Marcus Alanen | Torbjörn Lundkvist | Ivan Porres

A Mapping Language from Models to XMI[DI] Diagrams

Turku Centre *for* Computer Science

TUCS Technical Report

No 676, 4 April 2005

A Mapping Language from Models to XMI[DI] Diagrams

Marcus Alanen
Torbjörn Lundkvist
Ivan Porres
TUCS Turku Centre for Computer Science
Department of Computer Science,
Åbo Akademi University
Lemminkäisenkatu 14, FIN-20520 Turku, Finland
{marcus.alanen,torbjorn.lundkvist,ivan.porres}@abo.fi

TUCS Technical Report
No 676, 4 April 2005

Abstract

We study the problem of how to define the concrete syntax of a modeling language based on the standards proposed by the Object Management Group (OMG). The OMG has recently published the Diagram Interchange (XMI[DI]) standard that contains a language to describe the graphical representation of a model. XMI[DI] is required to enableinteroperability between modeling tools. However, XMI[DI] can only describe particular diagrams of a model. We consider that the definition of a modeling language should also include a definition of what legal diagrams exist for that modeling language. In this article, we present a language to describe mappings between modeling languages and diagrams, some example mappings and our experience using them.

Keywords: Visual languages, Diagram Interchange, XMI[DI], MOF

TUCS Laboratory
Software Construction Laboratory

1 Introduction

The Object Management Group (OMG) maintains a series of standards such as the Meta Object Facility (MOF) [7] and the UML 2.0 Infrastructure [5] that can be used to define new modeling languages. These standards are actually modeling languages that are used to define other modeling languages. Therefore, a model in these languages is often called a metamodel.

The MOF and UML 2.0 Infrastructure are rich and complex metamodeling languages and they can be used to define modeling languages as large and complex as the complete UML 2.0. They can also be used to define domain specific languages and extensions or profiles to the UML. However, these metamodeling languages can only describe the structure or abstract syntax of a modeling language. They can be used to define model elements and relationships between model elements, but not their visual representation. The UML language, as defined by its metamodel, may state that there is a class named "Person" in a model, but it cannot state that this class is represented by a rectangle in a diagram with a certain position, size, and color.

To remedy this situation, the OMG has proposed the XMI[DI] [9] standard. XMI[DI] is yet another modeling language that has been defined following the same metamodeling approach as the UML. The XMI[DI] language brings new concepts such as **GraphNode** and **GraphEdge** that can be combined to represent two-dimensional diagrams. These diagrams can represent UML models graphically as UML practitioners are used to. However, XMI[DI] is not limited to UML and it can be used to represent diagrams for other modeling languages as well.

XMI[DI] is a key standard to exchange models between tools that need to represent, create or transform diagrams. Examples of these tools range from a simple diagram viewer to a full-featured interactive model editor or model transformation tool. Currently only one commercial tool, Gentleware's Poseidon, supports XMI[DI]. We consider this tool as a reference implementation of XMI[DI]. We also note that there exist important tools that do not actually need to process diagram information in a model. Examples of these tools are code generators or OCL [8] constraint evaluators. We should also note that despite the name similarity, XMI[DI] has no actual relation with XMI. XMI[DI] is not a model interchange format but a metamodel to describe diagram information. XMI[DI] models are serialized using XMI in the same way as a UML or a MOF model.

We consider that XMI[DI] is definitely an step forward in the OMG modeling standards. However, the current XMI[DI] standard does not tackle two important aspects that are necessary to completely define the appearance of new modeling languages. First, XMI[DI] does not describe how nodes and edges are rendered in a diagram. Neither the XMI[DI] metamodel nor the UML metamodel contain any information that states that a **GraphNode** representing an **Actor** is represented as a “stickman” while a **GraphNode** representing a **Class** is represented using a rectangle. However, the most important omission is that the standard does not include a mechanism to define how the abstract syntax of a model, expressed in a modeling language such as UML, relates to its concrete syntax, expressed in XMI[DI]. For example, the XMI[DI] standard does not state that a UML **Class** is represented as a **GraphNode** rendered as a rectangle that contains three other

GraphNodes that are layouted as a vertical stack, the top **GraphNode** representing the name of a Class, the middle **GraphNode** representing its lists of attributes and the bottom **GraphNode** representing its list of operations. This information is needed to create new XMI[DI] diagrams or to transform models that contain XMI[DI] diagrams.

Based on the limitations of XMI[DI], we consider that MOF, the UML 2.0 Infrastructure and XMI[DI] are not expressive enough to define new visual languages. A modeling language defined using MOF or the UML 2.0 infrastructure can describe the abstract syntax of a model but not its concrete syntax. An XMI[DI] model can provide a diagrammatic representation to a given model, but it cannot be used to define the concrete syntax of a complete modeling language. That is, XMI-DI can be used to represent particular diagrams but it cannot be used to define what are the all possible valid diagrams that can be used in a modeling language.

In this article, we tackle this last problem and study how to define a mapping between the abstract syntax of a modeling language described using the MOF or the UML 2.0 infrastructure and its concrete syntax described using the XMI[DI] standard. In the context of UML 2.0, this mapping is necessary to construct modeling and transformation tools that can create, transform and exchange model diagrams. In a broader context of Model Driven Engineering, this mapping is necessary to build generic modeling tools that can create and transform visual models in domain specific modeling languages.

We proceed as follows. In Section 2 we present the basis of XMI[DI] and define the need and uses of a mapping language from models to diagrams in more detail. Section 3 contains our proposal for such a mapping language and explains its semantics. We show a detailed example in Section 4 while we discuss how we have validated our approach in Section 5. We finally take a look at related work and conclude in Section 6, where we also consider future directions.

2 Models and Diagrams in the OMG Standards

In this section we describe how models and diagram are represented according to the UML and XMI[DI] standards.

We assume that a model is organized as an object graph that is an instance of a metamodel. Each node in this graph is an instance of a metaclass and each edge is an instance of a meta-association as defined in a metamodel. The UML metamodel contains more than 150 metaclasses such as **Actor**, **Class**, **Association** or **State** which describe the concepts that are familiar to UML practitioners. On the other hand, XMI[DI] is a rather small language with only 22 metaclasses. Figure 1 shows a fragment of its metamodel. There are basically three main concepts in XMI[DI]: **GraphNode**, **GraphEdge** and **SemanticModelBridge**. A **GraphNode** represents a rectangular shape in a diagram, such as a UML **Class** or an **Actor**, while a **GraphEdge** represents an edge between two other elements such as two nodes in a UML **Association** or a node and another edge such as in a UML **AssociationClass**. A **SemanticModelBridge** is used to establish a link between the semantic or abstract model and the diagrammatic model. For example, a **GraphNode** representing a UML **Class** is connected to that class using a SemanticMod-

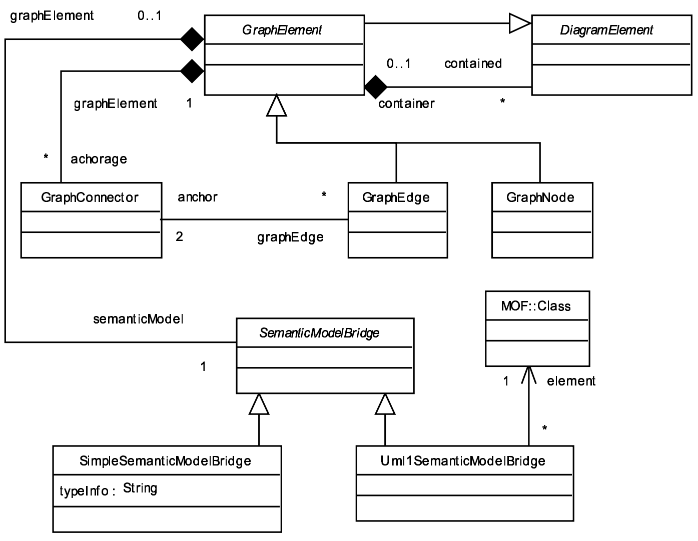

Figure 1: A subset of the XMI[DI] metamodel.

elBridge. There are two types of bridges. A **Uml1SemanticModelBridge** uses a directed link to an element , while a **SimpleSemanticModelBridge** contains a string named **typeInfo**. These concepts are explained in more detail in the XMI[DI] standard.

Figure 2 shows an example of a fragment of a UML model and its diagrammatic representation using XMI[DI]. The left part of the figure contains a single CompositeState containing two regions that are also CompositeStates. This UML model is an object graph organized according to the standard UML metamodel.

The middle part of the figure contains the CompositeStates represented using XMI[DI]. To simplify the figure we have abbreviated the **GraphNode** name to **GN** and omitted many XMI[DI] elements. Especially, we do not show the semantic bridge elements but just a link between XMI[DI] graph elements and the UML elements. We should also note that we show the links that correspond to composition associations using a black diamond. Although this notation is not defined in the UML standard it is useful for the purposes of this article. We can see that this particular XMI[DI] model contains GraphNodes that are not strictly necessary for this diagram. For example, the model contains a GraphNode to represent internal transitions of the state, even if there are no such transitions in the UML model. However, this is the XMI[DI] representation as produced by Poseidon.

Finally, the right side of the figure shows the XMI[DI] model rendered as an image, in this particular case as Encapsulated Postscript. This image was created by a tool based on the information contained in the UML model, the XMI[DI] model and built-in knowledge about the UML notation for Statecharts. Nothing prevents us to render the diagram to another format such as SVG or using a slightly different notation.

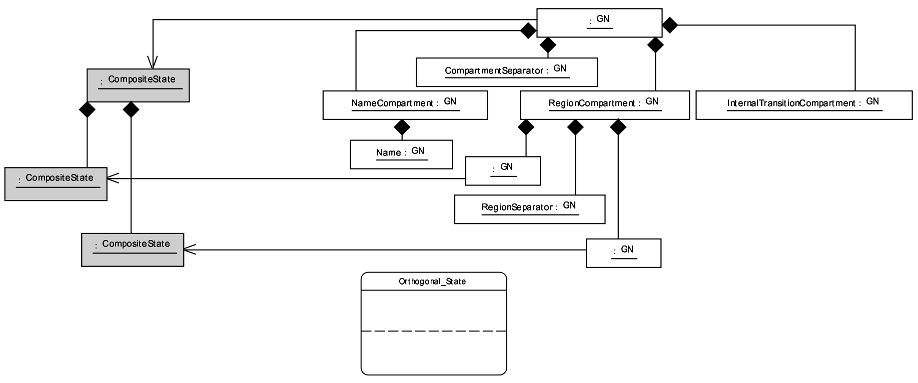

Figure 2: (Top) UML model with a composite state and two orthogonal regions and its diagram representation in XMI[DI]. (Bottom) XMI[DI] diagram rendered using the UML concrete syntax

2.1 From Models to Diagrams

We have seen in the previous example that the XMI[DI] provides us with the basic metaclasses that can be combined to create diagrams. However, neither the UML standard nor XMI[DI] tell us what metaclasses we should use to create a specific diagram to represent a specific model. As we have seen in the example, this task is not trivial since each UML model element is represented as many XMI[DI] elements and the mapping between the model element and its diagram representation is arbitrary. However the relation between UML model elements and their diagrammatic representation should be defined precisely since it is necessary to fully define new modeling languages to guarantee interoperability between modeling tools.

The relation between models and diagrams, or abstract and concrete syntax, can be defined as two different mappings, one from abstract to concrete syntax and vice versa. We consider the mapping between the abstract syntax to the concrete syntax of a modeling language as the most important. There are three main uses for this mapping language:

Documenting UML and other languages: The mappings can be used simply as documentation to complement the existing UML standards. We consider that the current UML 2.0 standard should be extended to include precise mappings from the UML 2.0 diagrams to XMI[DI].

Creating new XMI[DI] diagrams: Another obvious application of the language is to generate new XMI[DI] diagrams based on abstract models. This step may be necessary e.g. after reverse engineering source code into a UML model. Also, existing modeling tools may use a different language than XMI[DI] to represent diagrams internally. These tools may need to transform their proprietary diagrams into XMI[DI] in order to interoperate with other modeling tools.

Reconciling diagram and models: The most ambitious application of the mappings is to synchronize changes in an abstract model into an existing diagram. In this case, the mappings should be applied incrementally, preserving existing diagram information such as layout and colors when possible. This application is also the most demanding since it needs to be fast enough to be used in interactive model editors.

Nothing prevents us from defining mappings in the opposite direction from the concrete syntax to the abstract syntax. It may be necessary to implement a tool that converts a diagram represented as a bitmap or as SVG to a UML model. However, in most cases, it is simpler and more efficient to consider that the concrete syntax is derived from the abstract syntax and not the other way around. Therefore the discussion of the mapping from a diagram to a model is not in the scope of this article.

3 A Model to XMI[DI] Mapping Language

In this section we present a language called MODEL2XMIDI that can be used to define mappings between abstract modeling languages and the XMI[DI] language. Its purpose is to describe a unidirectional mapping from abstract model elements to XMI[DI] elements. We expect that the MODEL2XMIDI language can serve the three purposes described in the previous Section: First, it can be used to document the UML 2.0 standard. Secondly, given a model and its corresponding diagrams, we can map changes in the abstract model to the already-existing (but now obsolete) diagrams. Finally, we can create a diagram purely out of abstract model data.

This section discusses general requirements of such a mapping language, describes the concepts we have used in creating the language and the semantics of the language metaclasses. It is important to notice that we do not describe algorithms per se on how to implement the actual synchronization. Rather, given an abstract model, we provide a description of how the corresponding XMI[DI] diagram should be and leave the synchronization algorithms as a different topic. This split enables us to concentrate on acquiring a usable structure and semantics, while leaving the algorithms as a quality-of-implementation concern for modeling tools. However, in many cases it is quite self-explanatory how the synchronization needs to do, although it might not be trivial to accomplish. In our opinion this split works favorably for both standardization as well as enabling competing implementations.

3.1 Requirements

Since our understanding of modeling is the seamless combination of models of several modeling languages, our first goal is metamodel-independence. Given any modeling language, it should be possible to describe as many different kinds of diagrams as possible using some kind of mapping rules. This requirement also automatically satisfies any requirement for displaying models using different profiles, since they can be seen as different kinds of diagrams. This first requirement works nicely together with the current XMI[DI] metamodel which is also independent of the metamodel(s) of the abstract models.

Second, the system used to describe the diagrammatic mapping of a model—be it a language or even something completely different—should not require too much effort to implement. Especially this means that the mapping engine should be a generic solution without metamodel-specific quirks, able to map any abstract

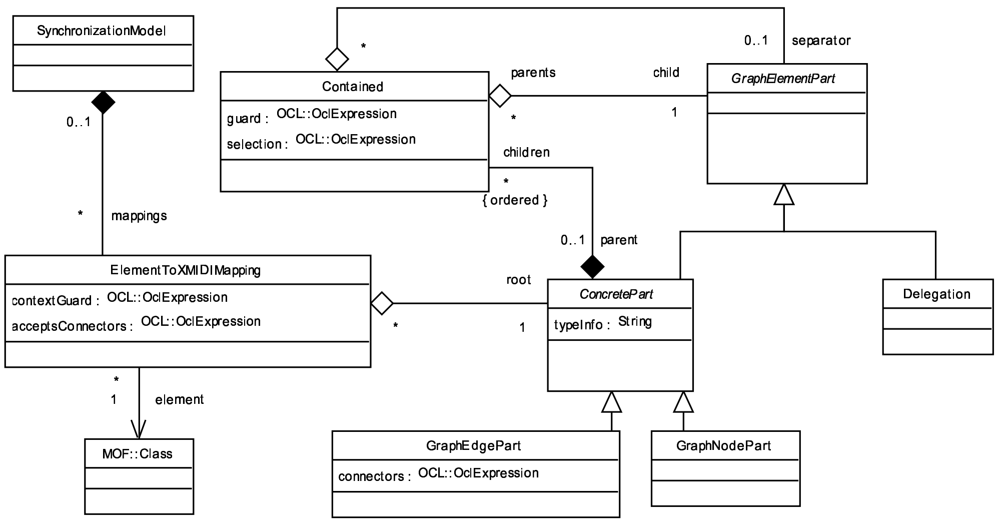

Figure 3: The MODEL2XMIDI metamodel.

model to its corresponding diagrams given suitable mapping rules. Supporting new metamodels should only require their MODEL2XMIDI mappings without changes to the mapping engine.

Third, the mapping should be efficient to be a viable mechanism for modeling tools. Although we do not have any formal definition of what efficiency means in this context, the intuition is that changes in the abstract model only trigger such mappings that are necessary to bring the diagrams up-to-date. Unnecessary work, by e.g. doing a lot of queries or calculations where none are necessary should be avoided. It is natural consequence of this requirement that the mapping works incrementally.

Finally. we should note that diagrams contain information about size, color and layout of its elements that is not present in the abstract models. Therefore, when we apply a mapping between the abstract to concrete syntax we may need to preserve as much information from existing diagrams as possible. That is, it should be possible to apply the mappings incrementally when the target diagram already exists.

3.2 Concepts

The metamodel for the MODEL2XMIDI mapping language is shown in Figure 3. In the Figure, **MOF::Class** represents the type of any metaclass, not just UML metaclasses. The **OCL::OclExpression** refers to any OCL expression. The following concepts and/or requirements are used when describing the semantics of the mapping language.

Abstract Element An *abstract element* is a model element which cannot be displayed as such. Diagram elements must be used to represent an abstract element graphically. In this article, abstract element types are mapped to XMI[DI] subtrees using MODEL2XMIDI mappings.

MODEL2XMIDI Mapping A MODEL2XMIDI mapping refers to a specific ElementToXMIDIMapping instance and its transitive children. A mapping always references a specific metaclass and there can be several mappings for a metaclass.

Current Abstract Element When mapping a part of a model, OCL evaluations are done in the context of an abstract element. The *current abstract element* points to that abstract element and is changed during the course of a mapping.

Subtree A MODEL2XMIDI subtree consists of a root ConcretePart and its transitive children. Parts of a MODEL2XMIDI subtree are mapped to an XMI[DI] subtree according to various constraints. The XMI[DI] subtrees can then be connected together and displayed on-screen.

Connecting a Subtree A MODEL2XMIDI subtree contains special **Delegate** elements which denote the position at which an XMI[DI] subtree can connect to another XMI[DI] subtree.

OCL Very good OCL support is required for advanced mapping needs. However, our experience tells us that most diagrams require only very simple OCL expressions to make the model to diagram mapping possible, so the OCL evaluation engine does not have to be very powerful.

3.3 Semantics

The main idea of the MODEL2XMIDI mapping language can be stated in three assumptions. First, that our diagrams can be built top-down, i.e. starting from the XMI[DI] Diagram element, child elements can be transitively connected to form a complete diagram without any changes required in their parents during diagram construction. This means that a parent diagram element does not depend on what child diagram elements exist underneath it. Second, that an abstract element can be mapped into an arbitrary XMI[DI] subtree with a single root element. The exact contents of this subtree may depend on the context of the abstract element as well as any transitive parent XMI[DI] GraphElements. Third, that there are rules describing how to connect these subtrees together to form the final XMI[DI] tree.

Next, the semantics of each construct in the language is specified. The root element in a MODEL2XMIDI model is a **SynchronizationModel** element, which contains a set of mapping rules in its **mappings** slot.

3.3.1 ElementToXMIDIMapping

A mapping model consists of several ElementToXMIDIMappings. Such an element **m** is the primary artifact designating one mapping rule. It is valid for instances of **m.element** or its subclasses. The **m.contextGuard** expression defines when a mapping is valid and can be used. It receives one parameter, **xparent** and is executed in the context of the current abstract element which must be an instance of **m.element**. The **xparent** parameter is the parent GraphElement in the XMI[DI] model. This is guaranteed to exist for any **GraphNode** or **GraphEdge** except for

Diagram, which has no XMI[DI] parent. If this expression evaluates to *true*, the mapping can be applied. If several mappings are valid a nondeterministic choice is made. If no mappings are valid, the element is ignored and cannot be mapped to XMI[DI] in the given context.

When the mapping is applied, the **m.root** ConcretePart is *accepted* in the context of the current abstract element and designates the starting point of the XMI[DI] subtree creation. Accepting a ConcretePart means that a corresponding XMI[DI] element (GraphEdge for GraphEdgePart and GraphNode for GraphNodePart) is created or already exists from a previous synchronization.

3.3.2 ConcreteParts

The semantics of a ConcretePart **c** is its corresponding XMI[DI] element connected to a **SemanticModelBridge** element. This **SemanticModelBridge** is a **SimpleSemanticModelBridge s** if **c.typeInfo** is nonempty and thus **s.typeInfo** is set equal to **c.typeInfo**, or a **Uml1SemanticModelBridge u** if **c.typeInfo** is empty and thus **u.element** is set to the current abstract element. This last part connects the XMI[DI] elements with their corresponding abstract element.

For a GraphEdgePart **p**, **p.connectors** describes the expression that when evaluated results in a sequence of abstract elements. For each element **e** in the sequence, a **GraphConnector** is created (or must already exist) and anchored to the GraphEdge corresponding to **p**. The owner of the GraphConnector must then be found in the set of all GraphElements whose corresponding abstract element is **e**. This GraphElement must correspond to a root ConcretePart in a ElementToXMIDIMapping mapping such that its **acceptsConnectors** is satisfied. The **connectors** expression is evaluated in the context of the corresponding abstract element and receives the GraphEdge as an additional parameter. The **acceptsConnectors** expression does not receive any parameters.

Although this scheme sounds complicated, it or similar functionality is required since not all GraphElements may be connected to and the only distinguishing mark is the context. In our work, this context is provided by the different ElementToXMIDIMappings.

The ordered list of Contained elements are *entered*. This is described next.

3.3.3 Contained

Creating different XMI[DI] diagrams based on context requires a framework which can conditionally add information.

Entering a Contained element **c** from a **ConcretePart p** in the context of the current abstract element **a**, the OCL expression **c.guard** is evaluated. An empty guard is assumed to be *true*. If the guard holds there are two slightly different outcomes. If **c.selection** exists it is evaluated. This expression must return an OCL collection **s** of elements. For each element **e** in **s**, the **c.child** GraphElementPart is accepted in the context *self* = *e*, *xparent* = *p*. It is worth emphasizing that this recursively accepts the “next” node in the ElementToXMIDIMapping subtree and changes the current abstract element to **e**. Usually in this case **c.child** is a **Delegate** element (described later). If **c.selection** does not exist, it is assumed that the current abstract element is not changed and the **child** GraphElementPart is accepted in the context *self = a*, *xparent = p*. Usually in this case **c.child** is an instance of ConcretePart with a nonempty **typeInfo**.

If **s** is ordered, the child mappings must also be done in that order. The **c.separator** is an optional subtree that will be added between each child in **s**. This enables us to easily model the very common occurrence of having a simple separator between values, such as a comma sign between the parameters in an operation in a UML class diagram.

The **guard** and **selection** expressions allow us to create a mapping to a subtree highly context-dependent on the abstract model element and all the other abstract model elements as well as the sequence of parents in the XMI[DI] tree. They, together with instances of ConcretePart and Delegate are the primary means to represent an XMI[DI] subtree as a MODEL2XMIDI subtree. Due to guards holding at different times and differing return values on the selections, the actual XMI[DI] subtree created looks like a subset of the MODEL2XMIDI subtree.

By using arbitrary OCL expressions the subtree can be dependent on any parts of the abstract model or any XMI[DI] parents. It must be emphasized that the **Contained.selection** allows us to "jump" in the abstract model from the current abstract element via several associations to other abstract model elements. Thus the mapping language is not limited to the structure of the abstract model regardless of the metamodel of that abstract model empowering us to create very versatile XMI[DI] models.

3.3.4 Delegation

In order to ensure the viability of our one-way synchronization language and make the language usable in practice, we need to decouple the representation and computation of the individual subtrees. This is done using Delegation elements. Such elements denote a change of ElementToXMIDIMapping rule and a new subtree creation can begin in the context of a new current abstract element and a GraphElement xparent. When the new subtree is ready, the Delegation element is replaced by the subtree.

3.4 Updating and Creating an XMI[DI] Diagram

Our update scenario from Section 2.1 assumes a model and a diagram which correctly reflects the model using a set of MODEL2XMIDI mappings. When a change to the model is done, the mapping framework inspects which abstract element and abstract element slots have changed. Based on the various **guard** and **selection** OCL expressions in the MODEL2XMIDI mappings, the OCL subsystem can *backtrack* which changes have invalidated which XMI[DI] elements. Based on this the mapping framework must somehow reconcile the XMI[DI] elements so that they will again correctly reflect the model. At its disposal it has the the current abstract model, the changes made to it, the now obsolete diagrams and the MODEL2XMIDI mappings and how the diagrams relate to the MODEL2XMIDI mappings. Because of the current abstract model and its changes the framework can also calculate information about the abstract model *before* the changes. As discussed in the introduction of this section, the exact algorithms used to do this are not explained further in this article.

In the MODEL2XMIDI framework, the problems of creating an XMI[DI] diagram and updating it based on changes in the abstract model are the same problem. Creation is just an extreme form of updating where the change to the abstract model can be seen as creating the whole abstract model, whereby the update phase will create the whole diagram.

4 Examples: UML CompositeState and UML Transition

In this section we present two example MODEL2XMIDI models. One mapping represents the concrete syntax of UML CompositeStates and the other mapping the concrete syntax of UML Transitions.

The representation of a UML CompositeState is quite interesting since it can be represented in two different ways: as a rounded rectangle or as an orthogonal region inside another CompositeState. These two representations are shown as two MODEL2XMIDI mappings in Figure 4. Depending on the context of the specific instance of the abstract element one of them is used. The first uses the contextGuard **self.isConcurrent**, which means that the mapping applies to orthogonal CompositeStates. The second uses a slightly more complicated contextGuard, **xparent.semanticModel.typeInfo == "RegionCompartment"**, to specify the mapping for regions in an orthogonal CompositeState. As stated in Section 3, each GraphNodePart in the mapping will map to a single GraphNode in XMI[DI]. All the different GraphNodePart children—for example the metaclasses **NameCompartment**, **CompartmentSeparator**, **RegionCompartment** and **InternalTransitionCompartment** — will also occur in XMI[DI] in the same order as in the MODEL2XMIDI mapping. In the **RegionCompartment** GraphNodePart, the **separator** is denoted by the dashed arrow and adds GraphNodes as separators between each region (given by **self.subvertex**. The separator has a SimpleSemanticModelBridge with "RegionSeparator" as typeInfo.

Other cases (i.e. mappings) of CompositeStates are not shown in the Figure.

A very important aspect of the XMI[DI] and MODEL2XMIDI models is the ordering of GraphElements and GraphElementParts. The **ConcretePart.children** property is ordered meaning that in the MODEL2XMIDI model the order of the occurrence of Contained should be preserved at any time during any possible incremental mapping to XMI[DI]. In the Figures the Contained edges are ordered left-to-right. Consider the scenario where one of the XMI[DI] creations is controlled by a **Contained.guard** which initially does not hold. Then a change occurs which makes the guard true, and the child XMI[DI] subtree is created. In an incremental mapping, the subtree is inserted in the corresponding position in XMI[DI] as specified in the MODEL2XMIDI model. E.g. the GraphNode mapped with GraphNodePart with typeInfo = "StereotypeCompartment" is always inserted after the GraphNode mapped with GraphNodePart with typeInfo = "Name". This also applies to Delegations. The GraphElementPart with typeInfo = "InternalTransitionCompartment" contains three Delegations that use the selections **self.entry–>asSet()**, **self.doActivity–>asSet()** and **self.exit–>asSet()**.

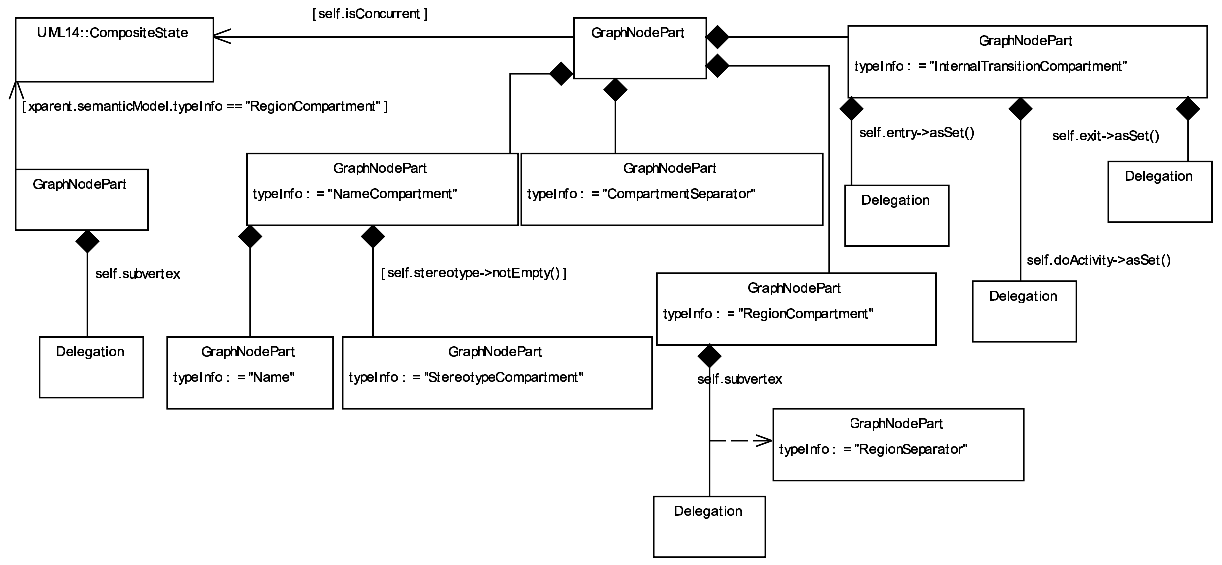

Figure 4: A subset of the XMI[DI] mapping model of CompositeState.

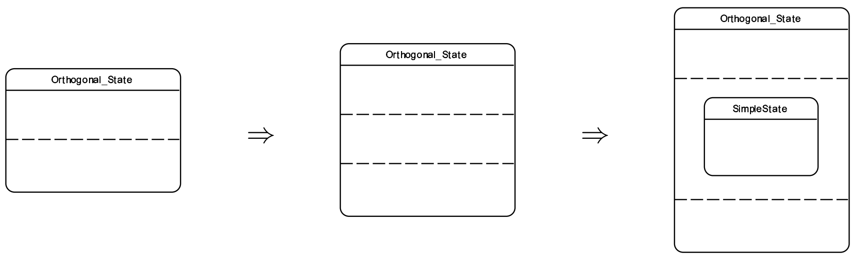

Figure 5: The different revisions of a CompositeState. The corresponding XMI-DI models are in Figures 2, 6 and 7.

An example of the application of this mapping for CompositeStates can be shown in Figure 2. If we apply this mapping to the UML model at the left side of the figure we will obtain the XMI[DI] model shown in the middle of the figure. A UML tool can layout the nodes, in this case resize them to a proper size, and render them into an image as shown in the right side of the figure.

In this example we can see how the ElementToXMIDIMappings are used for the CompositeStates. The topmost CompositeState has the property **isConcurrent** set to true (not shown in the Figure), hence the mapping with the contextGuard **self.isConcurrent** is used. The contents of the GraphNode with semanticModel.typeInfo = “RegionCompartment” is specified by a Delegation with a selection of **self.subvertex**. The selection contains two CompositeStates; now, using in turn the ElementToXMIDIMapping rule for regions in an orthogonal CompositeState, the subtree created and inserted contains one GraphNode mapped to the respective CompositeStates. Additionally, a GraphNode with the property semanticModel.typeInfo = “RegionSeparator” is inserted between the two regions.

We can also apply this mapping incrementally. Let us assume that we introduce a new region into the orthogonal CompositeState. This is achieved by inserting the new model element in the **subvertex** slot. This change triggers an incremental mapping of the XMI[DI] diagram. The incremental mapping component determines that there is one unmapped CompositeState, hence a new subtree for the region is inserted. The change also triggers a creation of a separator subtree, which is inserted before the new subtree for the region. The result is shown in Figure 6. Finally we insert a new SimpleState into the second of the three regions of the orthogonal CompositeState. The incremental mapping component determines that the **subvertex** slot of the region has changed, and inserts an XMI[DI] subtree representing a SimpleState into the XMI[DI] subtree of the region. The corresponding result is shown in Figure 7. The mapping for a SimpleState is not shown in any Figure due to space constraints.

The second MODEL2XMIDI example represents the mapping of a UML Transition and is shown in Figure 8. A Transition has only one ElementToXMIDIMapping without a contextGuard set, which implies that the same mapping is used for all instances regardless of the context. The root of the mapping is in this case a GraphEdgePart, since a Transition is mapped to a GraphEdge in XMI[DI]. The GraphEdgePart has specified two connectors, **self.source** and **self.target**, the sequence of abstract elements whose corresponding GraphNodes the GraphEdge can

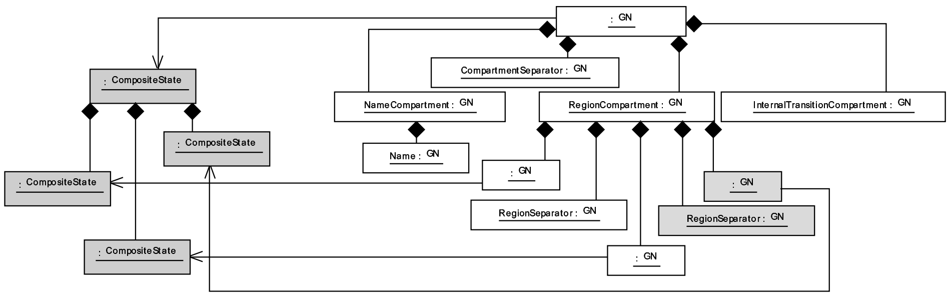

Figure 6: The abstract model (on the left side) and the XMI[DI] model (on the right side) after adding a third region.

connect to. Provided that the corresponding GraphNodes for these abstract elements can accept GraphConnectors, GraphConnectors will be created to connect the GraphEdge to the GraphNodes. A change in either **self.source** or **self.target** will trigger this mapping rule again and re-link the GraphEdge appropriately.

5 Validation of the MODEL2XMIDI Language and Known Limitations

We have built an experimental modeling tool called Coral that uses the XMI[DI] and simplified MODEL2XMIDI mappings to represent and maintain model diagrams. The tool has full interoperability with Gentleware's Poseidon. That is, it is possible create a UML model in Poseidon, save it into a file using the XMI, UML, and XMI[DI] standards, load it in Coral, transform the models and diagrams using the information provided in the MODEL2XMIDI mappings, save the model back into an XMI file and load it again in Poseidon without losing any model or diagram information. Figure 9 shows an screenshoot from Poseidon and Coral rendering the same model. The Coral tool supports other user-defined modeling languages and profiles besides standard UML. We have also created MODEL2XMIDI mappings to these languages although we are not aware of other tools that can process these models. All the figures in this article have been drawn using Coral. Coral is open source and it can be downloaded together with the UML to XMI[DI] mappings from `http://mde.abo.fi/`.

We have used a simplified MODEL2XMIDI mapping in Coral due to two reasons: performance and the lack of an OCL interpreter integrated in the tool. Therefore, we do not support full OCL expressions and we only check single property values in the **Contained.guard** and a subset (or subsequence) of a property value in the **Contained.selection**.

We have implemented mappings only for UML class, statechart, object and deployment diagrams but we are confident that the MODEL2XMIDI language can be used to define mappings for other UML diagrams. We consider that this simplified language is quite useful, but we acknowledge that the language proposed in this paper is more general. However, not even the full MODEL2XMIDI language is a complete transformation language, and there are some mapping constructs that

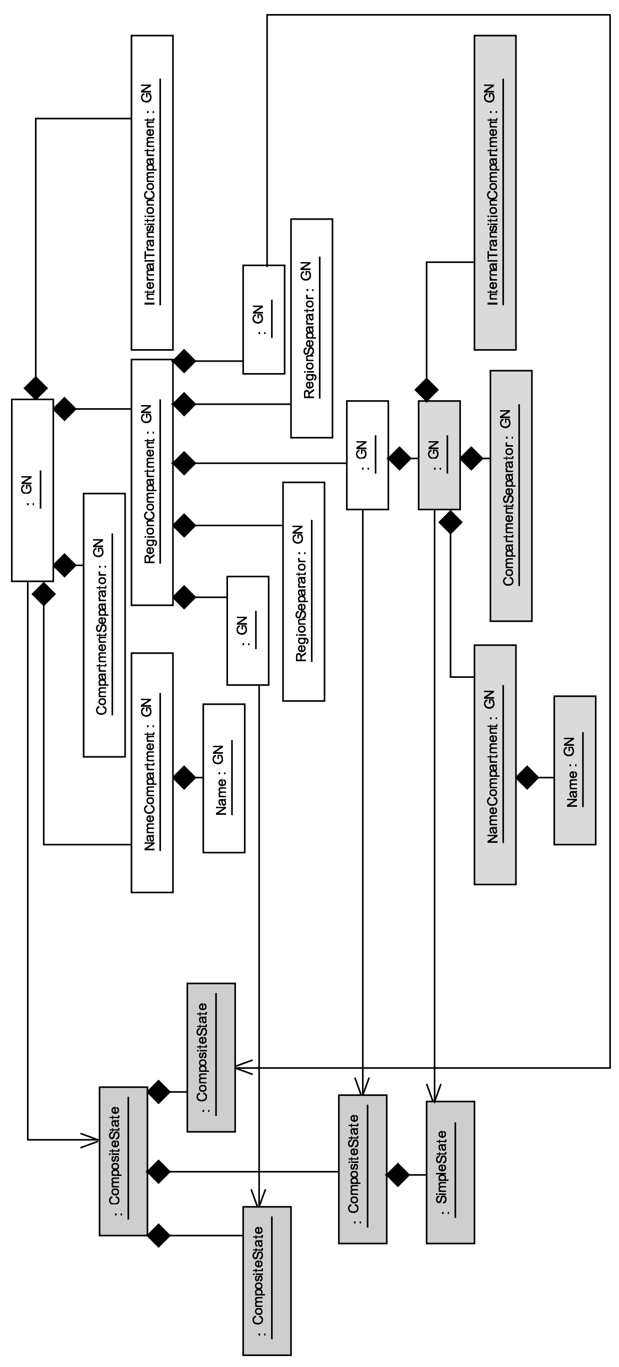

Figure 7: The abstract model (on the left side) and the XMI[DI] model (on the right side) after adding a simple state.

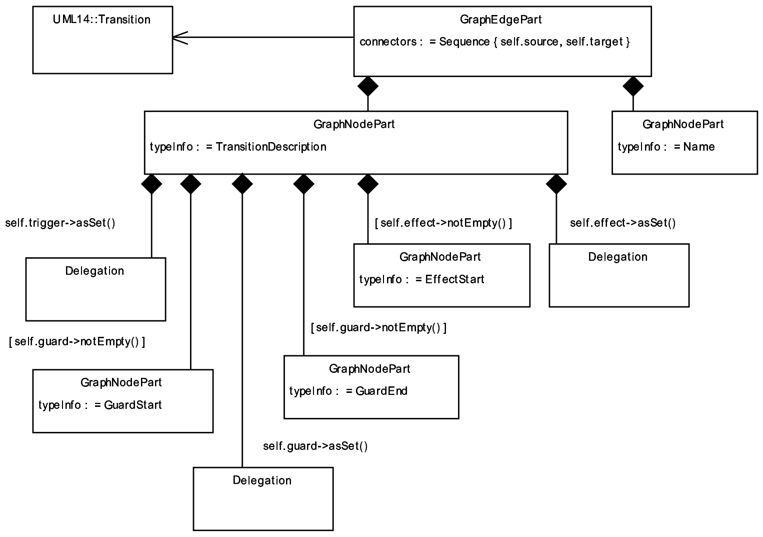

Figure 8: The XMI[DI] mapping model of Transition.

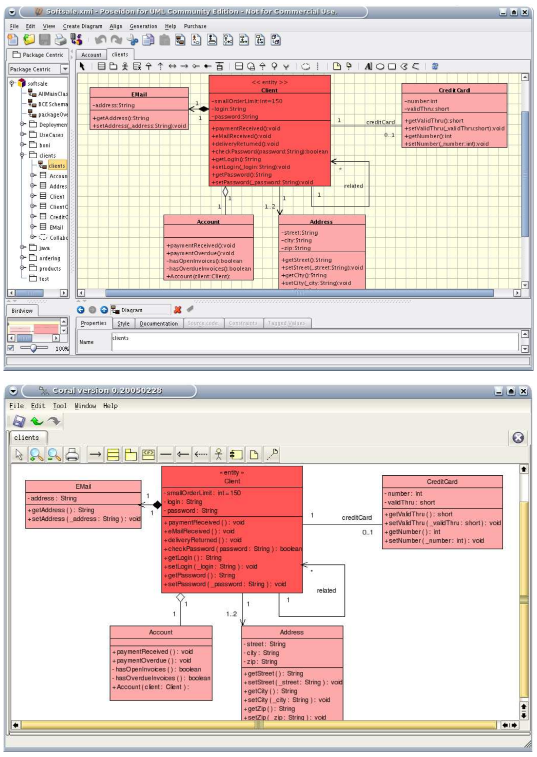

Figure 9: A diagram shown in Poseidon 3.0 (top) and Coral (bottom). The model was exchanged using the XMI 1.2, UML 1.4 and XMI-DI 2.0 standards.

cannot be expressed with it:

- It is not possible to obtain a **GraphNode** with a **SimpleSemanticModel-Bridge** with an empty string as the **typeInfo**.
- Since **GraphEdgePart.connectors** must return a sequence of abstract elements, it is not possible to create a **GraphEdge** which connects to elements with a **SimpleSemanticModelBridge**, only to elements connected using a **Uml1SemanticModelBridge** (or **CoreSemanticModelBridge**).

Additionally, several XMI[DI] metaclasses are not used by our mapping language at all. In some cases, a synchronization framework does not need them, whereas in other cases they might be useful. Some tools do not need a **CoreSemanticModelBridge** and handle everything via **Uml1SemanticModelBridge**. For example our Coral tool does this, effectively treating the XMI[DI] **Core::Element** as any [abstract] element and denoting it as **MOF::Class** in Figure 3. **Point** and **BezierPoint** are required for layout and position purposes and do not contribute to the structure of an XMI[DI] model, so they are not in the scope of MODEL2XMIDI or any similar framework. **Property** elements are not supported. While they have very little functionality as described in the XMI[DI] language and currently no defined structural functionality, they could be useful in the future. Similar considerations can be stated for **SemanticModelBridge.presentation**.

The reason no **Reference**, **DiagramLink** and **LeafElements** have been added is primarily that we have little experience with them and have so far opted to validate our research by basing it on working code. It can be noted that **Diagram** elements can easily be added into the MODEL2XMIDI language, but due to size constraints are not described further.

6 Conclusions and Related Work

In this paper we have studied a mapping language between the abstract syntax or semantic representation of a modeling language and its concrete syntax as a diagram. Beyond the scope of this paper is anything regarding diagrams that does not relate to the creation of XMI[DI] diagrams. This includes the layout of diagrams and the rendering of a diagram to an output device. Also, we did not discuss how an interactive editor can manipulate the elements in a diagram. For example, how and when the user can move a node or an edge in a diagram. Our future work in this area will explore more thoroughly what are the synchronization algorithms necessary and convenient in using the mapping language.

Several authors have proposed to use graph grammars to define visual languages [4] and there exist diagram editor generators for languages defined using graph grammars such as GenGed [1] and AToM [2]. We have followed a different approach and constrained our solution to the existing OMG modeling standards. In these standards, modeling languages are not defined using grammars but models. Therefore, it seems more adequate to use models to define the mappings between the abstract and concrete syntax of a language. This approach has important drawbacks: metamodeling is not as well defined and understood as formal languages and grammars are. Also, our mapping language is rather complex since it needs to support all the idiosyncrasies of the UML to XMI[DI] mapping. However, the advantage of following this approach is enormous: full compliance with existing standards and tools.

It can be argued that we could use the QVT [6] or another generic model transformation language to express transformations between the abstract and concrete syntax instead of creating a new mapping language. However, there is a need to implement these transformations to be as efficient as possible. That is why we have created a domain-specific transformation language that can only be used to define mappings between metamodels and XMI[DI]. There is a short discussion of a language to describe the concrete syntax of models in Chapter 9 of [3]. This language seems to assume that each model element is represented as one diagram element but this assumption does not hold for XMI[DI].

We have validated our approach by constructing an experimental tool and exchanging UML models and their diagrams with a commercial modeling tool that supports XMI[DI]. This allows us to conclude that the work presented in this article is a viable approach to define the concrete syntax of visual modeling languages based on the OMG standards. At the moment, the OMG does not have a Request For Proposals for a general mapping or transformation language from abstract models to XMI[DI] diagrams. We consider such a language important for interoperability reasons and hope that this article will spur further discussion on the topic.

References

[1] R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In Springer, editor, *Proceedings of the Fundamental Aspects of Software Engineering, 7th Intl. Conference, FASE 2004*, pages 214–228, 2004.

[2] J. de Lara and H. Vangheluwe. Using Meta-Modelling and Graph Grammars to process GPSS models. *Electronic Notes in Theoretical Computer Science*, 72(3), 2003.

[3] J. Greenfield and K. Short. *Software Factories*. Wiley, 2004.

[4] B. Meyer K. Marriot. *Visual Language Theory*. Springer, 1998.

[5] OMG. OMG Unified Modeling Language Infrastructure Specification, version 2.0, September 2001. Document ptc/03-09-15, available at `http://www.omg.org/`.

[6] OMG. MOF 2.0 Query / Views / Transformations RFP. OMG Document ad/02-04-10. Available at www.omg.org, 2002.

[7] OMG. MOF 2.0 Core Final Adopted Specification, October 2003. Document ptc/03-10-04, available at `http://www.omg.org/`.

[8] OMG. UML 2.0 OCL Specification, Ocober 2003. OMG document ptc/03-10-14, available at `http://www.omg.org/`.

[9] OMG. Unified Modeling Language: Diagram Interchange version 2.0, July 2003. OMG document ptc/03-07-03. Available at `http://www.omg.org`.

ISBN 952-12-1521-6
ISSN 1239-1891