Creating and reconciling diagrams after executing model transformations

Marcus Alanen, Torbjörn Lundkvist*, Ivan Porres

TUCS Turku Centre for Computer Science, Department of Information Technologies, Åbo Akademi University, Joukahaisenkatu 3-5 A, FIN-20520 Turku, Finland

Received 15 August 2006; received in revised form 13 April 2007; accepted 14 May 2007
Available online 30 June 2007

Abstract

In this paper we discuss how to create and update diagrams after the execution of a model transformation. This is achieved by creating an independent diagram reconciliation tool component that is based on a mapping language from the abstract syntax to the concrete syntax of a modeling language. This approach allows us to decouple model transformation from diagram handling in model transformation languages and tools.

Keywords: Model transformation; Visual languages; Diagram Interchange; DI

1. Introduction

To fully realize the Model Driven Engineering [30] vision we need to define modeling languages and model transformation languages rigorously and we need to provide software development tools supporting them. To ensure interoperability, long term availability and support, these languages and tools should support accepted standards. Software modeling languages are often based on visual notation since this brings important benefits to software development [26]. As a consequence, model transformation languages and model transformation tools need to support in one way or another visual notation.

The Object Management Group (OMG) maintains a series of modeling standards such as the Unified Modeling Language (UML) that are widely used by the industry and studied by the academia. One of the main characteristics of the technical space [9] defined by the OMG modeling standards is that the abstract syntax and concrete syntax of a model are two different artifacts that are defined and maintained independently. The abstract syntax of a language can be defined using the Meta Object Facility (MOF) [43] and the UML 2.0 Infrastructure [45]. To represent the concrete syntax or diagrams of a model, the OMG provides a standard to interchange two-dimensional diagrams called Diagram Interchange (DI) [42]. DI is a language that has been defined following the same metamodeling approach as MOF and the UML. However, DI is not specific to UML. It can be used to represent UML diagrams but also diagrams for domain-specific modeling languages as well.

* Corresponding author. Tel.: +358 405428990.
E-mail address: torbjorn.lundkvist@abo.fi (T. Lundkvist).

0167-6423/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2007.05.002
DI is a key standard to allow the interchange of models between tools that need to represent, create or transform diagrams. Examples of these tools range from simple diagram viewers to full-featured interactive model editors or model transformation tools.

However, the existing OMG standards do not specify how to define the relation between the abstract syntax of a language (given for example as a MOF model) and its concrete syntax (given as a DI model). To remedy this situation, the OMG has recently published a request for proposals for a 'model view to diagram' language [44]. We consider that it is also important to consider how this model to diagram language will be used in the context of a model-driven engineering tool.

In this article, we study this problem in detail: How to create and update diagrams after executing a model transformation. This article is a thoroughly reviewed and improved version of two articles previously published by the authors describing the basic diagram mapping language [2] and the reconciliation algorithms [3].

We assume a tool setting as shown in Fig. 1: model transformations are executed by a generic transformation component that updates the semantic information in a model based on a transformation definition. Once the transformation is completed, a generic diagram reconciliation component updates the diagrammatic representation of a model based on a diagram definition. A key requirement for these tool components is to follow the existing OMG standards and to be able to interoperate with existing modeling tools.

1.1. The diagram reconciliation component

The diagram reconciliation component can create new diagrams or update existing ones. New diagrams may be created if there were no previous diagrams in the model, for example the transformation component is actually a reverse engineering tool that has created a model from code, or when the transformation maps a model from one language to another. On the other hand, a transformation component may sometimes perform a partial change in a model, where only a subset of the existing elements are updated, added or deleted. In this case, the diagram reconciliation component should try to preserve as much information from existing diagrams such as layout, colors and fonts as possible. That is, the diagram reconciliation component should work incrementally, performing the minimum set of updates necessary to maintain consistency of existing diagrams with the abstract models.

In our approach we consider the transformation definition and the diagram definition to be two different artifacts that can be defined and maintained independently. The same applies to the transformation and the diagram reconciliation components: they can run in the context of an integrated modeling environment or they could be completely different tools.

We consider that there are several important benefits in this approach. First, the construction of new transformation tools and the definition of new transformations is simplified since they do not need to deal with diagrams. Second, it is possible to create different diagrammatic representations from the same semantic information. Also, it allows a market of independent tool components to transform and update models and diagrams. Finally, the tool component in charge of diagram handling can be optimized to its specific task and therefore it can be more efficient than a generic transformation engine.
We proceed as follows: First, we briefly discuss how models and diagrams are organized according to the OMG standards in Section 2. In Section 3, we describe a language to define mappings between models and DI diagrams. In the following two sections, we describe how these mappings can be used to construct a diagram creation and reconciliation component for new and existing diagrams. We discuss our experiences in implementing this approach in a tool in Section 6. Finally, we conclude with a discussion on related work in Section 7 and final remarks in Section 8.

2. Models and diagrams

The OMG modeling standards are based on the concept of metamodeling. A modeling language is defined using yet another model, called a metamodel. A language used to define metamodels is then called a metamodeling language.

The Meta Object Facility 2.0 (MOF) [43], the Eclipse Modeling Framework (EMF) [22,13] and the Graph eXchange Language (GXL) Metaschema [51] are well known and widely used metamodeling languages. These languages are rather similar [1], since they are inspired by the object-oriented software paradigm. A modeling language is defined as a collection of classes, each class containing a number of elementary properties or attributes. As an example, the UML metamodel contains more than 100 metaclasses such as Actor, Class, Association or State which describe the concepts that are familiar to UML practitioners. Properties may have type, multiplicity and aggregation characteristics. Finally, classes can inherit properties from other classes. A fragment of the UML metamodel describing state machines is shown in Fig. 2. It must be noted that we have simplified the metamodel for the purposes of this article.

A complete discussion of modeling languages is out of the scope of this article but has been studied elsewhere. Thomas Baar has defined the CINV language [6] using a set-theoretic approach. José Álvarez, Andy Evans and Paul Sammut describe a static object-oriented metacircular modeling language in [5], and the Metamodeling Language Calculus [18] by Clark, Evans and Kent is another very sophisticated one. Also, Nytun, Prinz and Kunert present in [39] a modeling framework in which all model layers are represented uniformly. However, from the point of view of this article, all these approaches exhibit these two important features:

- Classes and instances: The type of a model element is defined as a class that can inherit the properties of other classes. Each element has one single type that defines all its possible slots. Each slot always belongs to one element.
- Separation of abstract and concrete syntax: Models as such only contain semantic information but not how to represent it diagrammatically. A different language is used to define the visual appearance of a model. Therefore, a model and its diagram(s) are different artifacts maintained independently.
We consider that the work described in this paper can be used in any modeling framework that exhibits these two features. In the rest of this section we briefly discuss the language proposed by the OMG to define the concrete syntax of a model.

2.1. The diagram interchange metamodel

The purpose of the OMG Diagram Interchange is to allow the diagrammatic representation of concepts in a model. DI is a rather small language with only 22 metaclasses; a relevant subset of them is shown in Fig. 3.

There are basically four main concepts in DI: GraphNode, GraphEdge, GraphConnector 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 GraphConnector is used as an anchor point for an edge. Nodes, edges and connectors have different properties to define the position and dimensions in a two-dimensional space. It is also possible to define other features such as colors or fonts to be used to render these elements.

Finally, 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 [42].

Fig. 4 shows an example of a fragment of a UML model. The left part of the figure is an object diagram showing a simple state and its diagrammatic representation in DI. On the right hand side of the figure the same model fragment is shown, rendered as an image. The dashed arrows in the figure represents how each individual node in the DI model is rendered. 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 rounded corners. In this case the image was rendered by the tool using Encapsulated Postscript. However, it is also possible to render the image in other graphical formats, such as SVG.

From the object diagram we can see that this DI model contains elements necessary for displaying and laying out 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 the notation of this object diagram is not defined in the UML standard, it is useful for the purposes of this article.
Fig. 4. The rendering of a DI model to an image. The dashed arrows show how each GraphNode is represented in the rendered diagram.

Fig. 5. (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.

In Fig. 5 a larger example of a fragment of a UML model is shown. The top part of the figure is a simple UML statemachine model with two states connected with one transition, presented as a UML object diagram. The bottom part of the figure shows the same DI model rendered as an image. This example shows the states represented as nodes and the transition represented as an edge containing nodes in the same DI Diagram. From this figure we can also see that DI uses GraphConnectors for connecting the endpoints of an edge to other DI elements. Although the GraphConnectors are not visible in the rendered diagram, they are important for layouting the edges.
3. From models to diagrams

We have seen in the previous examples that the DI provides us with the basic metaclasses that can be combined to create diagrams. However, neither the UML nor DI standard tell us what metaclasses we should use to create a specific diagram to represent a specific model (Appendix C of [42] does not provide adequate information for either UML 1.4 or UML 2.0). As we have seen in the example, this task is not trivial since each UML model element is represents using many DI elements and the mapping between the model element and its diagram representation cannot immediately be derived. 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 be able to create a truly reusable and independent diagram reconciliation component, this component should support different modeling languages and should be based on standards. The first requirement implies that the rules to create or update diagrams for a given modeling language should not be hard-coded into the transformation tool but defined in a tool-independent format that can be loaded by the reconciliation component at run-time. The second requirement implies that the diagrams created or updated should conform to a standard diagram interchange format such as DI. This would allow the user to view and edit transformed models in a DI-compliant model editor. Currently, we are only aware of one commercial tool, Gentleware's Poseidon [25], that supports DI. We consider this tool as a reference implementation of DI.

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. In addition, the DIML language is tuned to deal with the hierarchical structure of DI diagrams. An overview of the DIML language with respect to other modeling languages can be seen in Fig. 6. In this figure, a dashed arrow indicates conformance, and a plain arrow indicates usage. We assume that this mapping language is defined using the 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 models and diagrams.

We can see three example DIML models for UML StateMachines, SimpleStates and Transitions shown in Figs. 7, 8 and 9 respectively. These mappings conform to the simplified structure of StateMachines presented in Fig. 2. 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 element ownership based on guard and selection statements, the former inside brackets. The hierarchy forms a parameterized skeleton which when transformed into DI elements in a specific context gives us the intended result.

An example of the application of these three mappings was seen in Fig. 5. The topmost part of the figure (colored gray) shows a StateMachine with two SimpleStates and one Transition. When the mapping for UML StateMachines (Fig. 7) is applied to the StateMachine, a DI Diagram will be created. When the mapping for UML SimpleStates (Fig. 8) is applied to the SimpleStates and the mapping for UML Transitions (Fig. 9) 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 Fig. 5 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.stereotypeNotEmpty()” in the DIML model returned false and thus no StereotypeCompartment was created.

Thereby the three mappings for StateMachine, SimpleState and Transition from Figs. 7–9 have been used to create several DI tree fragments as outlined by triangles in Fig. 10, yielding the final DI Diagram in Fig. 5. GraphConnectors are used to connect GraphEdges together with other GraphElements, as indicated by the horizontal lines between DI trees.
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 and is therefore tuned for this purpose, 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.

The metamodel for the DIML mapping language is shown in Fig. 11. In the figure, $MOF::\text{Class}$ represents the type of any metaclass, not just UML metaclasses. The $OCL::\text{OclExpression}$ refers to any OCL expression. OCL is a language for creating arbitrary queries on models. It can be used to return a collection of element references from models or to assert that certain properties hold in a model.

The MappingModel is a container to collect the mappings of a given modeling language or profile. Every DIML model must have one MappingModel as its root element. An ElementToDIMapping element $m$ is then a mapping of one abstract element of type $m.\text{element}$ to its diagrammatic representation as DI elements.
In Figs. 7–9, 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 Contained.guard expression, and a text without brackets is a Contained.selection expression.

The slot m.root points to a DIML tree.

### 3.2. DIML Tree

Each mapping rule is basically a tree of parts. Such 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 is accomplished by the Contained elements, and means here that the occurrence and recurrence of child GraphElementParts is determined by the slot values in Contained.guard and Contained.selection.

The guard and selection expressions allow us to create a mapping to DI context-dependent on the abstract model element and all the other abstract model elements as well as the chain of parents in the DI model. These expressions, 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

Fig. 7 shows an example mapping for a diagram. Such a mapping m has a DiagramPart element r in its m.root slot, with r.diagramType denoting what diagram type is being considered (e.g., “ClassDiagram” or “StateDiagram”). The m.contextGuard is evaluated and must return true for a mapping to be valid. The contextGuard is an OCL expression which receives the abstract element and diparent as its parameters. Diparent is the parent element in the DI model. It is guaranteed to exist for anyGraphNode or GraphEdge except for Diagram, which is the root in a DI model and hence has no DI parent. Thus, for diagrams, diparent is always a null pointer/reference. Using diparent, we can query the chain of parents in the DI model. The contextGuard can be used to limit whether or not it is allowed to create a diagram for the given abstract element.

The slots m.validIn and m.acceptsConnector are unused when the root element m.root is a DiagramPart, and must in this case be empty.

### 3.4. Support for GraphNodes and GraphEdges

Figs. 8 and 9 show example mappings for states and transitions. Such a mapping m is otherwise similar to a 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 diparent will now be a valid DI element in the diagram. An example of this can be seen in Fig. 9, where the mapping can only be used if the expression diparent.oclIsKindOf(DI::Diagram) holds, i.e., when the parent DI element is the root Diagram element.

The set m.validIn.diagramType denotes the valid diagram type set, for example {“StateDiagram”, “ActivityDiagram”} for UML Transitions. 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. Again, starting at m.root, the DIML tree can be described.

### 3.5. Correspondence of DIML elements with DI

In a DIML tree, an instance p of DiagramPart, GraphEdgePart or GraphNodePart corresponds to an arbitrary amount of instances of the DI elements Diagram, GraphEdge or GraphNode d, respectively. There is one DI element for each element found when executing the Contained.selection expression.

According to the DI standard, a Diagram has a SimpleSemanticModelElement s in its semanticModel slot such that p.diagramType = s.typeInfo, and a Uml1SemanticModelBridge in its Diagram.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 interpretation of a DIML mapping so far enables us to describe a tree of DI elements. However, a diagram in DI is not merely a tree, but a graph where GraphElements are connected together via GraphEdges. The problem is how to describe which connections are allowed, and which are not, specially when we consider that the same abstract element can appear several times in a diagram. For example, a UML Class can be shown both as a rectangle but also as the type of an attribute or a parameter of an operation. However, only when a class is represented as an independent rectangle can be used to connect Generalization or Association edges. Connecting an Association to the type of an Attribute can be considered valid from a semantic point of view, but it is against the presentation rules of UML class diagrams.

Our solution to this problem requires two properties. A GraphEdgePart \( p \) has a \( p\.connector \) expression. It is evaluated in the context of the corresponding abstract element and receives the GraphEdge \( g \) as an additional parameter. The evaluation 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 \( g \). 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, and is thus usually \text{true} \ or \text{false}.

This schema is required since not all GraphElements may be connected and the only distinguishing mark is the context. In our work, this context is provided by having several ElementToDIMappings for the same abstract metaclass.

### 3.7. Known limitations

There are some known limitations on what kind of diagrams DIML can describe. These limitations are a trade-off between expressiveness of the mapping language and the complexity of the algorithms to implement the mappings.

The first limitation is that the source of a mapping can only be a metaclass, not a property or a relation between metaclasses. Even though properties can appear in the mapping rules, specially in the Contained.selection expressions, they cannot map to the root InitialPart of a DIML tree. This limitation appears quite often in OMG standards. For example, the model interchange format XMI \[40\] can serialize an element and its slots, but it cannot represent individual slots. The consequence is that relationships between elements should be represented as metaclasses. For example, in UML, the generalization relationship is a metaclass, instead of simply properties (such as superclass or subclass).

The second limitation is that the target of a mapping is a single DIML tree instead of multiple DIML trees. Although this multiplicity would be easy to fix in the metamodel, the elements in different trees must also be able to reference each other (as per one of our use cases related to UML AssociationClasses). This would require more thorough changes. However, a simple workaround exists: in the mapping rules, one can include a new graph node that serves as container for all different DIML trees. This new mapping can be processed with the existing algorithms, but it would create superfluous graph nodes that should be rendered transparently.

### 4. Generation of new diagrams

As mentioned earlier, a use of DIML is the automatic creation of a specific diagram from the abstract model. In this section we assume that a new model that does not contain any diagram has been created and we wish to create a specific diagram to represent the model graphically.

This task can be described by a depth-first algorithm, of which an outline is seen in pseudocode in Fig. 12. The starting point is the function \textit{create_diagram}, which takes the abstract model element \( e \) and a diagram type.
string `diagramType` as its formal parameters. Since a diagram is a tree of DI elements with respect to element ownership and has a DI Diagram element as the root, we first need to find a valid `ElementToDIMapping` element `e` where `e.element` points to the corresponding abstract metaclass (MOF::Class). `e.root` points to a DiagramPart where `e.root.diagramType` tells what kind of a diagram the mapping describes. After that, the hierarchy of DI elements is created by recursing in the `create_di` function.

To summarize the algorithm, we can consider that a diagram is created as follows. In `create_di`, the generator follows the mappings given in the DIML model. Here, `e` is an abstract element, `diparent` is either the immediate parent DI GraphElement or the null pointer, and `part` is a DiagramPart, GraphNodePart, GraphEdgeParts or Delegation.

If `part` is a Delegation, we need to find a new mapping for the abstract element. The function `in_suitable_diagram` returns true when the mapping is valid in a specific diagram: for example, a mapping for a UML class is valid in Class diagrams, not in State diagrams. The actual definition of the function is simple and not described further.

Otherwise, we create a corresponding DI element on lines 26–38 and set the SemanticModelBridge: either a Uml1SemanticModelBridge or a SimpleSemanticModelElement. After that, the loop on line 39 is responsible for creating DI elements on one level of the hierarchy, with the recursion occurring on lines 41, 44 and 46. The guard evaluation on line 39 and the selection evaluation on line 43 give the developer of DIML models the flexibility to create a parameterized DI Diagram from abstract model data.
On lines 40–46 we do the following for every Contained element \( c \) in the children slot of the part, for which the \( c \).guard in the context of \( e \) and with diparent as its parameter holds.

- Evaluate \( c \).selection in the context of \( e \) and with diparent as its parameter. The expression must return an OCL collection of abstract elements. If the expression string is empty, it defaults to returning a set consisting of the current element \( e \); this is mainly used for children GraphElementParts with a typeInfo string. For each element \( s \) in the collection, the \( c \).child GraphElementPart is accepted in the context of \( s \) as the abstract element, and \( g \) as its diparent.

- If \( c \).separator is nonempty, 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 by recursing into create_di.

Delegation elements are used to decouple the representation and computation of individual DIML trees. When searching for a new mapping, only one mapping is allowed to be valid. No nondeterminism is allowed. Once a valid mapping is found, DI tree creation can begin again in the context of a new current abstract element and diparent.

Furthermore, the GraphEdge elements should be connected to other GraphNode or GraphEdge elements using GraphConnectors. Since the GraphConnectors are owned by the GraphNode or GraphEdge the new GraphEdge connects to, the creation of new GraphConnectors must occur after all other elements in the diagram have been created. This occurs in the operation attach_connectors, which takes a diagram as its only parameter. This function is called on line 6 in create_diagram. In attach_connectors, for each GraphEdge transitively contained in the diagram, the corresponding GraphEdgePart part from DIML is acquired (preferably retained from create_di). Then, part.connector in the context of the abstract element \( e \) mapped to the edge, returns a sequence of abstract elements. For each of these elements, a corresponding ElementToDIMapping \( n \) is located and \( n \).acceptsConnectors is evaluated. After \( n \) is found, the GraphElement where the GraphEdge should connect is located. The function in_same_diagram tests that both elements are in the same diagram. Finally, a GraphConnector is created to link these DI elements together.

Once all the DI nodes, edges and graph connectors are created they should be arranged using a layout algorithm that is appropriate for the particular type of diagram. Examples of a layout algorithm for class diagrams can be found in [21] and algorithms for statechart diagrams can be found in [16]. We consider this task to be beyond the scope of this paper.

5. Reconciliation of an existing diagram

This section discusses the principal idea of why diagram reconciliation is useful and how it works. We also present an algorithm for performing diagram reconciliation. Finally, we give a short example on modifying the abstract elements in a statemachine and see how diagram reconciliation can update the diagram.

5.1. Principle behind diagram reconciliation

As we have discussed in the introduction, there are situations where we want to preserve as much information from an existing diagram as possible after executing a model transformation. That is, the presence of elements in a diagram, their layout, text fonts and color should not change except when this is motivated by the execution of a transformation. In this case, it is possible to define a diagram reconciliation mechanism that can update existing diagrams while using the same DIML mapping language as before.

In principle, diagram reconciliation is an optimization of generating a new diagram. Technically, we could create a new diagram as described in Section 4 to replace the old diagram, but that would be too slow and some visual details would be lost. Diagram reconciliation should work at an acceptable speed and as if a new diagram had been created with the visual details intact. Since DIML mappings are declarative constructs instead of programs, they do not demand a certain algorithm for performing diagram generation or reconciliation. Instead implementations are free to use any algorithm that can provide a fast and correct solution. So, an advantage of this approach is that it can be much more efficient than generating a new diagram from scratch when only a relatively small part of a model has been changed due to the application of a transformation.
Diagram reconciliation assumes that it is possible to discover what has changed in a model during a transformation. We assumed in the introduction that the reconciliation component has access to the current abstract model and the obsolete diagrams. We now require that it also has access to either the old abstract model or to a change description. This change description can be a list of atomic changes done to the slots of the elements, or a special model reflecting the difference between the old and the new abstract model. We have previously published a model difference algorithm in [4].

Based on this information, the reconciliation component can inspect which abstract elements have changed. Using the Contained.guard and Contained.selection expressions in the DIML mappings, it can then calculate which changes have invalidated which DI elements [15] and then apply the mappings again.

Where order of elements is not important, i.e., where the selection expressions were unordered OCL collections, set operations can be used to calculate which elements should be removed, and which should be added. For ordered collections, there are several algorithms (e.g. [38]) for calculating the edit distance for transforming the old contents into the new contents.

The optimal reconciliation is one which reuses as much as possible from the obsolete diagram to bring it up to date. The level of reuse may depend on how sophisticated the reconciliation system is. For example, consider a UML Generalization connecting two classes, this being represented by a specially colored GraphEdge with a triangular endpoint. If a transformation changes which superclass the generalization points to, it can be argued that a valid reconciliation is to delete the edge and create a new edge in its place connecting to the new superclass. However, information such as color and the exact waypoints are lost. A better reconciliation approach is to reuse the existing GraphEdge and simply move its GraphConnector to the new superclass.

5.2. A diagram reconciliation algorithm

A simplified algorithm for reconciling a diagram after a change has occurred in the abstract model is outlined in the following paragraphs. In this algorithm we assume that a diagram is reconciled in place after each individual change in the abstract model. In addition we assume that it is possible to retrieve historical data of the model, that is, it is possible to find out the previous value of any expression provided it has changed since the diagram was reconciled. Reconciliation can then be performed by recording the changes made in the abstract model and thereafter calling the reconciliation component with the detected change as parameter.

```
1   function reconcile(element, slot):
2       find all selection or guard or connector expressions for which element and slot is part of
3       for each mapping m that these expressions are composed of:
4           for all diagram elements di mapped to m.root:
5               e ← di.semanticModel.element
6               reconcile_di(e, m.root, di)
```

The starting point of the algorithm is the function `reconcile`, which takes an abstract model element `element` and a `slot` containing a name of the slot, such as “subvertex” for an instance of a CompositeState. Here, `slot` is always a member of `element`. For a pair of `element` and `slot`, all corresponding selection, guard and connector expressions that might have to be recalculated due to the change are retrieved. Then, for each ElementToDIMapping `m` composed of the selection and guard expressions, the algorithm iterates over all diagram elements `di` mapped to the `m.root` Part and retrieves the abstract element instance `e` associated with the diagram element. It must be noted that `e` is always an instance of the abstract metaclass in `m.element`. Due to the fact that the expressions can be of arbitrary complexity, `e` and `element` are not necessarily equal. Finally, in the recursive function `reconcile_di`, the corresponding hierarchy of DIML elements starting from `m.root` is compared to the hierarchy of `di`. In this function, all selection, guard and connector expressions are re-evaluated in the context of the abstract model element `e` and `di`. Consequently, if a change has occurred in the abstract model that invalidates the diagram, the corresponding parts of the diagram are brought up to date.

The reconciliation of an individual DI subtree to the corresponding abstract model occurs in the function `reconcile_di`. Depending on the structure of DIML mapping, the abstract model and the existing DI subtree, different strategies for bringing the diagram up to date are used. These cases are outlined using three functions, `reconcile_connectors`, `reconcile_part` and `reconcile_delegation`. 
Function reconcile_di(e, parentpart, diparent):  
  if parentpart is a GraphEdgePart:  
    reconcile_connectors(parentpart, e, diparent)  
  for c in parentpart.children:  
    if c.child is a ConcretePart:  
      reconcile_part(c, e, diparent)  
    else:  
      reconcile_delegation(c, e, diparent)

For a GraphEdgePart and correspondingly a GraphEdge, reconciliation of the GraphConnectors is done on line 9 in function reconcile_connectors, defined on lines 15–32. This function takes a GraphEdgePart part, an abstract element element and the corresponding GraphEdge edge as parameters. On lines 16–17 the current value of the connector expression is retrieved from the abstract model and the previous values from a change description. These values are used to determine which diagram elements the both ends of the edge should connect to. Furthermore, the values are compared for changes. If a change has occurred, the GraphElement the edge should be connected to is found on lines 20–26. If a GraphConnector exists for the respective end, it is moved to the correct GraphElement on line 28. Otherwise a new Graphconnector is created on lines 30–32.

The hierarchy of DIML elements are further traversed on lines 10–14. For ConcreteParts the function reconcile_part is called. Here, the guard expressions in contained are evaluated in the context of the abstract element element and diparent, passed as parameters to the function. If the guard holds, a corresponding GraphElement needs to be created if it does not exist. If the guard does not hold, a GraphElement is not created, and will be removed if it is found in the diagram. This occurs on lines 33–47. Since a guard affects all transitively contained GraphElements, recursion only occurs by calling reconcile_di again on lines 33 and 39 where a guard holds.

For Delegation elements the function reconcile_delegation is called on line 14. This function is defined on lines 48–66, and takes the same parameters as the function reconcile_part. The reconciliation of Delegations involves creating and inserting DI subtrees for abstract elements that are not mapped in the diagram and removal of previously mapped DI subtrees that are removed from the abstract model. This is done by retrieving the current value of the selection expression and the previous value on lines 49 and 50. Using difference calculation, it is possible to determine which abstract elements that should be mapped to the diagram, and which to remove from the diagram. On lines 51–56 the elements deleted from the abstract model are removed from the diagram, along with possible separators that are invalidated. The creation of new DI subtrees for previously unmapped abstract elements occurs on lines 57–66. Here, the test on line 59 avoids the creation of subtrees of already mapped abstract elements. Furthermore, if a GraphEdge
is created, new GraphConnectors are created on line 63. Finally, possible separators are inserted between each of the created subtrees on lines 64–66.

```python
48 function reconcile_delegation(contained, element, diparent)
49     new_am ← contained.selection(element)
50     old_am ← retrieve previous value of selection
51     for each element in new_am and not in old_am:
52         remove corresponding diagram element from diparent.contained
53         if diagram element is a GraphEdge:
54             remove GraphConnectors connected to the edge
55         if contained.separator:
56             remove one corresponding separator if this is not the last element
57     for each element n in new_am:
58         find corresponding diagram element d to n in diparent.contained
59         if not d and n not in old_am:
60             d ← create_diaselem, diparent, part
61             insert d into diparent.contained as specified by diml rule
62             if d is a GraphEdge:
63                 create new GraphConnectors for d
64                 if contained.separator and not the last element:
65                     s ← create_diaselem, diparent, c.separator
66                 insert separator s into diparent.contained
```

It must be noted that the reconciliation algorithm presented here is very high level, and intended only as an example. It does not include details on how calculate the correct insertion points for new DI subtrees created by the algorithm, for example with respect to the ordering of abstract elements in selection expressions, or how to determine correspondence of a DIML part to a DI diagram element. These issues need to be addressed in an implementation of the algorithm. Due to the fact that DIML mappings are declarative constructs instead of transformation programs, reconciliation of diagrams after changes in the abstract model is not bound to any particular algorithm, and thus leaves room for competing approaches and optimization. The algorithm presented here reconciles the diagram after one single change in the abstract model. Larger transformations involving several abstract elements can be decomposed into a list of several changes made to individual abstract elements and slots, allowing the presented algorithm to be used also for larger transformation programs.

In the next section we will show some diagram reconciliation examples from UML state machines.

### 5.3. Reconciliation example

As an example, let us assume we introduce a new state into the model in Fig. 5, by inserting a new SimpleState (S3) into the StateMachine.subvertex slot. This invalidates the previously valid diagram since the new state is not represented.

Using a change description or by calculating a model difference [4], the reconciliation component determines that there has been a change in the subvertex slot of the StateMachine, and finds the corresponding ElementToDIMapping for it, that is shown in Fig. 7. Then it calculates the difference of the old self.subvertex and the new self.subvertex. The ElementToDIMapping for StateMachines states that new SimpleStates are inserted in the diagram owned by the StateMachine. The reconciliation component determines that the ElementToDIMapping shown in Fig. 8 should be used to create a new DI representation of the SimpleState. A new representation of the SimpleState is created and inserted accordingly in the diagram. The resulting UML and DI model along with its visual representation is shown in Fig. 13.

Next, assume the StateMachine in Fig. 13 is modified by changing the target of the Transition from state S2 to S3. This is achieved by assigning self.target := S3 for the Transition, assuming the link to S2 is thus lost. Again, the reconciliation component determines there has been a change in the target slot of the Transition, and finds the corresponding ElementToDIMapping for the Transition, shown in Fig. 9. The ElementToDIMapping states that Transitions are represented as GraphEdges in diagrams and connect to states such, that the first endpoint is represented by self.source and the second endpoint by self.target. As a result, the reconciliation component connects the Transition in the diagram to the GraphNode representing the SimpleState S3. The corresponding UML and DI model along with its visual representation is shown in Fig. 14.

Finally, assume the StateMachine in Fig. 13 is modified by removing the state S3 from the statemachine. This can be done by executing self.subvertex.remove(S3) for the statemachine. In addition, the transition T1 is removed from the statemachine by executing self.transitions.remove(T1). Consequently, the reconciliation component determines there has been a change in the subvertex and the transitions slot of the statemachine, and finds the corresponding
Fig. 13. (Top) UML model in gray after adding a third SimpleState and its diagram representation in DI. (Bottom) DI diagram rendered using the UML concrete syntax.

ElementToDIMapping for StateMachine, shown in Fig. 7. An evaluation of the selection expression `self.subvertex` and comparison to the previous value of the expression is used to determine that there is a SimpleState S3 mapped to the diagram, which will be removed from the diagram. Similarly, a comparison of the selection expression `self.transitions` to its previous value determines that the Transition T1 and its GraphConnectors should be removed from the diagram. The resulting UML and DI model along with its visual representation is shown in Fig. 15.

It might be argued that the transition was already invalidated by removing S3 from the statemachine, due to the fact that the transition’s target was empty, and this had left an edge with only one end connected in the diagram. While a modeling tool might decide to delete a transition when a source or a target is deleted from the abstract model, this is beyond scope of the diagram reconciliation component, as the diagram reconciliation component only modifies or creates diagrams based on given changes in the abstract model. That is, the diagram reconciliation component will not change the structure of the abstract model. A component that enforces well-formedness checks and performs
corrections on the abstract model, for example [36], can be regarded as a model transformation component and hence the diagram reconciliation component is executed after the model transformation component has finished.

6. Implementation

In this section we discuss some aspects related to the practical implementation and validation of a diagram reconciliation component.

6.1. Optimization of query expressions

A great deal of the flexibility of DIML comes from the use of OCL expressions. Using these expressions, DIML models can navigate through the abstract model and select the relevant subset of model elements that will be presented
in a diagram. However, this flexibility also leads to complex expressions, which further leads to a cost with respect to reconciliation: it is not apparent which changes in the abstract model will trigger changes in the diagram.

Thus, if we were to use the OCL expressions of DIML, we need to be able to quickly tell which values of OCL expressions have changed when the abstract model is modified; this is not easy. Furthermore, OCL allows computationally very complex expressions, although in [35] it is said to not be Turing complete.

The DIML metamodel uses OCL in five different properties. Since we wish to create an efficient reconciliation system, expressions that can take a huge amount of computational time are not desired. Our solution has been to use a restricted subset of OCL in some of the DIML properties. This OCL subset should be expressive enough to define the DIML mappings for modeling languages as complex as UML but it should also be possible to evaluate quickly.

The restricted language in the Contained.selection property only accepts expressions based on these patterns:

- `self.x`
- `self.x.select(e | e.oclIsKindOf(z))`
- `self.x.select(e | e.v = w)`
- `self.x.select(e | e.v ≠ w)`
- `self.x.select(e | e.oclIsKindOf(z) ∧ e.v = w)`
- `self.x.select(e | e.oclIsKindOf(z) ∧ e.v ≠ w)`.

Basically, this restricted language can navigate one slot away from the current element, and optionally filter out elements which are not of the correct type (or a subclass of that type). It can also be used to check primitive values for equality or inequality. This enables us to use the following expressions, for arbitrary property names `x`, `y` and `v`, metaclass `z`, and value `w`.

The GraphEdgePart.connector property can handle expressions either one or two slots away. That is, expressions such as `self.x` and `self.x.y` can be described. For example, the DIML mapping for Transitions uses the expressions `[self.source, self.target]` whereas the DIML mapping for Associations uses the expressions `[self.connection[0].participant, self.connection[1].participant]`.

At the moment, we do not restrict the Contained.guard expression language. In our implementation, these expressions are arbitrary programs returning true or false. This is a drawback in the implementation and makes it
impossible to determine when a change in the abstract model triggers a change in the result of a guard evaluation. A more sophisticated implementation would do similar optimizations as we have done, but behind the scenes, and also support more complex OCL expressions. However, implementing an efficient OCL interpreter can be a daunting task since determining quickly which OCL expressions need to be revalidated is not easy. There have been advancements in this area by Cabot and Teniente [14,15].

We consider that the simplified language serves the purposes we have outlined in Sections 1 and 2, but we acknowledge that the language proposed in this paper is more general. Even with these restrictions, the optimized query language has shown to be adequate in expressions for UML diagrams. It makes it easier to determine which parts of the diagram need to be updated and therefore it has enabled us to perform reconciliation of models and diagrams using algorithms of low computational complexity, 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.

6.2. Implementation in the Coral tool

We have implemented DIML in an experimental modeling tool called Coral. Coral is distributed as open source available at http://mde.abo.fi/tools/Coral. It is written in Python with a C++ core for handling models. It has a graphical user interface for metamodel-independent model management and different graph transformation mechanisms. Coral is not a replacement for any existing modeling tool but a demonstrator of different research ideas.

We have implemented a component for Coral that reconciles models and diagrams after executing model transformations or performing editing operations. Our implementation uses DIML mappings with the restricted expression language described in Section 6.1. Also, our implementation uses the Python language to define unrestricted OCL expressions such as the Contained.guard or ElementToDIMapping.contextGuard expressions. This is due to the fact that Coral lacks a complete OCL interpreter.

Our implementation is based on a model manager that automatically collects model changes into a transaction as described in [47]. A transaction is then of a list of individual atomic commands, such as “insert element e into slot s at position i” or “create new element of type t”. The diagram reconciliation component is invoked by the model manager at the end of a transaction so it can update all the necessary diagrams.

Coral can edit and transform models using an interactive graphical editor but also using different model transformation engines such as imperative programs written in Python [47] or graph transformations based on the double-push out approach [48]. Actually, the diagram reconciliation component does not distinguish between these two cases and it performs the same task independently if a model has been edited interactively by the user or transformed by a program.

Although most of Coral is written in an interpreted language which does not include a just-in-time compiler or similar feature, the model editor is nevertheless fast enough for interactive editing of models using a low-end desktop computer. Most of the time is in fact spent by the various change propagation components, and the graphical updates; the diagram reconciliation is a very small part of the computational time required.

6.3. Validation

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 1.4 in the Coral tool are available in [34]. From these mappings we can see that by using Delegation elements and DIML tree parameterization extensively, we have been able to support all the UML diagrams mentioned above. However, some constructs have not been possible to describe due to the known limitations from Section 3.7: the links between Comments and other elements, and AssociationClasses in class diagrams. The former maps a relation from the abstract model to the diagram, and the latter requires a complicated multirooted tree of DI elements. We plan to improve on DIML to overcome these limitations.

The Coral tool supports other user-defined modeling languages and profiles besides UML. We have used DIML to define the concrete syntax of MICAS [33], a domain-specific modeling language to define peripherals for mobile phones, and the concrete syntax of SOCOS [7], a domain-specific modeling language to define refinement diagrams. In addition, we have used DIML to define the concrete syntax of DIML itself. These examples show that DIML is
viable to define the concrete syntax of DSM languages that are different from UML, and that DIML does not require UML.

We have also assessed compatibility with Gentleware Poseidon version 3.0 from which the DIML mappings presented in this article are based. In Fig. 16 we see a model created in Poseidon. In Fig. 17, the exact same file has been loaded into Coral. The composite state is selected and the follow command is executed in the Coral shell:

```
self.subvertex.insert(coral.lang.UML14.SimpleState(name="SS1"))
```

The variable `self` refers in Coral to the current selection. Therefore this command creates a new state in the selected composite state. Then, the diagram reconciliation component notices the new state in the abstract model, and modifies the diagram accordingly. Finally, the model is saved again (into XMI [40]) and loaded back into Poseidon. As can be seen from the result in Fig. 18, it is (a) possible to load the file from Coral into Poseidon, and (b) the diagram information is compatible with the diagram information that Poseidon expects. The conclusion is that DIML mappings (for the UML 1.4 statecharts) as used by Coral work according to the DI reference implementation in Poseidon.¹

7. Related work

The work presented in this article is related to general purpose model transformation languages, languages specific to diagram definition and diagram editors and development environments that support user-defined visual languages.

7.1. Diagram editors

There exists full-featured metamodel-based editors that allow the user to create and edit models in user-defined languages. Examples of these approaches are AToM³ [20], MetaEdit+ [29], Pounamu [52]. These tools are complete environments while our work describes only one component that should work with any other editing and transformation component based on the OMG standards.

7.2. General purpose transformation languages

Many researchers have studied the definition of new model transformation languages and tools that support in one way or another the OMG modeling standards. Among the general purpose transformation languages are the relational approaches by Akehurst and Kent [19] and YATL [46] by Patrascu, both of which use OCL for the declarative expressions. The relational approach is further investigated by Hausmann and Kent in [27]. There is also

¹ We should note that newer versions of Poseidon have changed their internal DI mappings. Therefore, the current version Coral is not compatible with diagrams generated by newer Poseidon versions. However, this does not affect the actual diagram reconciliation component. In order to support the new Poseidon diagrams we should only update the DIML mappings.
a special graph grammar system in VIATRA by Varró [50], which relies on graph grammars instead of OCL and has operational semantics. Jean Bézivin proposes the Atlas Transformation Language [11], which has tool support in the Eclipse Modeling Framework [22]. The MOLA transformation language by Kalnins, Barzdins and Celms [28] has a graphical imperative programming language with pattern-based transformation rules. Perhaps the most important general purpose transformation language is the QVT language [41] from OMG. Also, the work presented in this article can be seen as a specific instance of a model composition framework, as described in [10].

However most of the existing transformation languages and applications on the previous tools are based on the abstract syntax or metamodel of a modeling language. Therefore, they do not deal with diagrams as such. A transformation definition could include rules to update the diagrams associated to a model, since, in fact, diagrams
are just organized as another model. We consider that this approach is not satisfactory due to the fact that diagram transformations are rather complex but independent of the semantic transformations and, therefore, they can be reused from one model transformation to another. That is, it is possible to define rules to create and update diagrams that are independent of the actual model transformations.

We have decided to use a special language to define the diagram mappings and a special tool to apply these mappings due to the need for specific algorithms to create and reconcile existing diagrams. The diagram mapping language and corresponding tool support should work preserving as much information as possible from existing diagrams. Also, we consider that DIML mappings are more succinct than the equivalent transformation in many general transformation languages, due to the use of OCL queries and expressions to describe many cases in a single rule. We also use special purpose DIML Delegation elements which strongly separate different diagram mappings, notably from that of the parent mapping.

An alternative approach to implement the work presented in this article mappings is to use a general model composition framework [10] such as the Atlas Model Weaver [23]. In this case, a composition framework would take one or two user models and the DIML mappings as input models and recreate the necessary diagrams by implementing the algorithms presented in this paper using transformation rules that are specific to the composition framework. It would be an interesting and worthwhile experiment to compare different model composition frameworks with DIML. There are several properties that should be analyzed, such as the size of the mapping definitions, understandability, maintainability, any expressivity restrictions and the need for traceability mechanisms and separate tracing models.

Model composition and weaving frameworks are being researched and developed at a fast pace, for example [23, 37, 8, 31]. Similarly, there is active research in model traceability [12, 32].

### 7.3. Diagram definition languages

There are other approaches and tools that support the reconciliation of abstract models and their concrete models. However, none of them seem to support DI. This makes comparison awkward, since one of our goals has been the usage of OMG standards. The diagram definition facility [17] by Celms, Kalnins and Lace specifically targets mapping to diagrams and uses its own diagram metamodel, although one different from DI in that it requires subclassing for each diagram element, whereas DI metaclasses are not intended to be inherited from. The work by Fondement and Baar [24] 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.

### 8. Conclusions

We have described an approach for creating new diagrams from the abstract syntax of a model and for bringing existing diagrams up to date after the execution of a transformation that updates the abstract syntax of a model. We consider that these problems are important because they are necessary to allow the automatic transformation of visual diagrammatic languages.

Our approach is based on a mapping language called DIML that describes the relation between the abstract syntax and the concrete syntax of a model defined according to the OMG standards. Our solution enables the creators of model transformation languages and tools to ignore diagrams and focus on the semantic information stored in models. One of the most important characteristics of DIML is that it is independent of the modeling and transformation language. This allows us to define metamodel-independent algorithms to create and update diagrams. On the other hand, DIML is specific to DI, the OMG diagram interchange standard. Other alternative diagram interchange languages are discussed in [49].

There are several practical benefits to reconciliation and using DIML. We realize that we have made several engineering decisions in creating a domain-specific weaving metamodel, and that it has limitations. At this time we do not know for certain to what extent the decisions are a hindrance, and how much they are beneficial. For example, by only allowing OCL expressions to query the chain of parent DI elements we establish that DIML rules are hierarchical. This makes the algorithms relatively simple, and we do not have to, for example, calculate any graph isomorphisms.
There are also some limitations in our work as described in Section 3.7, the most important limitation being the fact that a DIML rule cannot map relations, only a single metaclass.

We have validated our approach by implementing it in an open source modeling tool called Coral. Due to the common interchange format and adherence to OMG standards we can use both Poseidon and Coral to edit, transform and interchange models and diagrams. We believe that this is an indication of the viability of DI and the DIML language.

DI and the diagram reconciliation component do not contain all the functionality necessary for a full interactive editor since they lack a user interface, a layout and a rendering engine. We assumed that these problems are solved by other independent components. In some visual languages, such as UML class diagrams, the layout is a question of aesthetics. However, in other languages, such as UML sequence diagrams, the layout also conveys semantic information since the passage of time is represented in the vertical axis of a diagram. Therefore, we would like to extend the DI and the DIML language to include diagram layout constraints so that the layouting step can be integrated with the diagram reconciliation step.

Acknowledgments

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

References


[34] 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.


[38] Eugene W. Myers, An O(ND) difference algorithm and its variants, Algorithmica 1 (2) (1986) 251–266.


