappings and produces an executable model-to-model transformation. The generator is specific to the pair of source and target meta-modeling tools as the transformation must be able to read source models and produce valid target models. The generated model-to-model transformations are defined against the imported source and target meta-models. The final step is the execution of the transformation including the exchange of models by using a M3B.

#### 3.1 Import of Meta-Models

An important aspect of the mapping tool is the exchange of model data (models and meta-models) between our mapping tool and connected meta-modeling tools. In the first step of the approach, meta-models are imported in our mapping tool and in the last step models are exchanged between the different meta-modeling tools. For the import and export of modeling data, we use the M3B approach. Generally this approach allows a transformation of models and meta-models between different meta-modeling environments. Furthermore, the M3B concerns technical and meta-modeling tool-specific heterogeneity. An M3B comprises at least two transformations, one at meta-model level and one at model level. The meta-model transformation transforms meta-models between meta-modeling environments. Both meta-models are in an isomorphic relationship necessary for the latter transformation at the model level. The model transformation reads models of the source meta-modeling environment and creates models that are to be used in the target meta-modeling environment. The created target models must conform to the meta-model prior imported by a M3B. Based on the imported meta-models, we create a general tree representation.

#### 3.2 Representation of Meta-Models

Our mapping approach aims to provide a generic mechanism for specifying mapping regardless of the meta-modeling tool. For this reason, we provide a generic tree representation of meta-models. A binding component reads the meta-models and creates a tree structure of the meta-model elements.

Concepts of the generic tree structure are presented in Figure 2. Each meta-model in a mapping is represented by an element container (ElementContainer). Each element container has a name corresponding to the name of an imported meta-model.
The binding configuration in turn specifies binding details for the
the binding type is necessary for reading the binding configuration.
ination in order to get the native meta-model elements. Furthermore,
during the generation process of the model-to-model transforma-
tion into a generic tree structure. The tree elements have a refer-
ence to the native meta-model elements. This reference is necessary
during the generation process of the model-to-model transforma-
ion in order to get the native meta-model elements. Furthermore,
the binding type is necessary for reading the binding configuration.
The binding configuration in turn specifies binding details for the
element container. A value of the side parameter (side) determines
if the element container has a role of source or target container in a
mapping.

Each element container comprises zero or more elements (El-
ment). Each element has a binding string (binding) and a name
(name). The name attribute corresponds to the name of an origi-
nal element from an imported meta-model. The path to the original
element is stored in the binding string. The format of the binding
string depends on the technology of the original element.

3.3 The Mapping Language

The mappings between two tree elements are expressed by a mapping
language. In Figure 3, we present the abstract syntax of this
mapping language. The root element of a mapping description is a
mapping container (MappingContainer). A mapping container
comprises element containers, zero or more links (Link) and nodes
(Node). An element container, introduced in the prior Section 3.2,
contains elements that can be part of a mapping. Each mapping
container has at least one source and one target element container.
Each mapping is represented with a selector (Selector) linked to
source and target elements. Based on the number of source and
target elements participating in a mapping, we classify selectors as
one-to-one (OneToOne), one-to-many (OneToMany), many-to-
one (ManyToOne), many-to-many (ManyToMany), and zero-to-
any (ZeroToMany). Selectors have a name (name property) for their
unique identification in a mapping container, and select expres-
sion (selectExpression property) for specifying a filter on all source
elements. Each selector is connected to source and target elements
using links (Link). As each link connects an element to a selector,
conditions relating to the single element are specified at the level of
the link (condition property).

Mappings can depend on other mappings. The dependency of
mappings results from the parent-child relation of the element
tree from the source or target structure. If all elements that are
being mapped have no parent elements in the tree, the mapping
is considered to be a root mapping. On the other hand, if one of the
participating elements has a parent element in the tree, the mapping
is considered to be a dependent mapping.

A selector can use one function (Function) that performs calcu-
lations on source elements participating the mapping. The result of
a function is to be mapped to a target element. As a function repre-
sents a reusable unit of code, it is specified at the level of a mapping
container and can be used by multiple selectors. For each function
usage (SelectorFunction) in a selector, arguments have to passed
_ArgumentAssignment_ to the function. Passing an argument from a
selector to a function requires a specific source element, identified
by its link (linkId property), to be passed to a specific parameter,
identified by the ordinal number in its function (parameterNum-
ber property). Source elements can be passed as input parameters,
while target elements get their value from output elements. A type
of an argument assignment is specified by the _parameterType_ prop-
erty of the _ArgumentAssignment_ concept.

The only selector not requiring a source element to be mapped
to a target element is the zero-to-any selector. This selector pro-
vides creation of any number of target elements when correspond-
ing elements do not exist in source meta-model. Target elements
can be created without setting values of their attributes. However,
a result of a function without parameters or a constant value (Con-
stantValue) can be assigned to the target element attributes.

3.4 Implementation

Our mapping tool allows the specification of mappings between
arbitrary meta-modeling tools. For this purpose, we have chosen
the Eclipse Modeling Framework (EMF) [18] as a mediator. This
meta-modeling environment offers basic meta-modeling concepts
in comparison to other environments [11]. These basic concepts
allow the emulation of concepts from other meta-modeling envi-
ronments. Furthermore, EMF provides various model processing
tools that suit our needs and ease the development process. All of
the meta-models and models are imported into EMF by using an
M3B. There are existing implementations of M3B which allow
the connection of different meta-modeling tools and EMF. Bind-
ing components are implemented in Java. The graphical editor for
mapping specification is implemented by two different technolo-
gies. We use the Sirius framework [17] for the rapid prototyping of
the graphical editor. Sirius allows us to try out new language con-
cepts by automatically generating the editor based on the mapping
language. The drawback of such editor is its hard customization.
On the other hand, we use the Standard Widget Toolkit (SWT) [19].
The SWT-based editor has to be developed manually without auto-
matic generation. However, it provides us with the full control over
the features and customization of the editor. Finally, code genera-
tors are developed in Xtend [21].

4. Case Study: EPC Model Exchange

In this section, we present a case study to demonstrate a complete
exchange process provided by our tool. The case study shows the
exchange of models between MetaEdit+ and Microsoft Visio. The
models in this example conform to Event-Driven Process Chain
(EPC) language.

The exchange process starts with the import of EPC meta-
models from both meta-modeling tools. As our approach uses
EMF as the mediator, both meta-models are first transformed by
an M3B into EMF. The outputs of the transformations are the
MetaEdit-EPC and Visio-EPC meta-models conforming to Ecore.
Afterwards, the binding components create tree representations
of both meta-models. In Figure 4, we present a mapping between the
imported EPC meta-models. The meta-model from MetaEdit+ is
presented on the left-hand side and the Visio meta-model is de-
picted on the right-hand side in this screenshot. The presented tree
view comprises meta-model elements, meta-metamodel elements,
and generic elements. The meta-metamodel elements are presented in italics which helps in organizing different meta-model elements. The generic elements are independent of a concrete meta-model. Generic elements are not defined by the language engineer during the meta-model specification. The generic elements are given by the meta-modeling tool. For instance, in Visio each shape has a text property which is often used as label in a shape.

The central part of Figure 4 depicts mappings with their selectors and links (cf. mapping language). For instance, the MetaEdit+ “Event Driven Process Chain” graph type is mapped to a Visio page as Visio does not support a graph type concept. The name property of this graph type is mapped to the generic element “text” of a Visio page. This property mapping is considered to be a child mapping of the graph type mapping. Both mappings are one-to-one mappings. Next, the object types are mapped into the corresponding object masters. The relationship type “Arc” from MetaEdit+ is mapped to the Visio connection master “Dynamic connector”. This mapping is a many-to-one mapping because the mapping needs to include arc role types. The referenced object of the “From” role type is mapped to “source” and the object of the “To” role type is mapped to “target” of “Dynamic connector”.

After defining the mapping, a generator creates the model-to-model transformation. Generally we can distinguish two parts of the transformation: a meta-model independent (MMI) and a meta-model dependent transformation (MMD). A MMI-transformation implements rules that are independent of certain modeling languages. These rules only depend on the meta-modeling tools that are participating in the exchange. Once implemented, these rules can be reused for all transformations between languages specified in these tools. An example of a MMI-transformation rule is a transformation of symbol coordinates between MetaEdit+ and Visio. An object in MetaEdit+ may be related to a symbol that has a position on a diagram. Similarly, each shape in Visio also has a position on a page. The MMI-transformation rule is used to transform the MetaEdit+ symbol position into the Visio shape position.

The second part (named MMD-transformation), concerns the mapping of the language-specific concepts. These transformation rules depend on particular modeling languages and not directly on meta-modeling tools. Therefore, the generation of this transformation depends on the previously defined mapping rules. In this case study, the generator produces an Epsilon Transformation Language (ETL) transformation. The generation of other transformation languages is also possible by simply invoking a different generator. In Listing 1, we present a snippet of the MMD-transformation code. Due to space limitations, we do not present the MMI-transformation. The first rule in Listing 1 corresponds to the first mapping rule that creates a Visio page for each MetaEdit+ EPC graph. The name of a MetaEdit+EPC graph is assigned to the name of a Visio page. Similar to this, the next rules represent a transformation of model elements that are instances of the Event, Function, XOR, and Arc concepts. Figure 5 shows a MetaEdit+EPC model as input and the resulting Visio EPC model.

5. Evaluation

In the previous section we presented one application of our mapping tool. In addition to the EPC case, we applied our tool to the exchange of models expressed in other modeling languages, such as Unified Modeling Language (UML) class diagram and Business Process Modeling Notation (BPMN). Based on these applications, we evaluate the approach with regards to the following criteria:

(i) Completeness and quality. The mapping tool works well with simple meta-models (e.g. EPC meta-model) and more complex meta-models (e.g. UML class diagrams and BPMN). Organizing meta-model elements as element trees is suitable for the representation of any meta-model. The mapping language provides users with enough concepts to express all needed mappings and to generate an executable model transformation between the participating meta-models. The end results of the migration process are satisfactory as the created target models fully correspond to the source models. We identified one weakness which is not directly related to our mapping approach but rather on the import and export of meta-models and models in our mapping tool. The quality of the exchange process depends heavily on the quality of the imported and exported model data. The used M3B approach has proven as suitable, but in the end the quality depends on the model interfaces provided by the connected meta-modeling tools.

(ii) Usability. One of the main elements of our tool influencing the usability aspect is its user interface. We distinguish between two
Listing 1. Snippet of the generated ETL transformation

```java
rule graph2page
transform event_driven_process_chain_3395083925 : INMM ! Arc_3395083925
  to evisiopage : OUTMM ! EVisioPage extends Graph2Page
  {
    evisiopage.text := event_driven_process_chain_3395083925.Name;
  }
rule event2Event
transform event_3395083771 : INMM ! Event
  to event : OUTMM ! Event
  { event.text := event_3395083771.Name; }
rule function2Function
transform function_3395083784 : INMM ! Function
  to function : OUTMM ! Function
  { function.text := function_3395083784.Name; }
rule xor2xor
transform xor_3395083742 : INMM ! XOR
  to _xor : OUTMM ! XOR
  {}
rule arc2dynamicConnector
transform arc_3395083800 : INMM ! Arc_3395083800
  to dynamic_connector : OUTMM ! Dynamic_Connector
  {
    dynamic_connector.target := arc_3395083800.me_role.equivalent();
    dynamic_connector.source := arc_3395083800.me_role.equivalent();
  }
```

Figure 5. EPC model example

6. Related Work

We found several mapping approaches and environments proposed in literature. Wimmer proposes in [20] an approach to model-based tool integration in the context of the ModelCVS project. Similarly to our approach, Wimmer proposes three steps: importing models, specifying mappings, and executing the mappings. This approach however heavily depends on the EMF technical space. Mapping operators are directly based on the notions and concepts from the EMF technical space. This solution also proposes execution of the mappings by executing colored Petri nets while our approach generates executable transformation code in a desired transformation language.

In [1], authors present the ATLAS Model Management Architecture (AMMA). AMMA is a set of tools and languages that can be used in the process of tool integration. The central part of AMMA is a mapping language and a tool named AT Model Weaver (AMW) [13]. AMW provides an extendable core language for specifying platform independent transformations. Authors of the paper argue that for a tool integration process, a specific language should be derived from the core language in order to cover the specific need of that process. However, we feel that this could be burdensome for the users of such a tool as for each integration scenario they need to create new concepts. Our goal is to provide a simple yet powerful language that can be used regardless of the tools being integrated.

Several other mapping approaches can be found in literature, such as Clio [15], Rondo [14], RDFT [16], and the UML-based approach presented in [5]. All of these approaches focus on the integration of certain technical spaces or languages, such as, XML.
Relational Databases or UML. However, none of these approaches focus on the integration of (meta-)modeling tools.

The mapping language in our tool can be regarded as a graphical transformation language used for the specification of platform independent transformations. Bollati in [3] presents an approach to modeling transformations using a composition of a graphical modeling language and natural language to specify platform independent transformation models (PIMs). The author proposes a definition of PIMs at a high level of abstraction while covering a relatively small amount of details. In contrast to that, we provide in our approach a meta-model of PIMs that allows a modeler to specify transformations with a significantly greater number of details. In much greater detail, several other graphical transformation languages are surveyed in [4]. However, these languages are used for platform specific transformation specification. They are provided with their own execution environment and thus are often limited in the type of meta-models that may be transformed. The languages described in [4] are mostly limited to the EMP technical space.

7. Conclusion

In this paper we presented a mapping-based approach to exchange models between different meta-modeling tools. The approach has three phases. In the first phase, meta-models are imported from the meta-modeling tools. For the import of the meta-models, we use M3-Level-based bridges. Imported meta-model elements are represented in the form of the generic element tree structure. In the second phase, a user can use a declarative language with a graphical notation to specify mappings between source and target meta-models. The final phase of the approach comprises the generation of executable model-to-model transformations.

In order to evaluate the approach, we presented an application scenario for our mapping tool. The case study concerns the exchange of EPC models between MetaEdit+ and Microsoft Visio. The results of the evaluation show that the mapping tool allows a user to generate complete executable transformations. We also showed, that the generic tree structure and the mapping language concepts are sufficient for easy and efficient mapping specification. From the viewpoint of adaptability and extendability, connections to other meta-modeling tools and generators could be easily add to our tool in the form of plug-ins.

One direction of future work is finding ways to improve the user interface of the mapping tool. In case of large meta-models, a mapping diagram could get overcrowded with links and selectors. This could be improved by using a tabular view of mappings with less graphical lines between elements. Furthermore, imported meta-models often have repeating structures. To assist the user in specifying new mapping models, we want to reuse existing mappings. Future research will be focused on developing algorithms that choose appropriate existing mapping to be applied to new situations.

Acknowledgments

Research presented in this paper was supported by the German Exchange Service and Ministry of Education, Science and Technological Development of Republic of Serbia as part of the bilateral project “Discovering Effective Methods and Architectures for Integration of Modeling Spaces with Application in Various Problem Domains”, 2014 - 2015.

References


