A Mapping Language from Models to DI Diagrams

Marcus Alanen, Torbjörn Lundkvist and Ivan Porres

TUCS Turku Centre for Computer Science
Department of Information Technologies,
Åbo Akademi University
Lemminkäisenkatu 14, FIN-20520 Turku, Finland
e-mail:{marcus.alanen,torbjorn.lundkvist,ivan.porres}@abo.fi

Abstract. The OMG MOF 2.0 standard is used to define the abstract syntax of software modeling languages while the UML 2.0 Diagram Interchange (DI) describes the concrete syntax of models. However, very few tools support the DI standard, leading to interoperability problems. The primary reason for this is the lack of a formal way to describe the relationship between the abstract metamodel and its corresponding diagrams. In this article, we present a language to describe mappings between modeling languages and diagrams, some example mappings and our experience in using them. Better and correct support for DI would ease interchange of visual models and hasten the adoption of model-driven development.
Keywords: Visual languages, Diagram Interchange, XMI[DI], MOF, UML

1 Introduction

In this paper, we study the definition of visual languages based on metamodeling and the modeling standards maintained by the Object Management Group (OMG), such as the Unified Modeling Language (UML) [22].

The UML has become the de facto standard for software modeling in the industry. However, its definition is still vague and incomplete, even at the syntactic level. A rigorous and complete definition of modeling languages is necessary to enable the automatic generation of tools supporting these languages, in the same way that the rigorous definition of textual languages has yielded automated generation of tools such as lexical analyzers, parsers and compiler toolkits [13].

Several authors have proposed the using of graph grammars to define visual languages [18] and there exist diagram editor generators for languages defined using graph grammars such as GenGed [3], AToM [7], Tiger [9] and DiaGen [14]. One of the main differences between the technical space [4] defined by the OMG modeling standards and previous approaches is that the abstract syntax and concrete syntax of a modeling language are two independently defined and maintained artifacts.

In a modeling language, the definition of its abstract syntax includes the definition of all model elements that can be used in a language, their properties and relationships with other elements. It can also include additional constraints, also know as well-formed rules. The definition of its concrete syntax includes the visual appearance of model elements and layout constraints. The complete definition of a visual modeling language

should include the mapping between the abstract and its concrete syntax, that is, the mapping between models and diagrams. This is necessary to create new diagrams from existing models or to parse a diagram into a model.

In the context of the OMG standards, the abstract syntax of a language can be defined using the Meta Object Facility (MOF) [20] and the UML 2.0 Infrastructure [21]. 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 that can be used to define modeling languages as large and complex as the UML 2.0 Superstructure. They can also be used to define domain specific languages and extensions or profiles to the UML.

The OMG has a standard for two-dimensional diagrams called the UML 2.0 Diagram Interchange [23] (DI). DI is a modeling language that has been defined following the same metamodeling approach as the UML. While DI has been developed to satisfy the need for diagram interchange for UML diagrams, it is not strictly restricted to UML in any way. That is, DI can be used to represent diagrams for other modeling languages as well. As a consequence, 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.

However, we should note that DI is a language to express concrete diagrams. It does not address the issue of defining the concrete syntax of modeling languages. That is, while DI can be used to represent and interchange diagrams for a model, it cannot be used to determine if a given diagram is valid for a given model, it cannot enumerate all the possible valid diagrams for a particular modeling language, and neither does it contain the necessary information to create a new diagram from an existing model. While Appendix A and C of the DI specification attempt to address this issue by providing an informal mapping from UML to DI, these mappings still lack the details required for determining precisely when a specific diagram is valid.

Considering this, we argue that the OMG standards cannot completely specify the concrete syntax of a visual modeling language. We can see an example of this in the definition of UML 2.0. In this language, there are a group of model elements in the interaction packages that can be represented in at least three different diagrams: sequence, interaction overview and communication diagrams. That is, the same concepts from the UML abstract syntax can be represented in three completely different ways in a diagram. Although these diagrams are explained informally in the UML standard, neither the UML nor the DI specification contains the information required to construct sequence, interaction and communication diagrams using the DI language. This has also been noticed by Dr. Guus Ramackers, who has notified the OMG about it [25].

In this article, we tackle this 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 DI standard. In the context of UML 2.0, such language is necessary to complete the definition of UML and to construct modeling and transformation tools that can create, transform and exchange UML model diagrams. In a broader context of Model Driven Engineering, this mapping

can be used to build generic modeling tools that can create and transform visual models and diagrams in domain specific modeling languages.

We proceed as follows. In Section 2 we present the basis of DI and define the need of and use for 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 discuss how we have validated our approach in Section 4. We finally take a look at related work and conclude in Section 5, where we also consider future directions.

2 A Mapping Language from Models to Diagrams

In this section we describe basic concepts behind the UML and DI standards and we describe the idea behind a mapping language between these languages.

In order to ensure interoperability between modeling tools, we consider that the mapping between the abstract and concrete syntax of a modeling language should be defined precisely. This is necessary in order to fully support DI diagrams for both new and existing modeling languages. This mapping can be defined using a mapping language, which we call DIML, from a modeling language to DI. An overview of this mapping can be seen in Figure 1. In this setting, we assume that this mapping language is defined using the OMG MOF standard. The actual mappings are described using a model in this mapping language. Each of these models maps an element in the modeling language to a set of elements in the DI language. This information can then be used by an application of this mapping language that interprets the mappings and applies them to actual model data.

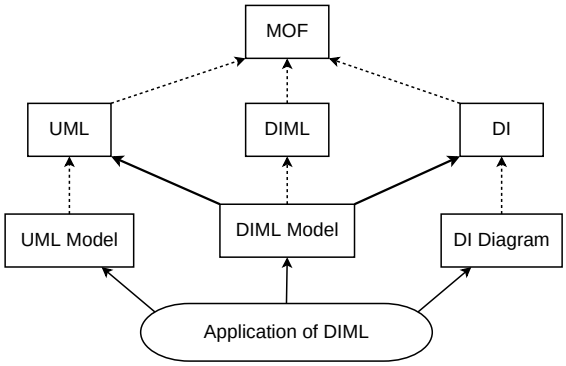

Fig. 1. Overview of the mapping between models and diagrams.

There are three main applications of this mapping language:

Definition of UML and other languages: It 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 definitions of the valid UML 2.0 diagrams using the DI standard.

Creation of new DI diagrams: Another obvious application of the language is to generate new DI diagrams based on abstract models. This step may be necessary e.g. after reverse engineering source code into a UML model or converting models from one modeling language to another. Existing modeling tools may use a different language than DI to represent diagrams internally. However, these tools may need to create diagrams into DI in order to interoperate with other modeling tools using the OMG standards.

Reconciliation of diagram and models: The most ambitious application of the mappings is to reconcile 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.

2.1 The UML 2.0 Diagram Interchange

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, DI is a rather small language with only 22 metaclasses; a relevant subset of them is shown in Figure 2. There are basically three main concepts in 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 SemanticModelBridge. There are two types of bridges. A *Uml1SemanticModelBridge* uses a directed link to an element , while a *SimpleSemanticModelElement* contains a string named *typeInfo*. These concepts are explained in more detail in the DI standard.

Figure 3 shows an example of a fragment of a UML model and its diagrammatic representation using DI. The top part of the figure is a simple UML statemachine model with two states and one transition, presented as a UML object diagram. From this object diagram we can see that this DI model contains elements necessary for displaying and layouting information retrieved from the UML model. To simplify the Figure, we have omitted some UML and DI elements. Especially, we do not show the Uml1SemanticModelBridge elements but merely a directed link between 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.

Finally, the bottom part of the figure shows the same 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, such as the name of the states, the DI model, such as the layout of the states, and built-in knowledge about the UML notation for state machines, such as the fact that a state is represented as a rectangle with

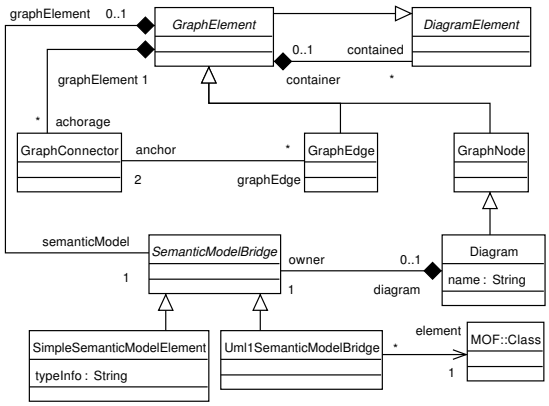

Fig. 2. A subset of the DI metamodel.

rounded corners. Nothing prevents us from rendering the diagram to another graphical format such as SVG.

2.2 DIML: From Models to Diagrams

We have seen in the previous example that the DI provides us with the basic metaclasses that can be combined to create diagrams. However, neither the UML standard nor 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 using many DI elements and the mapping between the model element and its diagram representation is arbitrary. This in turn complicates the interchange of DI diagrams between modeling tools, as diagrams created by one tool may not be compatible with the diagrams the other tool creates. Full compatibility can be ensured only if the tools use the same definitions for creating the diagrams.

To address this issue, we have created a language called the Diagram Interchange Mapping Language (DIML). Its purpose is to define mappings between metaclasses in MOF-based modeling languages, such as UML, and corresponding elements in the DI language. We can see three example DIML models for UML StateMachines, SimpleStates and Transitions shown in Figures 4, 5 and 6 respectively. It must be noted that we have simplified the structure of StateMachines for the purposes of this article. In the figures, an abstract element on the left is mapped to a hierarchy of diagram elements as DIML *Parts*. Each Part, shown as rectangles, maps to a GraphNode, GraphEdge or Diagram in DI. The directed arrow corresponds to the mapping concept, whereas the edges with black diamonds correspond to parameterized element ownership based on *guard* and *selection* statements. The hierarchy forms a skeleton which when transformed into DI elements give us the intended result.

An example of the application of these three mappings was seen in Figure 3. The topmost part of the figure (colored gray) shows a StateMachine with two SimpleStates and one Transition. When the mapping for UML StateMachines (Figure 4) is applied

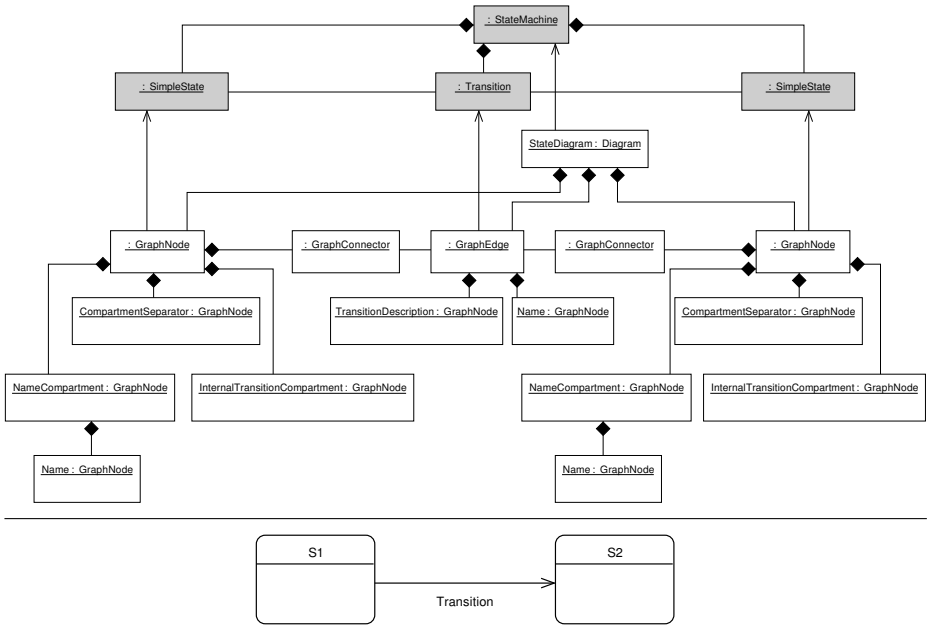

Fig. 3. (Top) UML model in gray with two SimpleStates and a Transition and its diagram representation in DI. (Bottom) DI diagram rendered using the UML concrete syntax.

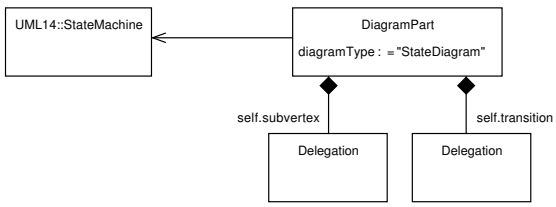

Fig. 4. The DI mapping rule of StateMachines.

to the StateMachine, a DI Diagram will be created. When the mapping for UML SimpleStates (Figure 5) is applied to the SimpleStates and the mapping for UML Transitions (Figure 6) is applied to the Transition, DI elements will be created for these UML elements. Finally, these DI elements will be connected to the Diagram. As a result, the DI model shown in the middle of the figure is obtained. By comparing the DIML models to the actual diagram, we see that not all DIML Parts are represented in the resulting diagram. For example, there is no *StereotypeCompartment* for the SimpleStates. This is an example of the parameterization; since the SimpleStates had no abstract Stereotype elements, the guard "self.stereotype–>notEmpty()" in the DIML model failed and thus no StereotypeCompartment was created.

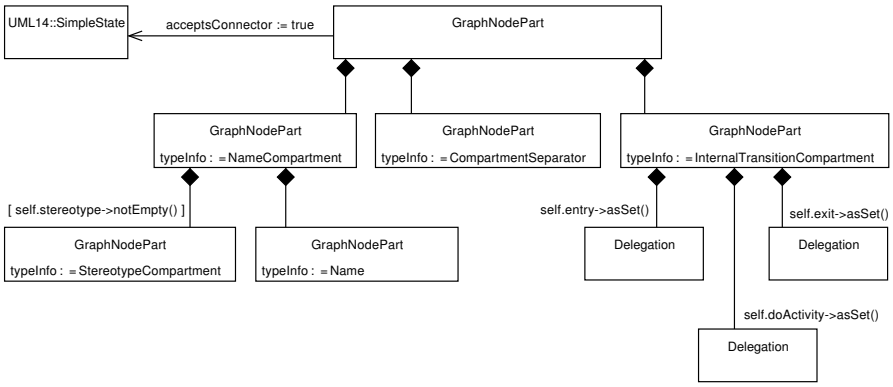

Fig. 5. The DI mapping rule of SimpleState.

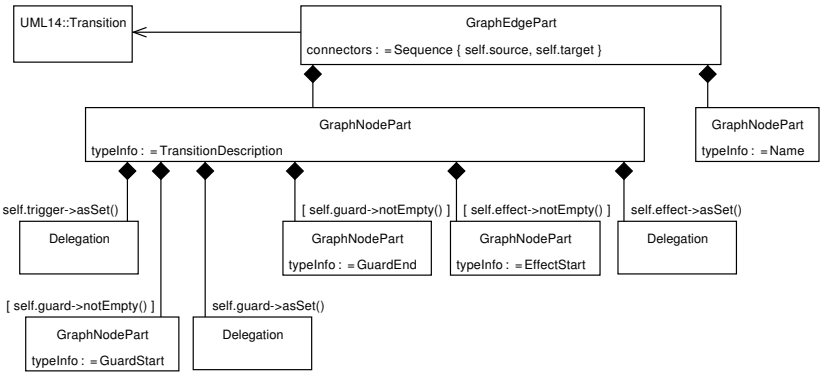

Fig. 6. The DI mapping rule of Transition.

3 Metamodel and Semantics

This section discusses the concepts we have used in creating DIML and the semantics of the language metaclasses. It is important to notice the separation between the DIML language itself and the various applications of the DIML language. While the main use of DIML is to define diagrams using the OMG standards, DIML does not define or enforce any particular method for applying these mappings on model data. Assuming that a DIML mapping is correct, any tool is still allowed to maintain the abstract model and concrete models in any way it wants as long as the end result is correct, i.e., as if it had used DIML. This *as if* rule is well-known from for example C compiler technology and gives implementations the greatest leeway while still retaining compatibility between implementations.

This separation enables us to concentrate on acquiring a usable mapping language and its semantics, while leaving the actual applications of DIML as a separate concern for modeling tools. In our opinion this separation works favorably for both standardization as well as enabling competing implementations.

3.1 The Basics of the Metamodel

The metamodel for the DIML mapping language is shown in Figure 7. In the figure, *MOF::Class* represents the type of any metaclass, not just UML metaclasses. The *OCL::OclExpression* refers to any OCL expression. OCL is a language for creating arbitrary queries on models. It can be used to collect some elements from models or to assert that certain properties hold in a model.

The *MappingModel* is a simple container metaclass to collect all the mappings as children under instances of it. Every DIML model must have one MappingModel as its root element. An *ElementToDIMapping* element *m* is a description of mapping one abstract element of type *m.element* to corresponding DI elements. Thereby the three mappings for StateMachine, SimpleState and Transition from Figures 4, 5 and 6 have been used to create several DI tree fragments as shown by triangles in Figure 8, yielding the final DI diagram in Figure 3.

Every mapping is considered in the specific context of *xparent*, which is the parent element in the DI model. It is guaranteed to exist for any *GraphNode* or *GraphEdge* except for Diagram, which has no DI parent.

In Figures 4, 5 and 6, the ElementToDIMapping elements are denoted by directed arrows and the Contained elements are the composition links. There can be two different text strings next to those links; a text in brackets is a *guard* expression, and a text without brackets is a *selection* expression. We will explain these and the *contextGuard*, *acceptsConnector* and *validIn* properties later.

3.2 DIML Tree

A DIML tree consists of an InitialPart as its root, and a hierarchy of Contained and GraphElementPart (and its subclasses) elements. Leaves in the tree are either of type Delegation or have no *children* Contained elements. The purpose of a DIML tree is to

describe a parameterized skeleton which can be used to compute a resulting DI tree. Parameterization here means that the occurrence and recurrence of child GraphElementParts is determined by the slot values in *Contained.guard* and *Contained.selection*.

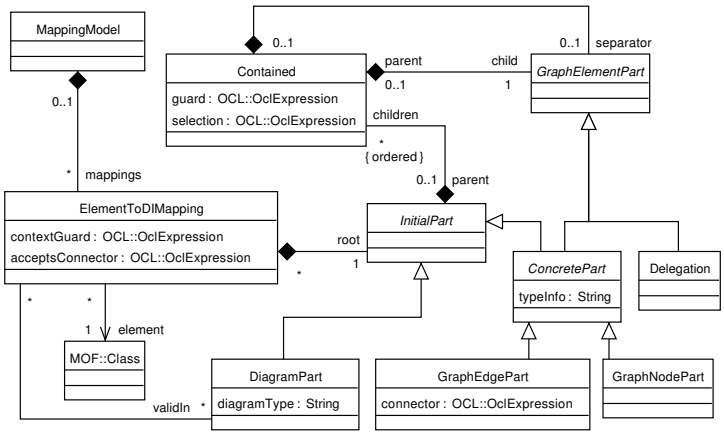

Fig. 7. The DIML metamodel.

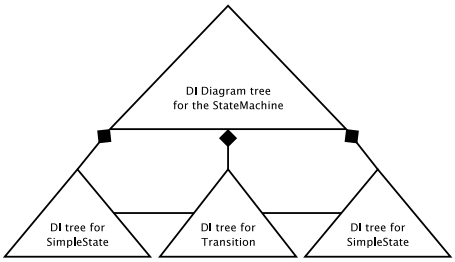

Fig. 8. DI fragments created by the DIML mappings are combined into the final DI diagram.

The DIML tree can be computed in the context of an InitialPart *i*, its current abstract elements *a* and its *xparent*. For every Contained element *c* in the *children* slot of the InitialPart, we must do the following:

- Evaluate *c.guard* in the context of *a* and with *xparent* as its parameter. If it does not hold, we must proceed to the next Contained element.
- Evaluate *c.selection* in the context of *a* and with *xparent* as its parameter. The expression must return an OCL collection *s* of abstract elements. For each element *e* in *s*, the *c.child* GraphElementPart is *accepted* in the context of *e* as the abstract element, and *i* as its xparent.

– If *c.separator* is non-empty, it denotes a DIML subtree with corresponding DI elements that must be placed between each accepted element. 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.

Here, accepting means that the same computation must be performed on the new child DIML element if the child is a ConcretePart. Delegation elements on the other hand arise from the need to decouple the representation and computation of individual DIML trees. If the new child DIML elements is a Delegation, we must search for a new valid mapping for the abstract element *e*. If several mappings are valid a nondeterministic choice is made. If no mappings are valid, the element is ignored and cannot be mapped to DI in the given context. Once a valid mapping is found, DIML tree creation can begin in the context of a new current abstract element and xparent. The corresponding DI elements of the parent DIML tree and the child DIML tree are then connected together at the place of the Delegation element in the parent DIML tree.

The *guard* and *selection* expressions allow us to create a mapping to DI highly context-dependent on the abstract model element and all the other abstract model elements as well as the sequence of parents in the DI model. They, together with instances of ConcretePart and Delegation are the primary means to represent a collection of similar DI fragments (modulo the parameterization) as one DIML tree.

3.3 Support for Diagrams

A mapping *m* of a diagram is such that *m.root* is a *DiagramPart* element *r*, with *r.diagramType* denoting what diagram type is being considered (e.g. “ClassDiagram”). The *m.contextGuard* is evaluated and must return true. It is an OCL expression which receives the abstract element and xparent (which in this case is a null pointer/reference) as its parameters. It can be used to limit whether or not it is allowed to create a diagram for the given abstract element.

The *m.validIn* slot is unused and must be empty. The *m.acceptsConnector* is unused. Starting at *r*, the DIML tree can be described.

3.4 Support for GraphNodes and GraphEdges

The mapping *m* for GraphNodes or GraphEdges is otherwise similar to the mapping for a Diagram, but with some small differences. The element *m.root* must either be a *GraphEdgePart* or a *GraphNodePart*, with *m.root.typeInfo* being the empty string.

The *m.contextGuard* must still hold, but the xparent will now be a valid DI element in the diagram. The set *m.validIn.diagramType* denotes the valid diagram type set, e.g. { “ClassDiagram”, “SequenceDiagram” }. This is the set of types of diagrams in which the mapping can be applied. Although technically the validIn information could be embedded in the contextGuard, it is more convenient to have a set of diagrams where a mapping can be applied because a) it avoids unnecessarily long OCL expressions in the contextGuard, and b) the information about suitable diagrams is easier to extract from a slot made for that purpose rather than extract it by parsing an OCL expression.

We will explain *m.acceptsConnector* and *GraphEdgePart.connector* later. Again, starting at *m.root*, the DIML tree can be described.

3.5 Correspondence of DIML Elements with DI

An instance *p* of DiagramPart, GraphEdgePart or GraphNodePart corresponds to an instance of the DI elements Diagram, GraphEdge or GraphNode *d*, respectively.

A Diagram has a SimpleSemanticModelElement *s* in its *semanticModel* slot such that *p.diagramType* = *s.typeInfo*, and a Uml1SemanticModelBridge in its *owner* slot which points to the abstract element for which the diagram was created for. A GraphEdge or GraphNode has either a Uml1SemanticModelBridge or a SimpleSemanticModelElement. If *p.typeInfo* is empty, *d* must have a Uml1SemanticModelBridge which points to the abstract element. Otherwise, *d* must have a child element *s* of type SimpleSemanticModelElement such that *p.typeInfo* = *s.typeInfo*.

3.6 Connecting Edges to GraphConnectors

The *connector* expression is evaluated in the context of the corresponding abstract element and receives the GraphEdge as an additional parameter. For an instance *p* of GraphEdgePart, *p.connector* 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 in the same diagram whose corresponding abstract element is *e*. This GraphElement must correspond to a root ConcretePart in an ElementToDIMapping *m* mapping such that *m.acceptsConnector* is satisfied. The *acceptsConnector* 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 ElementToDIMappings.

3.7 Limitations

Having explained the semantics of DIML, we must also be concerned about its limitations. The main idea of the DIML language can be stated in three assumptions or limitations, depending on the point of view. First, that our diagrams can be built top-down, i.e., starting from the 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 DI element does not depend on what child DI elements exist underneath it. This is emphasized by the Delegation elements in the DIML models; the decoupling they provide allows us to mix several kinds of diagrams together. Although the various OCL expressions have access to the chain of parents, they cannot modify them since OCL is a side-effect free query language, and in our semantics of DIML they would nevertheless not be allowed to modify them.

Second, that an abstract element can be mapped into a DIML tree with a single root element. The exact contents of this tree may depend on the context of the abstract element as well as any transitive parent DI elements. In general, by using arbitrary OCL expressions the tree can be dependent on any parts of the abstract model or any DI parents. It must be emphasized that the *Contained.selection* allows us to navigate 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 DI models.

Third, that there are rules describing how to connect these trees together to form the final, complete DI tree. These rules have been described in this Section.

4 Validation of the DIML Language

We have built an experimental modeling tool called Coral that uses the DI and simplified DIML mappings to represent and maintain model diagrams. We have implemented a component for this tool that reconciles models and diagrams after executing model transformations or performing editing operations [2, 17], based on the abstract model and the DIML mappings.

The guards of the rules in the simplified DIML mapping language use a very reduced version of OCL. This is done for performance reasons. Instead of allowing complete OCL queries that could require the traversal of the whole model, we allow the checking of single property values in the *Contained.guard* and a subset (or subsequence) of a property value in the *Contained.selection*. This restriction has enabled us to perform reconciliation of models and diagrams using linear algorithms with very few exceptions, while still being able to support large and complex languages such as UML. This ensures that diagram reconciliation is not an expensive operation, and hence it is fast enough to be integrated with an interactive model editor; our implementation is of sufficient speed for interactive editing. We consider that the simplified language serves the purposes we have outlined in Section 2, but we acknowledge that the language proposed in this paper is more general.

We have implemented mappings for the UML 1.4 class, statechart, object, use case and deployment diagrams but we are confident that the DIML language can be used to define mappings for other UML diagrams. The mappings we have used for UML in the Coral tool are available in [17]. From these mappings we can see that by using Delegation elements and DIML tree parameterization extensively, we have been able to support all the above mentioned UML diagrams.

The Coral tool supports other user-defined modeling languages and profiles besides UML. We have used DIML to define the concrete syntax of MICAS, a domain-specific modeling language to define peripherals for mobile phones [16]. This example shows that DIML is viable to define the concrete syntax of DSM languages that are different from UML.

All the model figures in this article have been drawn using Coral. It is open source and can be downloaded with the UML to DI mappings from `http://mde.abo.fi/`.

5 Related Work and Conclusions

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 or reconciliation of DI diagrams. This includes the layout of diagrams and the rendering of a diagram to an output device.

We have validated our approach by constructing an experimental tool and exchanging UML models and their diagrams with a commercial modeling tool that supports 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 DI diagrams. We consider such a language important for interoperability reasons and hope that this article will spur further discussion on the topic.

Several authors have addressed the issue of defining the concrete syntax of modeling languages. The Penguins system by Sitt Sen Chok and Kim Marriot [6] is based on the intelligent diagram metaphor and uses constraint multiset grammars to map the concrete syntax of a diagram to the abstract syntax of a model. This differs from DIML where we have a unidirectional mapping from the abstract to the concrete syntax. While the authors show that their approach can be used to define the semantics of a diagram, it is unclear whether a similar approach could be applied in the context of DI. There are several reasons for this, the most important of which is that DI uses Uml1SemanticModelBridges to relate to the abstract syntax and to determine how to render the objects to an image. That is, using DI it is implicit that the abstract model exists prior to a diagram, which is not the case in the Penguins system. Péter Domokos and Dániel Varró [8] use model transformation rules for transforming the abstract syntax into their own language for drawing primitives representing the concrete syntax of models. This approach, however, involves several off-line transformations between intermediate models, which in turn makes diagram reconciliation difficult to achieve. The work by Frédéric Fondement and Thomas Baar [10] formalizes the relationship between abstract and concrete syntaxes with OCL expressions using their own concrete syntax. While the ideas presented are interesting, it does not yet have any tool support and although diagram reconciliation is recognized as a problem, the authors do not offer any solution. In fact, our work addresses some of their concerns on DI.

It can be argued that DIML is simply a specific-purpose model transformation language and that the mapping between models and diagrams can be expressed using existing general-purpose model transformation languages. Many model transformation languages have been developed and researched. Examples are the relational approach by David Akehurst and Stuart Kent [1], and Octavian Patrascoiu's YATL [24], both of which use OCL for the declarative expressions. The relational approach is further investigated by Hausmann and Kent in [11]. There is also a special graph transformation / graph grammar system in VIATRA by Dániel Varró [26], which relies on graph grammars instead of OCL and has operational semantics. Also the MOLA transformation language [12] by Audris Kalnins, Janis Barzdins and Edgars Celms has a graphical imperative programming language with pattern-based transformation rules. Perhaps the

most important general-purpose transformation language is the Query-View-Transform (QVT) [19] language from OMG. None of these technologies are (or should be) limited in which transformations they can accomplish, which makes them more flexible but perhaps harder to understand visually.

It must be noted that we are not proposing that DIML be used as a general-purpose transformation language. There are several limitations in it, but nevertheless we find that a domain-specific transformation language can still bring benefits. It might be easier for users of the transformation language to understand and use, and it certainly is easier to define the transformation rules, although it is clear that an implementation might wish to use its underlying general-purpose transformation technology and display a simplified version (i.e. the mappings shown here) to the user. We firmly believe that there should and will be different transformation languages for models, just as there are different transformation languages for text files, such as `sed`, `awk` and `perl`.

We have not found transformation technologies that specifically address transforming between abstract and concrete models using the DI standard. This is unfortunate because it also makes comparison more difficult as the differences between the diagram languages themselves must be taken into account. Otherwise, in [5, 15], Audris Kalnins et al. show a diagram definition facility which extends the presentational metamodel for every concept that needs to be displayed from the abstract metamodel. This seems complicated in light of the DI standard which is a static metamodel. Additionally there is no explanation on how to declare restrictions on the mappings, which we have solved using OCL expressions, and abstract elements seem to simply map to exactly one concrete element, which is not true for DI.

Acknowledgments

Marcus Alanen would like to acknowledge the financial support of the Nokia Foundation.

References

1. D. H. Akehurst and S. Kent. A Relational Approach to Defining Transformations in a Metamodel. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *Proc. UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany*, volume 2460 of *LNCS*, pages 243–258. Springer, 2002.
2. Marcus Alanen, Torbjörn Lundkvist, and Ivan Porres. Reconciling Diagrams After Executing Model Transformations. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC 2006)*, Dijon, France, April 2006.
3. 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.
4. J. Bézivin. On the Unification Power of Models. *Springer Journal on Software and Systems Modeling*, 3(4), 2004.
5. Edgars Celms, Audris Kalnins, and Lelde Lace. Diagram Definition Facilities Based on Metamodel Mappings, October 2003. Invited talk at the Third OOPSLA Workshop on Domain-Specific Modeling.

6. Sitt Sen Chok and Kim Marriott. Automatic Generation of Intelligent Diagram Editors. *ACM Transactions Computer-Human Interaction*, 10(3):244–276, 2003.
7. 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.
8. Péter Domokos and Dániel Varró. An Open Visualization Framework for Metamodel-Based Modeling Languages. In Tom Mens, Andy Schürr, and Gabriele Taentzer, editors, *Proc. GraBaTs 2002, International Workshop on Graph-Based Tools*, volume 72 of *ENTCS*, pages 78–87, Barcelona, Spain, October 7–8 2002. Elsevier.
9. Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer. Towards Graph Transformation Based Generation of Visual Editors Using Eclipse. *Electronic Notes in Theoretical Computer Science*, 127(4):127–143, 2005.
10. Frédéric Fondement and Thomas Baar. Making Metamodels Aware of Concrete Syntax. In *European Conference on Model Driven Architecture (ECMDA)*, volume 3748 of *LNCS*, pages 190 – 204, 2005.
11. Jan Hendrik Hausmann and Stuart Kent. Visualizing model mappings in UML. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 169–178, New York, NY, USA, 2003. ACM Press.
12. Audris Kalnins, Janis Barzdins, and Edgars Celms. Basics of Model Transformation Language MOLA. In *Workshop on Model Transformation and Execution in the Context of MDA (ECOOP 2004)*, June 2004.
13. Paul Klint, Ralf Lämmel, and Chris Verhoef. Towards an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.
14. Oliver Köth and Mark Minas. Structure, Abstraction, and Direct Manipulation in Diagram Editors. *LNCS*, 2317:290–304, 2002.
15. Lelde Lace, Edgars Celms, and Audris Kalnins. Diagram Definition Facilities in a Generic Modeling Tool. In *International Conference on Modelling and Simulation of Business systems*, pages 220–224, 2003.
16. Johan Lilius, Tomas Lillqvist, Torbjörn Lundkvist, Ian Oliver, Ivan Porres, Kim Sandström, Glenn Sveholm, and Asim Pervez Zaka. An Architecture Exploration Environment for System on Chip Design. *Nordic Journal of Computing*, 2006. To appear.
17. Torbjörn Lundkvist. Diagram Reconciliation and Interchange in a Modeling Tool. Master's Thesis in Computer Science, Department of Computer Science, Åbo Akademi University, Turku, Finland, November 2005.
18. K. Marriot and B. Meyer. *Visual Language Theory*. Springer, 1998.
19. OMG. MOF 2.0 Query / Views / Transformations RFP. OMG Document ad/02-04-10. Available at www.omg.org, 2002.
20. OMG. MOF 2.0 Core Final Adopted Specification, October 2003. Document ptc/03-10-04. Available at `http://www.omg.org/`.
21. OMG. UML 2.0 Infrastructure Specification, September 2003. Document ptc/03-09-15. Available at `http://www.omg.org/`.
22. OMG. UML 2.0 Superstructure Specification, August 2003. Document ptc/03-08-02, available at `http://www.omg.org/`.
23. OMG. Unified Modeling Language: Diagram Interchange version 2.0, June 2005. OMG document ptc/05-06-04. Available at `http://www.omg.org`.
24. Octavian Patrascoiu. YATL:Yet Another Transformation Language. In *Proceedings of the 1st European MDA Workshop, MDA-IA*, pages 83–90. University of Twente, the Nederlands, January 2004.
25. Guus Ramackers. OMG issue 7663. `http://www.omg.org/issues/issue7663.txt`.
26. Dániel Varró. Automatic Program Generation for and by Model Transformation Systems. In Hans-Jörg Kreowski and Peter Knirsch, editors, *Proc. AGT 2002: Workshop on Applied Graph Transformation*, pages 161–173, Grenoble, France, April 12–13 2002.