Diagram Interchange for UML

Marko Boger[1], Mario Jeckle[2], Stefan Mueller[3], and Jens Fransson[3]

[1] Gentleware AG,
Vogt-Koelln-Str. 30,
22527 Hamburg, Germany
marko.boger@gentleware.de
http://www.gentleware.de
[2] DaimlerChrysler Research and Technology,
Wilhelm-Runge-Str. 11,
89013 Ulm, Germany
mario.jeckle@daimlerchrysler.com
http://www.jeckle.de
[3] University of Hamburg,
Vogt-Koeln-Str. 30,
22527 Hamburg, Germany
stefan@stmonline.de, jfransson@gmx.de

Abstract. XMI is a standardized mechanism for exchanging UML models. However, this mechanism does not sufficiently fulfill the goal of a model interchange: it does not include the exchange of diagram information. XMI as defined for UML 1.x is only capable of transporting information on the elements in an UML model but not information as to how these elements are represented and laid out in diagrams.
This paper proposes an extension to the UML metamodel to represent diagram information in a graph-oriented manner. The approach presented is able to fix the deficiency for UML 1.x and solve the problem for UML 2.0. The approach was handed in for standardization to the OMG in response to the Diagram Interchange RFP.

1 Introduction

The Unified Modeling Language (UML) [8] is a modeling language for object-oriented software systems with a strong emphasis on graphical representation. It is employed throughout the software development process, and a wide variety of different kinds of tools can be utilized during this process. Tools vary greatly from those geared to design the diagrams or to check the consistency of models to those suitable for storing them for persistence or for versioning, for generating code, for preparing demonstrations, presentations or documentation and many more. The ability to seamlessly use and combine all of these various tools is highly valuable and desirable. Accordingly, a standardized mechanism for representing (and thus exchanging) model information was included in the first UML standard. However, the mechanism laid out in UML 1.x supports only the definition of elements in a model. While this is important for tools that check

consistency of a model or generate code, this information is not sufficient for graphically oriented tools. This excludes a wide variety of tools which make use of the graphical information, including UML tools themselves. In this respect, the model interchange mechanism of UML 1.x falls short and the need to correct this has been recognized and addressed by the Object Management Group (OMG) in a Request for Proposals (RFP) for Diagram Interchange. This paper summarizes a response to that request by a syndicate made up of Gentleware, DaimlerChrysler, Telelogic, and Adaptive.

The general mechanism applied within the OMG to transport meta-information is the XML Metadata Interchange format (XMI) [10]. XMI is an application of the *Extensible Markup Language* (XML) [11] standardized by the World Wide Web Consortium (W3C) and was designed to be capable of transporting information which is to a great degree internally referential. Object-oriented models and metamodels, in particular, fall into this category. XMI is currently applied by many commercial tools to transport UML models. This is achieved by employing a standardized XML grammar able to represent all the modeling constructs defined by the UML metamodel. The XMI format is defined by an XML *Document Type Definition* (DTD), which is generated by applying the rules for DTD generation as also defined by XMI to the concrete UML metamodel itself. In summary, the XMI standard, which consists of the XML DTD designed for interchanging UML models, co-offers a defined process for generation of XMI-compliant DTDs.

To distinguish between XMI in general and its application to UML, we will refer to the latter format as XMI[UML] in this paper. This mechanism has proved to be highly useful despite the fact that graphical information was not included.

UML diagrams, similar to UML models, may be described by means of a *metamodel*. The term metamodel here is used in the same way the UML does it throughout its specification, as a model of model defining some aspects of the structural semantics. The metamodel itself is formulated re-using the language it describes, which is in our case UML.
In our approach we suggest a separate metamodel for diagram information which can easily be added as a separate package to the existing UML metamodel. Just as with the UML metamodel, XMI can be applied to transport instances of this diagram metamodel. The corresponding DTD (which simply extends the DTD of XMI[UML]) is created directly from the metamodel by a generator which is compliant to UML's meta-metamodel-the OMG-standardized *Meta Object Facility* (MOF). This format will be referred to as XMI[DI] in this paper. The conjunction of both, which makes up the complete model interchange mechanism, will be referred to as XMI[UML+DI] or simply as XMI for reasons of brevity.

The metamodel proposed conforms to the MOF metamodel facility [6]. XMI defines how to create a DTD or an XML schema from an MOF-compliant metamodel and how the schema is to be applied to XMI. That means that once an

MOF compliant metamodel has been agreed on, its representation in XMI and the corresponding DTD is taken care of.

The metamodel itself was developed with two main goals in mind. First, it was to flexibly extend the existing UML metamodel (as well as its future revisions) without either interfering with it or modifying it. At the same time it was to carry as little redundant information as possible and instead reference the UML metamodel to access that information. Second, it was to allow any tool to easily render the diagram from the given information. This includes not only UML tools but also web browsers, office suites, graphical editors, etc.

The tools mentioned are very different in their needs. While UML tools usually carry out the rendering of model elements to lines and text themselves, text editing tools have no knowledge at all about model elements and require a diagram to take on a common graphical format. And graphic tools require a rich vector-oriented format for manipulating and scaling.

SVG [7], the *Scalable Vector Graphics* format, is a new W3C standard which promises to be supported by a wide variety of these tools. It is based on XML and can easily be loaded and processed and then transformed into many other formats. It is equally suited for text tools, office suites, and graphic tools. It is also suitable for web browsers, which are expected to directly support SVG in the nearer future (note that, in the interim, freely available plug-ins may be used).

However, SVG is not well suited for UML tools. UML tools do not require merely the lines and text but information at a higher level: for they not only display the diagrams graphically, they also need a semantic understanding of the model elements represented by the graphical primitives.

Yet one of the great advantages of XML is that data expressed in any XML data format can easily be transformed into a different XML data format as long as all the necessary information is present. Therefore we suggest a metamodel transported using XMI[DI] and provide a transformation from this model to SVG using the *Extensible Stylesheet Language for Transformations* (XSLT) which defines a XML vocabulary that forms a Turing-complete functional programming language capable of transforming XML input into arbitrary textual streams as well as into XML. This approach makes it possible to satisfy the needs of a very wide range of tools. All other required formats can then be produced from this.

So, what is the abstraction which commonly expresses the additional information to allow interchange for all diagrams in UML? One alternative is to introduce special classes for every kind of shape UML diagrams consist of. However, if UML is to be extended or its scope broadened or if the core mechanisms are to be reusable for other modeling notations, this approach seems too inflex-

ible. The diagram interchange should not restrict the extensibility of UML. If possible, the diagram interchange mechanism should not have a notion of concrete shapes or other elements. The drawing of the concrete forms of the shapes used are the responsibility of the UML tools or of an SVG renderer, i.e. a software component visually displaying the two dimensional object described by the XML encoded SVG input.

It can be observed that most diagrams in UML follow the graph scheme as known from graph theory: they consist of nodes (which can be rectangles, ovals, circles or other shapes) and edges (connecting lines between the nodes with different arrows and shapes at the ends). Nodes may contain compartments and annotations, edges may have annotations attached to them. Some nodes may be nested in others, edges always connect two nodes.

The graph scheme is a very powerful, well-understood abstraction that is used in many areas of visual modeling. It is the abstraction used in our approach and proves to be fully sufficient, as will be discussed in this paper.

After the introduction to the topic in the current section, we provide an architectural overview. Section 3 discusses the diagram metamodel in detail and illustrates how it extends the UML metamodel. Section four describes how this can be applied to render diagram information in any kind of tool. Finally, we conclude and summarizes the results.

2 Architectural Overview

The XMI[UML] extension XMI[UML+DI] utilizes several technologies to create both its metamodel and instantiated objects of the model mentioned. The technologies involved have been published as standards by the OMG and the W3C and are in the public domain. The core technology, which serves as foundation for all the others involved in this process, is XML. It consists of basic rules (e.g. well-formedness) specifying how to create documents, describing their content by tagging. This commonly accepted mechanism is supported by many commercially available tools. The figure below depicts all the technologies involved in the creation process of the metamodel and DTD.

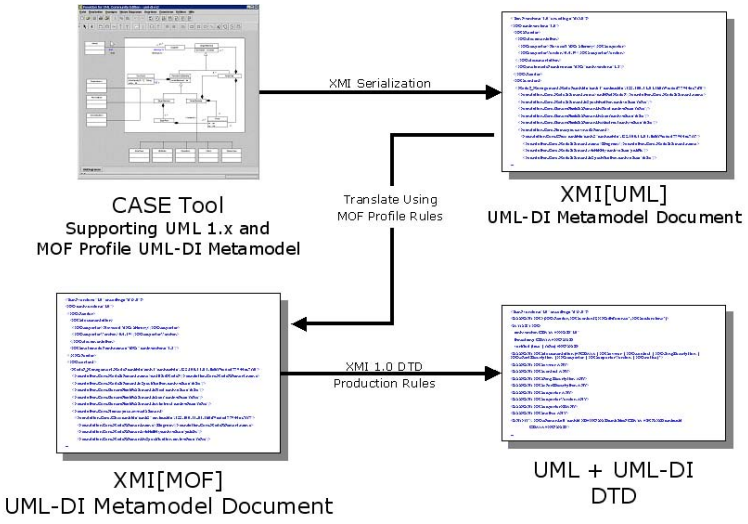

Fig. 1. XMI[DI] DTD creation process

The XMI[DI] metamodel is created with a UML modeling tool: a MOF-compliant metamodel (i.e. a model of a model, or *M2* for short) is created using the UML profile for MOF, describing the UML M2 extension. This tool needs the capability to create an XMI[UML] document compliant to XMI version 1.0 by serializing the model to an XML stream. And, according to the rules set out by the grammar defined by XMI[UML] 1.0 DTD, this is a document.

As mentioned, the generation process produces an XMI 1.0-conformant document which contains the content of the metamodel created with the help of the UML tool. In order to generate the new DTD for [UML+DI], the new M2 must be expressed in MOF form by translating the XMI[UML] document to an XMI[MOF] document. The rules for the mapping from the profile to MOF are included in the UML profile for MOF. Another option would be to generate the XMI[MOF] document directly from the CASE tool. Finally, the XMI DTD production rules can be applied to the MOF representation of the M2. Using the rules in the *XMI Production for XML Schemas* specification, an XML schema can also be generated.

Based on this extended XMI[UML+DI] metamodel, it is now possible to include the graphical information of an XMI[UML] model when exchanging it. Furthermore several representations of a model may be created. One alternative is to create a representation in SVG.

SVG is a technology geared to describe two-dimensional vectorized graphics in a clear text (XML-based) format and derive a visualization from this. It was

recently published in the form of a corresponding recommendation by the W3C. In contrast to plain graphics such as bitmaps, it is-just to mention some of the possibilities-possible to scale such graphics, rotate them or invoke these methods on single elements of a graphic. SVG is also capable of handling user interaction, offering a wide range of options when working with SVG-based graphics.

XSLT [12] is another W3C recommendation which defines how to create stylesheets (in themselves XML documents) for the transformation of one XML document into another (usually also an XML) document. In this case, the stylesheet is used to transform the XMI[UML+DI] XML document containing the UML model and diagramming information into an SVG XML document containing full graphical rendition information which can be displayed by a browser. Note that XSLT engines to perform stylesheet-driven transformations are commonly available, including some in open source (most notably *Xalan* from the Apache project).

The following diagram sketches an overview of the technologies involved in the creation of an SVG document from a UML modeling tool.

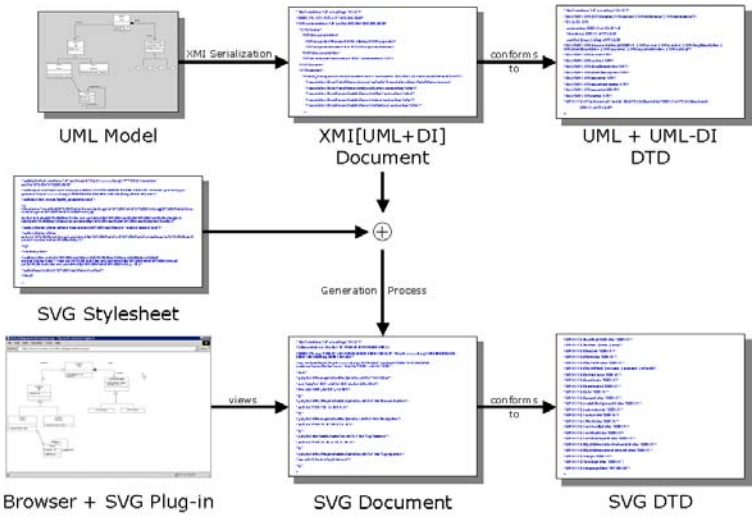

Fig. 2. SVG graphic creation

As when creating the XMI[DI] extension, the starting point is a UML modeling tool used to describe a model. On the basis of this model, an XMI document is created by a serialization process using the XMI 1.0 production rules. The result is again an XMI 1.0 document containing the content and, in contrast to the past, the graphical information of the model. The document may be validated against the XMI[UML] 1.x DTD containing the XMI[DI] extension to check the

syntactic correctness of the XMI document. In addition, the well-formedness of the document is always checked. The next step is to create an SVG document out of the XMI source document. This is done by using an XSLT stylesheet, which only has to be produced once and is then applicable to all XMI documents containing the XMI[DI] extension. Applying this stylesheet to the XMI[DI] source document generated yields the SVG document as output. The SVG is used with an SVG viewer to visualize the model by rendering the input document. These viewers may be integrated as a plug-in in any common Internet browser. As a result, it is possible to view and navigate UML diagrams in a browser with a high level of user interaction, which can be designed to be as intuitive and easy as UML tools can currently be deployed.

3 Proposed Metamodel Extension

The following chapter sets out the metamodel for diagram information which the diagram interchange mechanism relies on. It is an extension of the UML metamodel (currently based on UML 1.4). The existing mechanism of XMI[UML] to exchange models via XMI includes only the model information, leaving out the graphical information. The extension described in the following targets including the graphical information of diagrams in UML models, thus linking graphical elements to their model elements.

This extension adds a new package to the state-of-the-art UML metamodel packages. The existing standard is not changed in any way. Also, changes to the UML metamodel due to version updates should not impact this model as long as the high-level notion of *ModelElement* is maintained. The suggested extension and the UML metamodel are kept largely independent with solely links from the extension to the UML metamodel included. Thus, we achieve a clean separation of graphical and model information.

In addition, conflicts with tools supporting the current standard are avoided and full backward compatibility is maintained. Flexibility for future extensions to UML itself is ensured.

The proposed package contains elements for reflecting the diagram information of any diagram element of the standard UML. Tool-specific extensions can be defined in additional packages. For example, if a tool adds drawing capabilities for additional shapes, then an additional package to describe these can be provided. The underlying reason is to make sure that tools which do not support these additional extensions can still generate a graphical representation simply by ignoring the information from an extra package.

Figure 3 depicts the class diagram representing the metamodel for the diagram information.

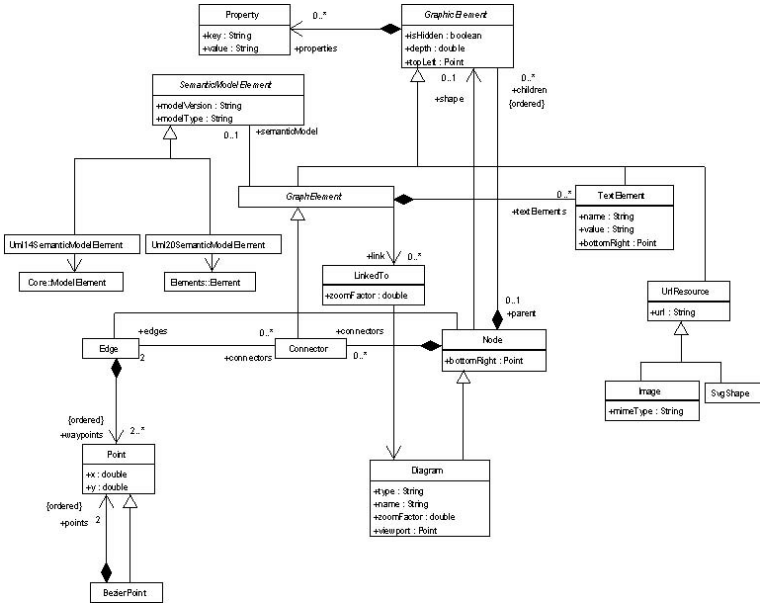

Fig. 3. Metamodel for diagram information

3.1 Extended Graph Model

This metamodel is built upon a metamodel for graph modeling. The core classes are *Node* and *Edge*. They are linked via a class called *Connector*. A Connector is a connection endpoint for an Edge which is owned by a Node. Hence, every Edge is linked to Connectors and the position of the Connector exactly marks the connection point of the Edge to the Node. A Connector may be endpoint for more than one Edge, but every Edge has exactly two Connectors. The superclass of this graph element is the class *GraphElement*. These elements allow a pure mathematical graph to be described without any semantic meaning.

Other graphic elements besides pure graph elements may also be described, e.g. graphic primitives such as circles and rectangles with no graph context. These have to inherit from the superclass for all graphic elements, *GraphicElement*. The inheritance from *GraphicElement* is intended be employed as an extension mechanism for this metamodel. Any sublasses of GraphicElement added should be placed in their own package, e.g. a package for graphic primitives.

Another extension of the pure graph model is achieved by building a hierarchy of nested nodes. Each Node may contain an unlimited number of GraphicElements. With this concept, every Node could contain an entire subgraph.

A special Node is the *Diagram*. It is the topmost Node of any graph or diagram in this terminology and recursively contains all other GraphicElements. A Diagram has a name and a type, which is important for the semantic context of the diagram (see below). It also has a zoom factor, which allows it to be shown in a different size.

Note that any graph element may be linked to other diagrams. These elements could be used if a graph element can be represented by another diagram, e.g. on a more detailed level or if the graph element has a special semantic relation to other diagrams. These links may have special zoom factors, so a diagram can be displayed with different zoom factors in each context.

The appearance of GraphicElements can be managed by properties. In the pipeline is a standard set of properties to be defined by the UML 2.0 specification and to contain font family and size as well as line style, thickness, and color. These properties, however, are optional. Every property overwrites any existing property of the same type of an enclosing GraphicElement. If a property is not set, the GraphicElement utilizes the property of the enclosing Node. Properties of different types could be added but should not be part of any standardization. GraphicElement has a depth to indicate the position of the z-coordinate in order to fix the elements which should be shown if they are overlapping.

The attribute *isHidden* allows a GraphicElement to not be shown and, if these should exist, all nested elements if this attribute is set to true. Even though the elements are not shown, they still exist. This means that if they are made visible by setting isHidden to false, they appear the same as before.

The class *TextElement* allows the representation of text. It contains a name/value pair, enabling the text to be linked to a semantic context (see below).

3.2 Positioning

Position and size of elements are specified through the class *Point*. The point *topLeft* of nodes, connectors, and text elements indicates the position of the element, while the optional point *bottomRight* allows the specification of a size. If this point is missing, the size must be calculated by considering the nested elements. For TextElements, the position is optional. If the position is not given, it is calculated by considering previous elements. For example, in a text such as an operation which consists of many TextElements the position of any TextElement is determined by the position of the previous TextElement.

Edges consist of an ordered list of waypoints and are represented by lines between these points. If a waypoint is an instance of a *BezierPoint*, the edge is represented by a Bezier curve.

Diagrams do not need to have a surrounding (parent) element and the position (topLeft) of a diagram is (0, 0) by default. The point bottomRight indicates the size of the diagram. The viewport is the coordinate of the point in the diagram which is currently shown in the top left position of the view. The viewport coordinate does not necessarily have to be (0, 0) if the view has been scrolled.

Every coordinate represented by Point is relative to the surrounding element if there is one. In order to change the position of a surrounding element including its subelements, only the coordinates of the surrounding element need to be changed. A typical example for such a surrounding element is represented by the rectangle forming a class symbol which contains all the text blocks used to represent attribute names, etc.

3.3 The Semantic Model

This model can be used to represent graphs with an additional semantic meaning by linking a GraphElement to the ModelElement of a semantic model via the *SemanticModelElement*. Every GraphElement may have an optional link to an instance of a concrete subclass of SemanticModelElement, such as *UMLSemanticModelElement*. This allows supplementation of UML-specific information to the graph. Other semantic models might be added as well, e.g. Entity-Relationship diagrams. The concrete SemanticModelElement has a link to a model element of the metamodel of the semantic model, e.g. the UMLSemanticModelElement is linked to elements of the UML metamodel. Note that this is a unidirectional link: there is no need for elements of the semantic model to have a link to their representation elements.

The model is designed to minimize the amount of redundant information. For this reason, there is no extra attribute to indicate the semantic type of an element. To ascertain the semantic type of an element, the SemanticModelElement must be examined. If there is no SemanticModelElement directly attached to the element, the SemanticModelElement of the surrounding element and, if there are any, the SemanticModelElement of the children have to be examined.

This model may be used in concert with the UML metamodel to represent any UML diagram through a graph with semantic links to the UML metamodel. For most of the UML diagrams, this can be done intuitively. Sequence diagrams are somewhat different, but there are ways to represent them using this metamodel and to unambiguously associate every model element with a graph element.

For example, in a class diagram, classes, interfaces, and packages are represented by Nodes, while associations, generalizations, and dependencies are represented by Edges. These Nodes and Edges have links to the related model elements of the UML metamodel. A class may contain multiple compartments. These are represented through nested nodes of the class-node with no link to the

semantic model, since compartments are not part of the UML metamodel-only part of its representation. Attributes and operations are nested nodes of the compartment nodes with a link to the corresponding attributes and operations of the UML metamodel. An attribute has properties such as visibility, type, and return value. These are represented by TextElements, e.g. textElement.name = visibility, textElement.value = public. TextElements are typically used to represent attributes of model elements which can be expressed through text.

The appearance of the symbols at the ends of the edges (associations, generalizations etc.) are figured out by the corresponding UML metamodel elements.

Other diagrams are represented in a similar fashion. The tool which uses this metamodel is responsible for the exact representation of elements which refer to semantic model elements. For example, the tool has to know that a node representing a class should be visualized through a rectangle. Its position, size, line style etc. are determined by the objects of the metamodel. If the shape of an element is complex, e.g. an actor in UML, a Node can optionally be linked to a GraphicElement representing the shape of the Node. This may be a subclass of GraphicElement which contains an SVG vector graphic.

4 Transformation to Graphical Representation

4.1 Transformation to SVG via XMI

Expressed in XMI, a model can be interchanged between tools that are aware of OMG's XMI format. However, this format is not very consumable for the human reader nor for tools that are purely graphically oriented. For this reason, a transformation into a graphical format is needed. The most promising format for this purpose is SVG. The format proposed in this paper is well suited for a transformation into SVG and was explicitly designed to make this transformation as straightforward as possible. Both data formats, XMI and SVG, are applications of XML, and the common set of XML tools can be used to manipulate them. XSLT [12] is such a mechanism, which is designed to transform one XML format into another. For proof of concept, we implemented a set of XSLT scripts in order to transform a model given in XMI[UML+DI] into SVG. Currently, this is applied to information regarding class diagrams only.

These style sheets extract information from an XMI file with model and diagram interchange-related data and build an SVG document. The resulting SVG file contains the representation of a single class diagram in UML notation.

4.2 Using the Metamodel Extension for Diagram Interchange in an Application

In this section, we introduce a possible approach which allows the integration of the metamodel extension for diagram interchange into an application. The details described belong to an ongoing project whose goal is to implement a framework in Java which allows viewing and editing of UML graphs.

For better understanding, some terms which will be used in this section are defined below:

- Metamodel[UML]: the UML metamodel.
- Metamodel[DI]: the metamodel extension for diagram interchange.
- Metamodel[UML + DI]: the combined metamodel.

The term *model[...]* describes an instance of the metamodel[...] created by an MOF-compliant generator. For reasons of brevity, metamodel is used as a synonym for metamodel[UML+DI] and model as a synonym for model[UML+DI] when there is no danger of confusion. It is essential to note that the metamodel[DI] allows access to the metamodel[UML] via the UMLSemanticModelElement as explained in section 3.

4.3 Extending the Metamodel for Diagram Interchange with Custom Operations

In order to simplify interchangeability between different UML tools, the metamodel[DI] itself should be completely passive, i.e. there are no operations in its classes. However, the model[DI] created from the metamodel[DI] by an MOF-compliant generator may have operations. Nevertheless it would be desirable to extend the model[DI] itself by adding custom operations in order to simplify its use in an application.

To solve this dilemma, a variation of the flyweight design pattern [2] was realized in a pilot implementation which assigns a tool class to each class of the model. Tool classes are obviously not subject to the UML diagram interchange discussed here since they have to be implemented for the chosen tool by the tool vendor itself. These classes contain the operations which are missing in the corresponding model class. The tool classes form exactly the same inheritance hierarchy as the classes of the metamodel[DI]. At runtime, a single instance of a tool class is assigned to all instances of the corresponding model[DI] class. For this reason, the instance of the model class concerned must be passed as a parameter to each call of an operation of the tool class. Aside from this small drawback, this implementation provides nearly all the advantages which the direct integration of the operations into the metamodel[DI] would have offered. For example, it is possible to override methods within the inheritance hierarchy of the tool classes as would have been done in the inheritance hierarchy of the

metamodel[DI] classes.

4.4 The Concept of Interaction

As the general concept of interaction, a variation of the model-view-controller architecture (MVC) [4] was applied. To be exact, two models are employed, because data is present in both the model[UML+DI] and in the GUI widgets. This is a result of the usage of Java/Swing GUI elements which combine the *Model* and the *View*.

A class called *Renderer* is a synonym for the *View*.

The Renderer reads data from the model[UML+DI] and transfers it to the GUI element, while a class called *Controller* takes care of the inverse direction. It reads the data from the GUI element and transfers it back into the model. In addition, the Renderer listens for changes in the model[UML+DI] in order to transmit them autonomously into the GUI. Here too, the Controller has the inverse task, i.e. to listen for changes in the GUI and to transmit them independently into the model[UML+DI].

Despite this symmetry, it seemed appropriate to break it in order to model Controllers as attributes of a Renderer. The reason for this is that the metamodel[UML+DI] has a very fine granularity, while the GUI elements used by a Renderer are often extremely complex. For example, an operation in the metamodel extension consists of several TextElements, while it would be appropriate for a Renderer to use a single text input field for the entire operation. In this case, a Renderer would own several Controllers to parse the text in the input field and transfer the result back to the model[UML+DI]. This simple approach would not be possible if a Renderer had a forced 1:1 relationship to a Controller.

4.5 Renderer and Controller hierarchies

Figure 4 shows a realization of the Renderer and Controller hierarchies. Each Renderer and Controller owns an association to a GraphicElement, i.e. the model[DI]. In each hierarchy there exists one class which corresponds to a class of the model[DI]. While both hierarchies are symmetrical, this symmetry is not a requirement as described above. The instances of Renderers form an object hierarchy at runtime, which can be seen from the associations children and parent of the class Renderer. Hence, like usual GUI elements, e.g. Java/Swing widgets, Renderers can be nested. This nesting is a simplification of the hierarchy of instances of the model classes which is built from the following associations:

- *Parent – children* between Node and GraphicElement.
- *Owner – textElements* between GraphElement and TextElement.

– *Node – connectors* between Node and Connector.
– *Edges* between Connector and Edge.

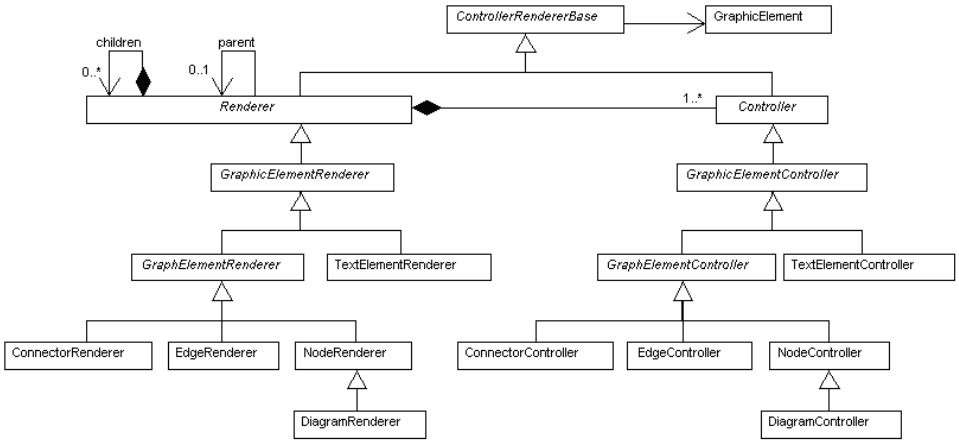

Fig. 4. Renderer and Controller hierarchies

This nesting allows Renderers to be built recursively at runtime and simplifies their use. For example, all the Renderers of a diagram are accessible indirectly through the Renderer at the root of the hierarchy, i.e. the *DiagramRenderer*.

The classes shown are only the most simple ones that are necessary for the integration of the diagram interchange information. For the editable graphical representation of complex UML elements such as a class, it may be desirable to extend the Renderer hierarchy by an additional Renderer for UML classes. This would not necessarily mean introducing a corresponding Controller, as the existing Controllers may be sufficient.

5 Conclusion

This paper describes the diagram interchange metamodel, which is an extension to the UML metamodel. It allows the standardization of the exchange of graphical information needed by UML models. The basic idea is that each UML diagram can be represented as a graph extended by the concept of nested nodes.

The metamodel extension needed to support diagram-specific information is linked to the UML metamodel. As this is a general mechanism, links to other semantic models are also possible. These might be other graphical description

languages, such as Entity/Relationship diagrams. This concept is flexible enough to support future versions of UML because the diagram interchange metamodel makes no presumptions about the structure of the semantic model. Instead, the application is responsible for its correct interpretation.

An example for such an application is an SVG transformer, which allows viewing of UML diagrams in a standard web browser. Notably, the SVG no longer includes semantic information. An approach for integrating the metamodel used to describe the diagram interchange related data into a Java-based application has been illustrated.

References

1. Harel, D. and Gery, E.: Executable Object Modeling with Statecharts. Proceedings of the 18th International Conference on Software Engineering, http://citeseer.nj.nec.com/article/harel97executable.html, 1997.
2. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
3. Gentleware: http://www.gentleware.com, 2001.
4. Glenn E. Krasner and Stephen T. Pope. A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming, September 1988.
5. Boger, M., Baier, T., Wienberg, F, Lamersdorf, W.: Extreme Modeling. Proceedings of the 1st International Conference on Extreme Programming, Italy, 2000. Addison-Wesley, 2001.
6. Object Management Group (ed.): Meta Object Facility (MOF) Specification v1.3, Framingham, USA, http://cgi.omg.org/docs/formal/00-04-03.pdf, March 2000.
7. Ferraiolo, J. (ed.): Scalable Vector Graphics (SVG) Specification v1.0, W3C Recommendation, Boson, USA, http://www.w3.org/TR/SVG, September 2001.
8. Object Management Group (ed.): Unified Modeling Language (UML) Specification v1.4, Framingham, USA, http://cgi.omg.org/docs/formal/01-09-67.pdf, September 2001.
9. Boger, M., Jeckle, M., Bjrkander, M., Rivett, P., Emmerich, W., Nentwich, C., and Baier, T.: Response to the UML 2.0 Diagram Interchange RFP, Object Management Group, OMG Document ad/2001-02-39, Framingham, USA, 2001.
10. Object Management Group (ed.): XML Metadata Interchange (XMI) Specification v1.2, Framingham, USA, http://cgi.omg.org/docs/formal/02-01-01.pdf, January 2002.
11. Bray, T., Paoli, J., Sperberg-McQueen, C. M., and Maler, E (eds.): Extensible Markup Language (XML) 1.0 (Second Edition), Boston, USA, http://www.w3.org/TR/2000/REC-xml-20001006, October 2000.
12. Clark, J. (ed.): XSL Transformations (XSLT) v1.0, Boston, USA, http://www.w3.org/TR/xslt, November 1999.