UML 2.0
Diagram Interchange

Second (Final) Revised Submission in Response to
OMG Document ad/2002-12-20

Version 1.0
January 6, 2003

Submissions due: January 6, 2003

Submitted by

Gentleware AG
DaimlerChrysler AG
Telelogic AB
Adaptive
Rational Software Corporation
Sun Microsystems

Supported by

Compuware Corporation
TogetherSoft Corporation
I-Logix
Softeam
University College London
Hamburg University

OMG document number ad/2002-12-20

Contents

Part I

1 Foreword

This document is a response to the Request for Proposal of the Object Management Group (OMG) as published in document ad/2001-02-39. It has been established in cooperation between Gentleware AG, DaimlerChrysler AG, Telelogic, Adaptive, Sun Microsystems, Compuware Corporation, Rational Software Corporation, TogetherSoft Corporation, University College London, and Hamburg University.

The goal of this submission is to enable a smooth and seamless exchange of documents compliant to the UML standard (in the following referred to as UML models) between different software tools. While this certainly includes tools for developing UML models, it also includes tools such as whiteboard tools, code generators, word processing tools, and desktop publishing tools. Also, special attention is given to the Internet as a medium for exchanging and presenting UML models.

Although a mechanism for exchanging UML models had been specified for UML 1.x using XMI, namely the XML Metadata Interchange, (in the following referred to as XMI[UML]), this mechanism did not fully fulfill the goal of a model interchange. Foremost, it did not include the exchange of diagram information. This mechanism was only capable of transporting information on **what** elements are contained in a UML model but not the information on **how** these elements are represented and laid out in diagrams. Thus, if a UML model is stored in one UML tool and then loaded in a different UML tool (or even the same tool) using XMI[UML], all diagram information is lost. This limitation is not due to XMI itself but instead to the fact that the UML metamodel does not define a standard way of representing diagram definitions.

The solution proposed extends the UML metamodel by a supplementary package for graph-oriented information while leaving the current UML metamodel fully intact. Moreover, it is compliant with the upcoming UML 2.0 metamodel and should also be unaffected by any subsequent changes to the UML metamodel. An MOF-compliant metamodel for UML diagram information is provided as extension to the UML metamodel, allowing the DTD for the XMI to be extended. The resulting XMI can then be used to exchange UML models between various tools without information loss.

To assure the exchange to tools that do not have a notion of model elements but of lines, text, and graphics, a transformation mechanism from XMI to SVG is provided. SVG is an XML-based format for scalable vector graphics that has been adopted as a W3C Recommendation. Well-suited to express any diagram of the UML, it will be a commonly used format for a wide variety of tools (graphical, desktop publishing, etc) and was created to be fit for the web.

In combination with a tighter definition of XMI[UML] in RFPs for Infrastructure and Superstructure for UML 2.0, this proposal will make a smooth and seamless exchange mechanism for UML models available.

2 Copyright waiver

3 Submission contact points

Dr. Marko Boger
Thorsten Sturm
Gentleware AG
Vogt-Kölln-Str. 30
22527 Hamburg
Germany
Marko.Boger@gentleware.de
Thorsten.Sturm@gentleware.de

Mario Jeckle
DaimlerChrysler AG
Wilhelm-Runge-Str. 11
89081 Ulm
Germany
mario.jeckle@daimlerchrysler.com

Pete Rivett
Adaptive
Dean Park House
8-10 Dean Park Crescent
Bournemouth
BH1 1HL
United Kingdom
pete.rivett@adaptive.com

Morgan Björkander
Telelogic AB
PO Box 4128
Kungsgatan 6
SE-203 12 Malmö
Sweden
morgan.bjorkander@telelogic.se

Wim Bast
Compuware Europe B.V.
Hoogoorddreef 5
P.O. Box 12933
1100 AX Amsterdam
The Netherlands
wim_bast@nl.compuware.com

Martin Matula
Sun Microsystems
Evropska 33e
16000 Prague 6
Czech Republic
martin.matula@sun.com

Jochen Seemann
Rational Software Corporation
8383 158th Ave. NE
Redmond, WA 98052
USA
jseemann@rational.com

Alexander Aptus
TogetherSoft Corporation
Gewerbestrasse 42
70565 Stuttgart
Germany
alex.aptus@togethersoft.com

Eran Gery
2 Pekeris St.
Tamar Park
Rehovott 76100
Israel
erang@ilogix.co.il

Philippe Desfray
144 avenue des Champs Elysees
75008 Paris
France
philippe.desfray@softeam.fr

Wolfgang Emmerich
Christian Nentwich
Department of Computer Science
University College London
London
WC1E 6BT
United Kingdom
w.emmerich@cs.ucl.ac.uk
c.nentwich@cs.ucl.ac.uk

Jens Fransson
Stefan Mueller
Arbeitsgruppe Verteilte Systeme
Vogt-Kölln-Str. 30
22527 Hamburg
Germany
3fransso@vsys.informatik.uni-hamburg.de
1mueller@vsys.informatik.uni-hamburg.de

The material presented in this proposal is available for download from the following website:

http://www.gentleware.com/projects/diagraminterchange/

For comments and suggestions, an email address has been installed. Please send email to umldi@gentleware.de.

4 Overview

A number of different paths to a solution for a diagram interchange mechanism can be taken. Therefore, this chapter introduces the solution we have chosen and explains the rationale behind why the solution suggested was preferred over other alternatives. Finally, the structure of the document is outlined.

4.1 Overall design rationale

UML is a modeling language for object-oriented software systems with a strong emphasis on a graphical representation. It is deployed throughout the software development process and there is a wide variety of tools that can be utilized during this process. Tools vary greatly: there is an extensive range geared to design the diagrams, to check the consistency of models, to store them for persistence or for versioning, for generating code, for preparing demonstrations, presentations or documentation and many more applications. The ability to seamlessly use and combine all of these various tools trouble-free is highly valuable and desirable. Accordingly, a mechanism for representing (and hence exchanging) model information was included in the first UML standard. However, the mechanism laid out in UML 1.x merely supports the definition of elements in a model. While this is essential for tools that check the consistency of a model or generate code, this information is not sufficient for graphically oriented tools. This thus excludes a wide variety of tools that make use of 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 deficiency has been recognized and addressed by the OMG in the RFP ad/2001-02-39. This document is a response to that RFP.

The general mechanism applied within the OMG to transport meta-information is XMI. XMI is an application of XML which lends itself to transporting information that is highly internally referential. Both object-oriented models and the models of such models fall into this category but are only one example. XMI was applied to transport UML models by generating a special DTD through applying the rules of XMI to the concrete UML metamodel. To distinguish between XMI in general and its application to UML, we will refer to the latter as XMI[UML] in this document. This mechanism has proved to be highly useful despite the fact that graphical information was not included.

UML diagrams, just like UML models, can be described using a metamodel. This document suggests a separate metamodel for diagram information that can simply be added as a separate package to the existing metamodel of UML. And just as with the metamodel of UML, XMI can be applied to transport instances of this diagram metamodel. The corresponding DTD (which simply extends the DTD of XMI[UML]) is generated directly from the metamodel and provided in this document. This format will be referred to as XMI[DI] here. The conjunction of both, which makes up the complete model interchange mechanism, will be denoted as XMI[UML+DI], or simply as XMI for brevity.

In the current version, one DTD is generated, describing the XMI[UML] and the XMI[DI] part. In next version of the submission, using XLink for referencing other ModelElements, both parts can be placed in different files, allowing different DTDs for each part to be generated.

The metamodel proposed in this document conforms to the MOF metamodel facility. XMI defines how to create a DTD or a schema from an MOF-compliant metamodel and how this is to be applied to XMI. Thus, 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 key objectives in mind. First, it should flexibly extend the existing UML metamodel (as well as future revisions) without interfering with it and without modifying it. At the same time it should carry as little redundant information as possible and instead reference the UML metamodel to access this information. Second, it should allow any tool to easily render the diagram from the given information. This, of course, includes UML tools but also web browsers, office suites, graphical editors, etc.

The tools addressed are very different in their needs. While UML tools usually enable 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 be in a common graphical format. And graphic tools typically require a rich vector-oriented format for manipulating and scaling.

SVG, the scalable vector graphic format, is a new W3C standard that promises to be supported by a wide variety of these tools. It is based on XML and can easily be read in, processed, and transformed into many other formats. It is equally suited for text tools, office suites, and graphic tools. Moreover, it is suitable for web browsers that will soon support it directly (but in the interim can use a freely available Adobe plugin).

However, SVG is not a well-suited mechanism for UML tools. UML tools do not require solely the lines and text but need information at a higher level for they do not merely display the diagrams graphically they also call for a semantic understanding of the ModelElements that are represented by the graphical primitives.

Yet one of the great advantages of XML is that data expressed in one XML data format can easily be transformed into a different XML data format as long as all required information is present. Therefore this proposal suggests a metamodel transported using XMI[DI] and provides a transformation from XMI[UML+DI] to SVG using XSLT. This approach satisfies the needs of the widest range of tools. All other required formats can then be produced from this.

So, what is the abstraction that commonly expresses the additional information to allow interchange for all diagrams in UML? While one alternative would be to introduce special classes for each and every kind of shape that UML diagrams consist of, 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 inflexible. 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 lie in the responsibility of the UML tools or of an SVG renderer.

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

The graph schema is a very powerful, well understood, abstraction that is employed in many areas of visual modeling. It is the abstraction used by this proposal since it proves to be fully sufficient, as will be discussed in this document. Most UML diagrams have a natural and straightforward mapping to a graph. This is also the case for class diagrams, use case diagrams, collaboration diagrams, object diagrams, component diagrams, and deployment diagrams. In the current form of the UML, it is only sequence diagrams that do not have such a natural mapping to nodes and edges; yet a mapping can clearly be found. It is argued that, although the graph based approach may not be the obvious choice for all diagram types, it is well suited for most and fully sufficient to represent any current and future UML diagram types.

4.2 Document structure

This document is structured as set out in the RFP. The proposal consists of three parts. The first part contains a short introduction, formal information as copyright waiver, and contact points supplemented by the overview and a statement of proof of concept. The second part is the core of the specification. A full example is given in the third and final section.

5 Statement of proof of concept

The proposed specification is at the stage of the first revised submission. The metamodel described has been fully implemented and tested using a set of different diagrams. The concepts presented are regarded as proven. However, this mechanism has thus far not been employed for interchange between different vendors and it might be necessary to tune some of the details to ensure interoperability.

A more detailed evaluation of whether all the aspects of UML 2.0 have been taken into account is needed. However, this can only be done after UML 2.0 has been stabilized. In the current version, UML 1.4, XMI 1.2, and MOF 1.4 are used. It should not be very difficult to finalize this proposal to use UML 2.0, XMI 2.1, and MOF 2.0.

The example provided in chapter 10 and used throughout the document has so far been transformed into its XMI representation by hand rather than automatically. However, this example has been validated against the DTD provided and translated to SVG using the XSLT made available.

After the adopted submission, the proposed specification will be fully implemented by several parties and the interoperability between different tools will be tested during the finalization process.

5.1 Resolution of RFP mandatory and optional requirements

The RFP includes a number of requirements, some of which are mandatory and some of which are optional. This section clearly states which requirements are met by this proposal and summarizes how in short.

5.1.1 Mandatory requirements

Mandatory requirements are listed in the RFP in section 6.5 and are cited below for reference.

6.5.1 DIAGRAM INTERCHANGE METAMODEL FEATURES. The Metamodel shall cover the representation and interchange of the following diagramming features :
- Diagram placement (X, Y and optionally Z co-ordinates)
- Grouping of diagram elements
- Alignment of diagram elements
- Fonts
- Support for character sets
- Color
- Attachment points on relationships
- Visibility on artifacts (eg. hide operations in a class, classes in a model etc.)
- Non-UML artifacts on diagrams (ie. artifacts not covered by the current UML notation)
- Scaling information, rotation information etc.
- User-defined icons and shapes
- Positioning aspects of diagram elements (eg. position of polygon vertices, position of diagram element names , line segments etc.). For example, some tools allow a line segment to be represented as a collection of ordered points

to allow custom routing and positioning of lines. The metamodel shall support this capability. Submitters shall justify additions or removals from the diagramming features listed above in the diagram interchange metamodel.

- Diagram elements are represented by their positions within the diagram. Their position is expressed using points with coordinates in x and y dimensions. The z dimension is expressed through the order of nested elements which reflects the inner working of many graphical tools well. The coordinate system is that of SVG.
- Grouping of elements and their alignment are not represented explicitly: these are regarded as a functionality of the modeling tools at runtime. Grouping of primitive graphical elements to UML model elements is done by nesting. Since the placement is transported using coordinates, alignments are implicitly transported.
- Fonts, character sets, and color are expressed using tagged values, referred to as properties. The names of the tags and the sets of possible values are taken from SVG, thus making the full expressiveness of SVG available.
- Attachment points of relationships are modeled by the physical location of GraphConnector.
- Visibility of nodes and of parts of nodes such as compartments or method names of a class can be specified as hidden.
- Non-UML artifacts are dealt with in an extra package that reuses a subset of SVG (in the pipeline) or as an SVG file that is referenced through a URI.
- Scaling and rotation are currently not included but can be specified on the diagram level as attributes in XMI. Positions, however, should be saved relative to 100% scale and no rotation.
- User-defined icons and shapes will be transported inline using their MIMEtype or as external URI resources or as SVG shapes.
- All elements, annotations, and waypoints are stored with their precise positions. Edges can be stored with an arbitrary number of waypoints. Waypoints can be simple points or Bezier points

6.5.2 REUSES UML Metamodel. The proposal shall not gratuitously change the UML Metamodel. The proposed metamodel shall be extensible. (Eg: for interchanging data models). Proposal shall support the interchange of UML models that have NO diagram information. This maintains upward compatibility with current designs (eg: UML Models that typically don't have much diagram information) and also allows interchange of designs between visual as well as non-visual tools.

The UML metamodel is left unchanged. It is referenced from the diagram interchange model but on a high level of abstraction. Hence, even changes to the UML metamodel (as expected with the transition to UML 2.0) do not effect the diagram interchange. Models can be transported without diagram information simply by leaving this information out.

6.5.3 VOCABULARY. The metamodel shall be based on the vocabulary and concepts of the UML Notation Guide as well as related graphics standards (eg: Scalable Vector Graphics…) as appropriate.

We have used the vocabulary to the best of our knowledge.

6.5.4 CHANGES TO UML METAMODEL. The proposal for the diagram metamodel can introduce changes to the UML semantics metamodel if those changes create a cleaner separation of the semantic and notational aspects of UML. For example, the proposal could remove PresentationElement and/or Geometry from the semantics metamodel. Submitters should be aware that these changes could be impacted by the related UML 2.0 RFPs that are proceeding in parallel: specifically UML Infrastructure and UML Superstructure RFPs.

As stated below Requirement 6.5.2, the UML metamodel is left unchanged.

6.5.5 NONREDUNDANCY. Properties of model elements (in the UML semantic metamodel) shall not be repeated in the diagram metamodel. Rather, presentation elements shall have references to UML metamodel elements and otherwise have only presentation properties. For example, a presentation element for a class shall not store an indication of whether to use Italics for the class name because the use of Italics is based on whether the corresponding class is abstract. Likewise, a presentation element for a visibility adornment on an association end shall not store visibility. It shall only store the presentation properties of the adornment. The visibility comes from the corresponding association end. It is the intention of this RFP to discourage submitters from adding diagramming concepts to the UML semantic metamodel itself.

Properties of the model elements are not repeated but instead referenced from the XMI[DI] part. Currently this is done using XMI references. The proposed diagram interchange mechanism does not transport font information UNLESS explicitly changed in the tool. The display of italic fonts, for example, is left to the renderer, which should conclude this from the information given in the UML model (such as abstract). No diagramming concepts are added.

6.5.6 PARTITIONING. The metamodel shall ensure that a UML model element can be presented in multiple diagrams. The metamodel shall also allow different diagrams of a model to be in different XML documents.

The proposed metamodel structures the XMI[DI] file by diagrams. From within each diagram description ModelElements are referred to by links. Thus, model elements can be present in any number of diagrams and it is ensured that the information they use is consistent since no data is kept redundantly but is referenced instead. The links employed will eventually be XLinks. These can be utilized both within a single file and across file boundaries. As a result the model information (XMI[UML]) and the diagram information (XMI[DI]) can be kept either in a single file or in separate files. Furthermore, diagrams can be kept together in one file or in one file per diagram or in any combination of multiple diagrams per file. Note, however, that the default setting should be to store everything in one file.

6.5.7 MOF and XMI COMPLIANCE. The diagram metamodel shall be MOF compliant and shall be provided as an XML document that conforms to the MOF Model DTD. Based on the MOF Specification the metamodel shall define programmatic interfaces (in IDL) to diagram information. Based on the XMI Specification the metamodel shall define XML diagram interchange using a UML Diagram Interchange XML DTD.

The proposed metamodel is compliant to MOF (currently 1.4) and is provided in XMI (currently 1.2), conformant to the MOF DTD. The DTD for XMI[DI] is also provided. In the final submission these will be in the versions of MOF and XMI then current, presumably MOF 2.0 and XMI 2.0.

6.5.8 DIAGRAM MANIPULATION WITHOUT CHANGING SEMANTICS. When dealing with complex designs, many tools provide the ability to perform various activities such as 'scale up/down', 'move', 'rotate' etc. on diagrams. It must be possible to interchange designs that have been manipulated in this manner. The diagram manipulation should not change the semantics of the design. Technologies such as Scalable Vector Graphics (SVG) provide some of this capability. The submitters are encouraged to review the XML grammar for SVG at http://www.w3.org/Graphics/SVG and use the concepts and terminology as appropriate in this proposal.

This proposal explicitly supports SVG. However, since SVG carries information at a level too detailed to be used directly as saving and exchange format, we propose that the graphical information be stored in an XMI-based format and that SVG be generated through a transformation from this (for example, using XSLT). The SVG can then be further processed or manipulated by other tools. Changes to these diagrams do not affect the semantics of the models that are saved in XMI and thus are clearly separated from their representation in SVG.

5.1.2 Optional requirements

Optional requirements are listed in the RFP in sections 6.6 and are cited below for reference.

6.6.1 3D Representation. The proposal may address visualization and representation of 3D models. (For example to represent and interchange very complex designs)

This proposal does not support 3D diagrams. The z dimension is used solely to represent ordering information.

6.6.2 Layering of Diagrams. The proposal may address the presentation of graphical elements on different layers of the same diagram. This is distinct from 3D representation, in that it would be possible to show, or emphasize, sub-diagrams by displaying one or more layers of the total number of layers. In graphics parlance this is the concept of viewplane (or cells in the context of animation). It might be allowed that a 3D presentation could display connections between entities between two or more layers. In general, it would be expected that the layers represent logical collections within one diagram (or model). Layers are primarily a way to filter the display and simplify complexity.

This is currently not included but desired in further development.

6.7 Issues to be discussed. Submitters should discuss any referential integrity issues which arise from addressing mandatory requirement 6.5.6.

Saving of diagrams in separate files will be supported as soon as XLink is used. Links from a diagram to the model are expressed by means of XLink. This causes a problem to crop up: the diagram can only be interpreted if the model is at the expected location. The alternative would be to save information redundantly but this is rejected as it leads to problems of consistency (and larger amounts of data). Therefore the user has to be made aware of this. To simplify the movement of files jointly, the set of files could be compressed into a single ZIP file. This has the additional benefit of reducing file size.

Part II

6 Architectural overview

The extension of XMI[UML] to XMI[UML+DI] employs several technologies to create both its metamodel and the instantiated objects of the mentioned model. The technologies involved have been published as standards by the OMG and the W3C and anyone is free to utilize them. The basic technology, serving as foundation for all the others involved in this process, is XML. It consists of fundamental rules (e.g. well-formedness) for how to create documents, describing their content by tagging. This commonly accepted mechanism is supported by many tools all around the world.

The following diagram provides an overview of all the technologies that are involved in the creation process of the metamodel and DTD.

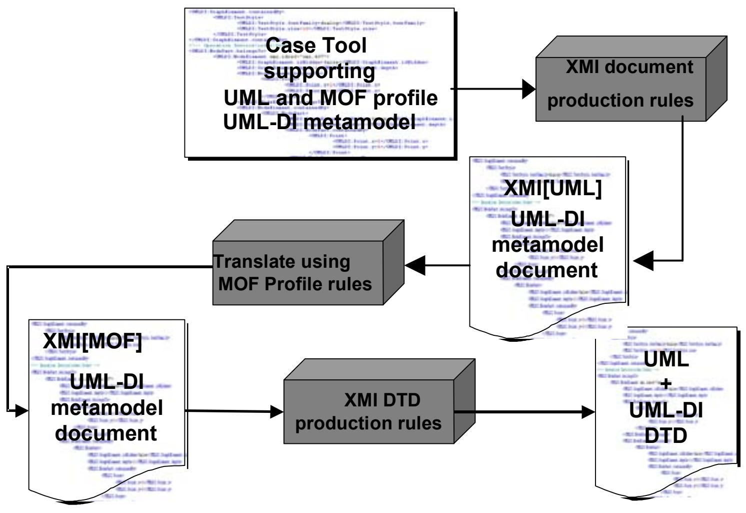

Case Tool supporting UML and MOF profile UML-DI metamodel
XMI document production rules
XMI[UML] UML-DI metamodel document
Translate using MOF Profile rules
XMI[MOF] UML-DI metamodel document
XMI DTD production rules
UML + UML-DI DTD

Figure 6.1: XMI[DI] DTD creation overview

The XMI[DI] metamodel is created with a UML modeling tool. With this tool, an MOF-compliant metamodel is created using the UML profile for MOF, describing the UML M2 extension. For conformance to the XMI specification, this tool needs the capability to create an XMI[UML] document. This is a document following the rules defined in XML and an XMI[UML] DTD belonging to it. This step is represented by the XMI document production rules. As mentioned, it produces an XMI-conformant document containing the content of the metamodel created with the help of the UML tool. In order to generate the new DTD for XMI[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 contained 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 can be created. One option is to create a representation in SVG.

SVG (Scalable Vector Graphics) is a technology geared to describe vectorized graphics in a clear text (XML-based) format and produce a visualization out of this. It was published as a Recommendation by the W3C. In contrast to plain graphics such as bitmaps, SVG enables, just to mention some of the possibilities, graphics to be scaled, rotated or methods invoked on single elements of the technology. It is also capable of handling user interaction, which offers many alternatives to work with such SVG-based graphics.

XSLT (eXtensible Stylesheet Language) is another W3C Recommendation which defines how to create stylesheets (themselves XML documents) for the transformation of an XML document into another (usually also 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 that can be displayed by a browser. Note that XSLT engines to perform stylesheet-driven transformations are commonly available including in open source (most notably Xalan from the Apache project).

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

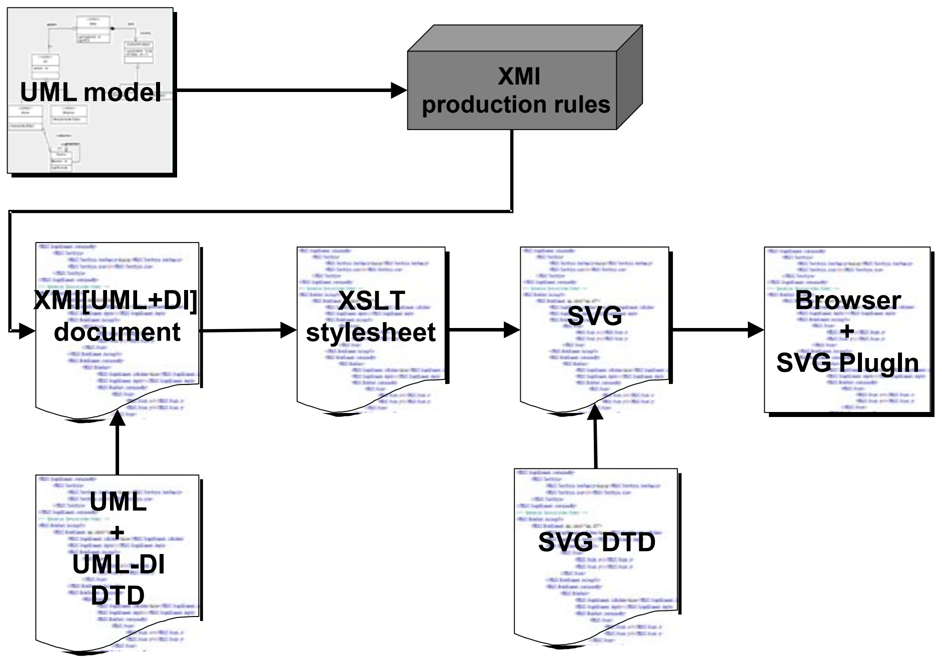


Figure 6.2: SVG graphic creation overview

As when creating the XMI[DI] extension, the starting point is a UML modeling tool to describe a model. Based on this model, an XMI document is created using the XMI production rules. The result is again an XMI document containing the content and, in contrast to past approaches, the graphical information of the model. The document can be validated against the XMI[UML] DTD containing the XMI[DI] extension in order to check the syntactic correctness of the XMI document. Apart from this, 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 needs to be produced once and is then applicable to all XMI documents containing both the model and the diagram information. Applying this stylesheet to the XMI[UML+DI] source document generated results in an SVG document. The SVG document can be displayed as a diagram using an SVG viewer to visualize the model by rendering the input document. Different viewers exist, including integrated plug-ins for common internet browsers. Thus, 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 used.

7 Proposed metamodel extension

This chapter describes the metamodel for diagram information which the diagram interchange mechanism relies on. It is an extension of the UML metamodel and is currently based on UML 1.4. The existing mechanism of XMI[UML] for exchanging models includes only the model information but not the graphical information. The diagram interchange extension allows graphical information to be included for diagrams used in UML models.

This extension adds a new package to the current UML metamodel packages. Yet the existing standard is not changed in any way. Also, changes to the UML metamodel due to version updates should not affect this model as long as the high-level notions of Core::Element (as used in UML 1.x) and Elements::Element (as used in UML 2.0 as well as all metamodels based on the Common Core) are maintained. The proposed extension and the UML metamodel are kept largely independent such that solely links from the extension to the UML metamodel are included. Thus, graphical and model information are cleanly separated.

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 provided.

The proposed package contains elements to reflect 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 purpose of this is to guarantee that tools not supporting these additional extensions are nonetheless able to generate a graphical representation by simply ignoring the information from an extra package. Figure 7.1 depicts the proposed metamodel for representation of diagram information.

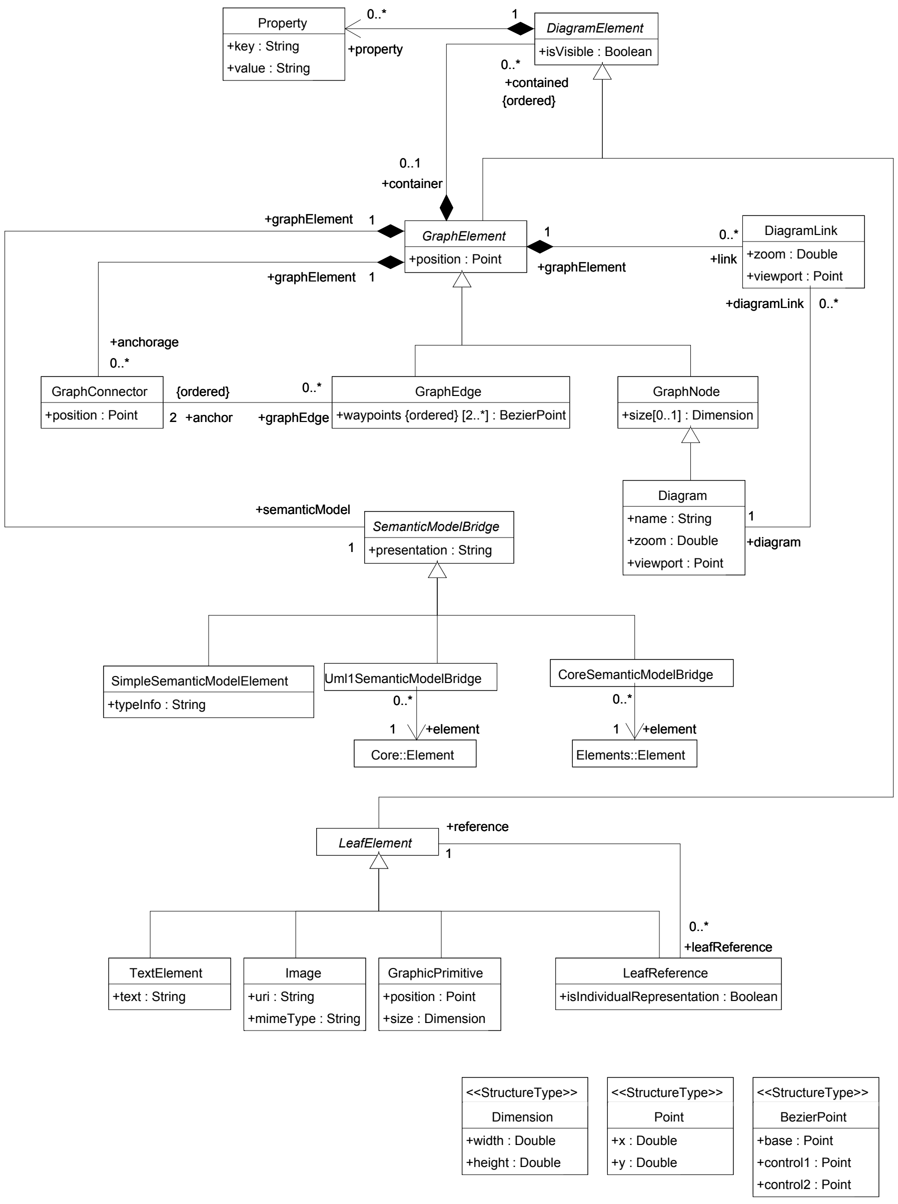


Figure 7.1: Diagram interchange metamodel

Extended graph model

The underlying concept of this metamodel extension is based on the idea of modeling the contents of the UML diagrams as graphs. The core classes are GraphNode and GraphEdge. Every visible model element is represented either by a GraphNode or by a GraphEdge. The base class of the graph elements is GraphElement. Graph elements are linked via a class called GraphConnector. This allows linking of a GraphEdge with a GraphNode or another GraphEdge. The latter case is an extension to the concept of a pure mathematical graph. A GraphConnector does not permit two GraphNodes to be linked.

A GraphElement can own any number of GraphConnectors, called anchorages. They permit any number of GraphEdges to connect to them. A GraphEdge references two GraphConnectors, which are its connection end points. From the perspective of the GraphEdge these are called anchors. The two GraphConnectors are ordered in the same way as the waypoints of the corresponding GraphEdge. The first GraphConnector corresponds to the first waypoint of the GraphEdge and the second GraphConnector corresponds to the last waypoint.

A GraphConnector is not obliged to have the same position as the end points of the GraphEdges which reference it. When being referenced by more than one GraphEdge, it serves as a virtual collection point for GraphEdges, e.g. in the middle of a node or at another specialized point of a node. All linked GraphEdges point straight to the GraphConnector while its waypoints may, for example, be defined as ending at the border of the shape of the connected node (Figure 7.2). A UML tool which reads this information may interpret it as clipping, which might be helpful for further editing.

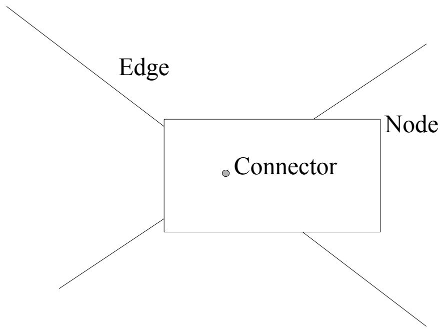


Figure 7.2: Example of a GraphConnector with radial edges

Nesting

The association container—contained between GraphElement and DiagramElement—constitutes a further extension to the model of a pure mathematical graph. It allows a hierarchy of nested elements to be built. Each GraphElement is able to contain an unlimited number of DiagramElements, so that it may contain an entire subgraph. The nested hierarchy is especially useful when modeling the representation of complex model elements.

For example, classes are represented by a GraphNode which owns nested GraphNodes as contained DiagramElements to represent the compartments, e.g. operation compartment, attribute compartment. Such a compartment GraphNode, e.g. an attribute compartment, owns further nested GraphNodes as contained DiagramElements which represent attributes. The attribute GraphNode again has nested contained DiagramElements which represent parts of the attribute, e.g. visibility, name, type or initial value.

Figure 7.3 sets out a simple class as an example of the nesting of GraphElements, which are GraphNodes in this case. A more complex class may contain more nested GraphNodes to be represented completely.

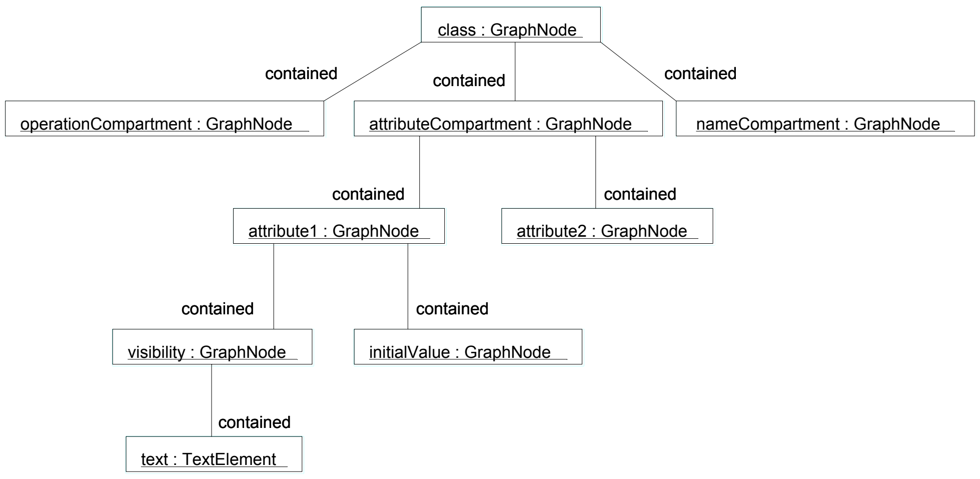


Figure 7.3: Example of nested GraphElements in the representation of a class

Leaves

The graphical representation of a GraphElement is largely derived from the semantic model (see below) and not explicitly stored in the DI metamodel to avoid redundancy. For example, the knowledge of drawing a class as a rectangle is not stored in the DI metamodel. Note that its position and its size, if needed, are stored. A drawing tool and the XSL stylesheet must have the semantic knowledge of drawing these elements correctly.

LeafElements can be utilized to represent model elements with special optional representation, e.g. actors. If a GraphNode representing an actor has the presentation of the SemanticModelBridge set to UserDefined (see below), it must have a LeafElement as child. This LeafElement represents the user-defined visualization of the actor.

A LeafElement can be a TextElement to show a simple text, a GraphicPrimitive to show a simple drawing such as an ellipse or a UriResource to show an external resource, e.g. an image in a separate file.

Specialized graphic primitives are not part of this submission. They are defined in an extra package and inherit from the class GraphicPrimitive. TextElements are employed to represent parts of attributes, operations, names, and other texts which are part of model elements. For example, the visibility of an attribute can be a text such as “public” or “+”. This text is stored in the attribute *text* of the TextElement. The container GraphNode of this TextElement has a link to the semantic model in the form of a SimpleSemanticModelElement. This indicates the part of the attribute which is being referenced. The attribute *typeInfo* of the SimpleSemanticModelElement contains *visibility* in this example. Naming rules for the attribute *typeInfo* are described in section ‘additional semantic information’ below. Figure 7.4 illustrates the relevant part of the objects involved. The SimpleSemanticModelElement of the GraphNode *name* is not shown in this figure.

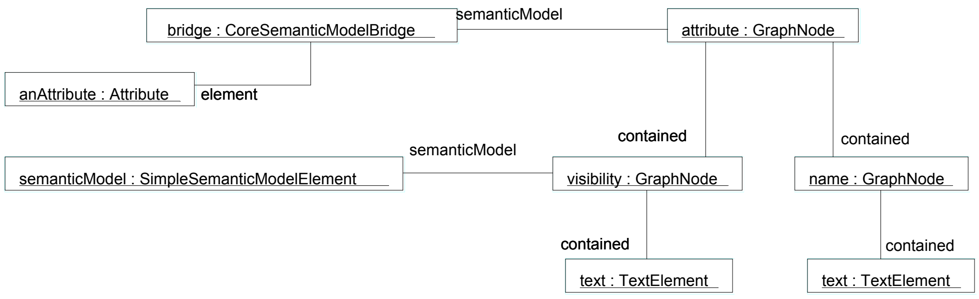


Figure 7.4: Example of a TextElement

If the visibility is to be represented by an icon, the TextElement is replaced by another LeafElement. If the icon is stored as an image in another file, the LeafElement is a subclass of a UriResource referencing this file.

LeafElements which are used many times are only needed to be instantiated once. These LeafElements are directly contained by the Diagram. Every time such a LeafElement is used, a LeafReference referencing this LeafElement is instantiated instead of a copy of the LeafElement. The isIndividualRepresentation flag is set to *true* to distinguish this usage of the LeafReference from the usage described below. This avoids multiple copies of the same visual component.

Diagram

A special node is the Diagram. It is the topmost GraphElement of any graph or diagram in this terminology and recursively contains all other GraphicElements. A Diagram has a name and a viewport. The viewport is a point which indicates the top left-hand corner of the current visible part of the Diagram. It also has a zoom factor, which allows it to be shown at a different scale. The type of the Diagram is stored in a SimpleSemanticModelElement, e.g. StateDiagram. The Diagram references it through its semanticModel.

A graph element can be linked to diagrams via a DiagramLink. Such a link could be employed if a graph element can be represented by other diagrams, e.g. on a more detailed level or if the graph element has a special semantic relation to other diagrams. A DiagramLink has its own zoom factor and viewport, so each diagram can be displayed with a different zoom factor and viewport in each context. An example for the use of a diagram link is a state diagram which displays the behavior of a class or a class diagram, which displays details of a package.

A simple example of a Diagram is given through the object diagram in Figure 7.5. It shows a class diagram with two classes which are linked through an association. The diagram and the classes are represented through GraphNodes, while the association is represented by a GraphEdge. This object diagram also shows the role which GraphConnectors assume in linking GraphNodes and GraphEdges.

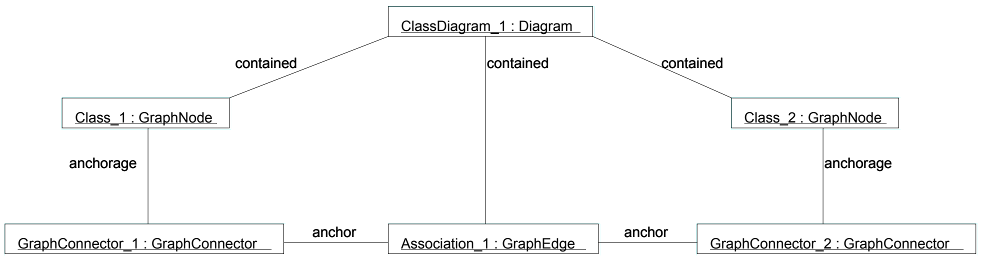


Figure 7.5: Example of a class diagram(only the UML[DI] part depicted)

Visibility of diagram elements

The attribute visible specifies whether a DiagramElement is being shown or not. Its value is applied recursively to all contained DiagramElements. If only one DiagramElement in the hierarchy has set its visible attribute to *false*, all nested elements are also hidden, regardless of their own visibility.

While a hidden element is not shown, it still exists. This means that if it is shown by setting visible to *true*, the element appears in the same way as before. This provides a comfortable way to fade out a compartment, for example, while keeping its representation. This makes it easy to show a previously hidden DiagramElement again later.

Properties

The appearance of DiagramElements is specified by properties. Properties can be added to a DiagramElement, influencing its appearance in different ways. The table below defines a standard set of optional properties: it comprises font family and size as well as line style, thickness, and color. Each property overwrites any existing property of the same type of an enclosing GraphElement. If a property is not set, the DiagramElement utilizes the property of its container GraphElement. Non-standard properties could be added but are not part of any standardization. Properties provide an extension mechanism for adding non-standardized elements to the diagram interchange metamodel.

Key	**Domain**	**Example**	**Description**
FontFamily	string	‘Times’, ‘Courier’	font name
FontSize	double	11.0	font size in pixel
LineStyle	string	‘solid’, ‘dotted’	line style
LineThickness	double	2.0	line thickness in pixel

FontColor, ForegroundColor, BackgroundColor	integer	FF00FF for magenta	24-bit color value in RGB format
Translucent	boolean	true, false	see text

Table 7.1: Standardized properties

The property 'Translucent' is set to *true* for GraphElements to indicate that their contained elements can be seen through the background of the containing GraphElement (see sequence diagram).

Coordinate system

The coordinate system used in the diagram interchange extension defines that the x-axis points to the east and the y-axis to the south. Position and size are defined through double values which employ pixels as the unit of measurement. The floating point type *double* allows sub-pixel accuracy. This prevents a possible loss of information when exchanging scaled or otherwise manipulated diagrams.

Position

Generally speaking, all position values used in this model are relative, i.e. any position value of a GraphElement is relative to its surrounding container. This means that the container's position is the origin for any child's position. To obtain the absolute position of a DiagramElement, it is necessary to navigate recursively through the container GraphElement until a GraphElement with no container, the root, is encountered. This root node must be a Diagram since it is only Diagrams that do not have a container.

The position of a GraphElement is specified through the StructureType point. This indicates the top left-hand corner of a GraphNode or Diagram. The x and y coordinates of a position are allowed to be negative.

For a GraphEdge, the position serves as an origin for all the DiagramElements contained. Since a GraphEdge is defined through its waypoints, position does not affect the graphical representation of the GraphEdge itself. It is drawn as a Bezier curve through all of its waypoints, which are of type BezierPoint. The control points are defined relative to base. Thus, if all control points are zero (0, 0), the result is a simple polygon line.

A Diagram does not need to have a surrounding container element. Its position is (0, 0) by default. The viewport specifies the top left-hand coordinate of a currently shown view within the Diagram. The viewport is different from (0, 0) if the view has been scrolled.

The relative coordinates allow hierarchies of nested nodes to be moved easily by changing the position of the root node. For example, moving a class can be done by changing the position of the GraphNode representing it. All the GraphElements contained automatically move with the class since they are positioned relative to it.

Size

For GraphNodes, an optional size can be provided using the StructureType Dimension, which encapsulates width and height of type *double*. None of these are allowed to be negative. If the size is not given, it is determined by its semantic model and the nested DiagramElements within this GraphNode. The following two cases are to be considered:

1. If the semantic model indicates that a graphical representation for the given GraphNode is necessary, its size must be calculated from its nested DiagramElements. For example, if not given, the size of a GraphNode representing a class must be determined by recursively examining the

extent of the contained compartments. If the size of a compartment is not given, its contained DiagramElements, e.g. attributes, must be analyzed and so on.

2. If the semantic model indicates an Element of the semantic model with no or a fixed graphical representation, this calculation is not necessary. An example for such a case is an AssociationEnd. Although, for example, GraphNodes containing TextElements for multiplicity and role name may be nested as contained into such a GraphNode, the AssociationEnd's GraphNode itself has no size—it is simply being used as a container. Even if the AssociationEnd is visible in a sense, e.g. a filled rhombus to indicate a composition, this is considered a fixed graphical representation which therefore does not have an extent.

Rendering order

The general rendering order of DiagramElements is defined through the following two mechanisms:

1. The nesting tree of DiagramElements within GraphElements has the highest priority. To render the DiagramElements, the tree is processed using a preorder traversal. This means that the most deeply nested DiagramElement of a path is drawn last and thus is shown on top in z-order.

2. The ordered association container—contained between GraphElement and DiagramElement—specifies the rendering sequence of DiagramElements contained within the same GraphElement. The first DiagramElement is drawn first, the second next, and so on.

There is a less common situation in which this straightforward rendering concept alone is not sufficient. LeafElements are employed to enrich a diagram with elements not belonging to UML. It may be useful to show such a LeafElement as a background for more than one GraphElement. An example for this case is an ellipse which is drawn through an Image. The ellipse contains two classes A and B from two different packages a and b, while being drawn on top of the two packages. Figure 7.6 illustrates this situation.

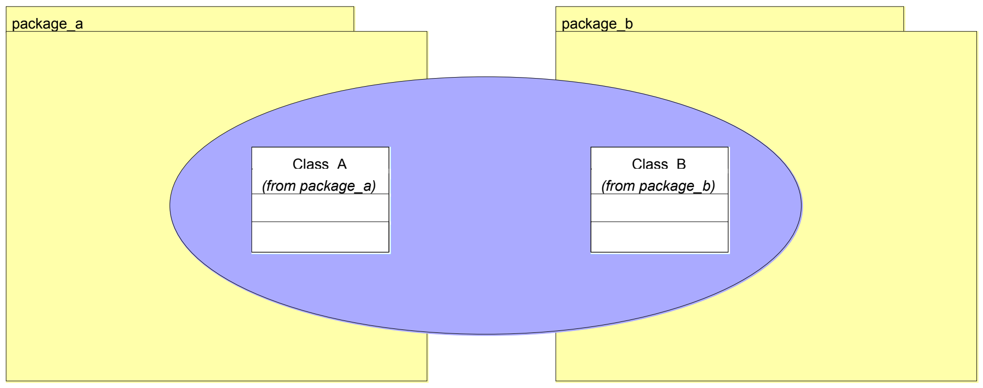


Figure 7.6: An ellipse contained within two paths in the tree of DiagramElements

Rendering figure 7.6 is impossible with the two mechanisms described above, since any DiagramElement, i.e. the Image, may only be part of a single path in the tree of DiagramElements. In order to be shown in this way, it must be part of both paths. To accomplish this, the LeafReference is used. The Image is included in a single path as a DiagramElement. For any other path in which it is wanted to be displayed, a LeafReference is included. Its reference points to the Image which indicates that it is part of both paths and is being rendered accordingly.

Semantic model

This diagram interchange metamodel allows diagrams to be represented with an additional semantic meaning by linking an element of an existing semantic model to a GraphElement via the abstract SemanticModelBridge. Each GraphElement has a link to a concrete subclass of SemanticModelBridge. If it is not a SimpleSemanticModelElement, it references the topmost super-class of the linked semantic model. For UML 1.x this is the class Element from the package Core, which is referenced by Uml1SemanticModelBridge. For UML 2.0 and later, this can be the class Element from the package Kernel, referenced by CoreSemanticModelBridge. This link allows the addition of UML-specific information to the graph. Other semantic models might be added as well, e.g. ER diagrams.

The link to the SemanticModelBridge is a unidirectional link. There is no need for elements of the semantic model to have a link to their representation elements, thus making it is unnecessary to effect any changes to the semantic model.

The model is designed to minimize the amount of redundant information. Due to this motivation, there is no extra attribute to indicate the semantic type of an element. In order to find out the semantic type of an element, the SemanticModelBridge must be examined. If the concrete subclass has a link to an element of the semantic model, all available semantic information about this element can be obtained.

Pre-defined presentations of elements

Some elements allow more than one standard presentation. For example, an actor may be shown as a rectangle or as a stickman graphic. In order to distinguish these cases, the attribute 'presentation' defined in SemanticModelBridge indicates the desired presentation. This avoids the need to create complex DiagramElements such as Images to display standard elements.

To achieve the standard visualization of an element, presentation must be set to '', the empty string. For a non-standard treatment, it must be set to 'UserDefined'. This means that it is shown exactly as specified through the DiagramElements contained in the GraphElement associated with the Element. As most elements have exactly one standard presentation, table 7.2 lists only those explicitly which have more than one. The others are collected under the term [Single Presentation].

Element	**SemanticModelBridge.presentation**
[Single Presentation]	'' (Default), 'UserDefined'
Actor	'' = 'Stickman', 'Rectangle', 'UserDefined'
Interface	'' = 'Rectangle', 'Circle', 'Lollipop', 'UserDefined'

Table 7.2: Permissible values for SemanticModelBridge.presentation

Additional semantic information

GraphElements which have no corresponding semantic model element are linked to the subclass SimpleSemanticModelElement. Its attribute *typeInfo* contains semantic information for the related GraphElement. The following rules define the usage of the SimpleSemanticModelElement:

- A compartment generally does not have a corresponding model element in UML but is visualized by a GraphNode. For example, the GraphNode of an attribute compartment is linked to a SimpleSemanticModelElement with the typeInfo AttributeCompartment. For an operation compartment this would be OperationCompartment and so on.
- For an attribute part, the representing GraphElement is linked to a SimpleSemanticModelElement where its typeInfo is set to the name of the

attribute. For example, for the attribute *visibility* of the class Feature in package Core the typeInfo is *visibility*.

- Diagrams have a typeInfo set to the name of the corresponding UML diagram. For example, a class diagram has the typeInfo *ClassDiagram*, a state diagram has the typeInfo *StateDiagram* etc.

Table 7.3 contains a list of special GraphElements representing elements which do not have a model element in UML and their corresponding typeInfos.

GraphElement	**SimpleSemanticModelElement.typeInfo**
Active part of a lifeline	active
Cross at the end of a lifeline	destroy
Header of a lifeline	header

Table 7.3: Permissible values for SimpleSemanticModelElement.typeInfo

Representing the different diagram types of UML

The diagram interchange model can be used with the UML metamodel to represent any UML diagram type through a graph with semantic links to the UML metamodel. For most of the UML diagram types there is an intuitive way to represent them as a graph. Sequence diagrams, however, are somewhat different as discussed below.

In a class diagram, for example, classes, interfaces, and packages are represented by GraphNodes, while associations, generalizations, and dependencies are represented by GraphEdges. These GraphNodes and GraphEdges 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 a link to a SimpleSemanticModelElement, as compartments are not part of the UML metamodel, being only part of its representation. An attribute or operation is a nested node of the compartment node with a link to the corresponding attribute or operation of the UML metamodel. An attribute itself has attributes such as visibility and type. These parts of the attribute are represented by LeafElements. If an attribute part is represented as text, the subclass TextElement is employed, otherwise an Image or GraphicPrimitive is used.

The appearance of the end of an edge, e.g. the composite of an association end, generalization etc., is determined by the corresponding UML metamodel elements.

A special case is the n-ary association. It consists of a GraphNode representing the diamond which links the parts of the association. These parts again are shown as GraphEdges. Each of these is linked via a SemanticModelBridge to the same corresponding association. This means that several DiagramInterchange model elements (the GraphEdges) are utilized to visualize a single UML model element (the association). Figure 7.7 illustrates this mapping.

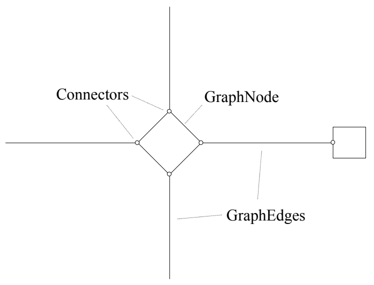


Figure 7.7: An n-ary association

Other diagrams are represented similarly. The tool which makes use of 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 elements of the metamodel.

An object in a sequence diagram is modeled as a GraphNode with a semantic model bridge to the corresponding instance. The height of this GraphNode is the entire height from the lifeline including the object at the top of the lifeline. The dashed lifeline is the only visible element of this GraphNode. The contained DiagramElements of the GraphNode are GraphNodes which represent the activated section of the lifeline. Such a GraphNode has a corresponding SimpleSemanticModelElement with the typeInfo *active*. Only one child of the top GraphNode has a corresponding SimpleSemanticModelElement with the typeInfo *header* to represent the rectangle at the top of the lifeline. The GraphNode with the TextElement is nested into this GraphNode. Another child of the top GraphNode can be a GraphNode to represent the cross at the end of a lifeline (see figure 7.8). The cross is represented by a LeafElement. The arrows between the lifelines are GraphEdges with the stimulus as corresponding model element. The arrowhead of such a GraphEdge is not modeled explicitly, yet the type of the arrowhead can be found out through the corresponding action.

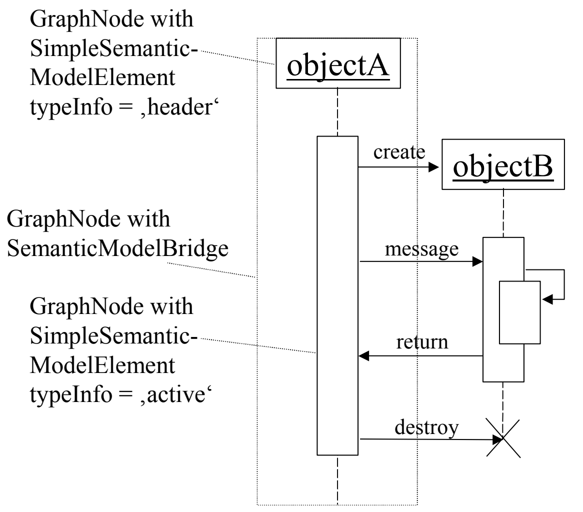

GraphNode with SimpleSemantic-ModelElement typeInfo = ‚header'
objectA
create
objectB
GraphNode with SemanticModelBridge
message
GraphNode with SimpleSemantic-ModelElement typeInfo = ‚active'
return
destroy

Figure 7.8: Sequence diagram

InteractionOccurrences can overlap multiple lifelines and other elements of sequence diagrams. To indicate that the elements below the InteractionOccurrence are visible through the InteractionOccurrence the Translucent property of the GraphNode of the InteractionOccurrence is set to *true*.

Components with ports and interfaces

Components are able to contain ports and interfaces which may be required or provided. Components and their contained elements are represented as GraphNodes. The interfaces are contained either in a port if one should exist or directly in the component itself. Whether the circle showing the interface is open or closed is determined solely through the semantic model, i.e. it depends upon whether the interface is required or provided. Interfaces visualized through a line with an open or closed circle are distinguishable from those showed as a rectangle by setting the attribute *representation* of the SemanticModelBridge to *lollipop*. The GraphNode of an interface shown as *lollipop* contains a GraphEdge for the line of the Lollipop and a GraphNode for the circle as shown in Figure 7.9.

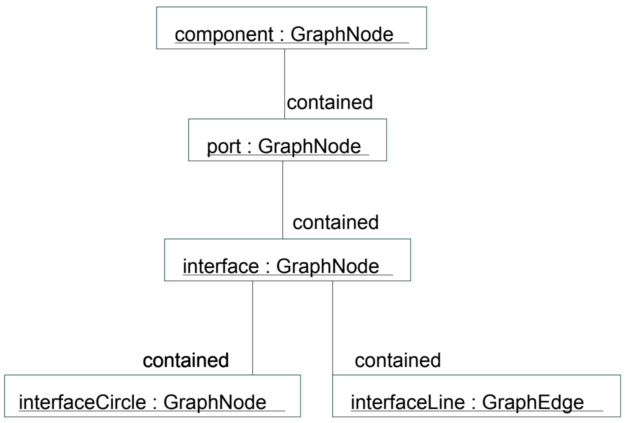


Figure 7.9: Component

8 Derivation of presentational views

Expressed in XMI, a model can be interchanged between tools that are aware of model elements. However, this format is not very consumable for the human reader neither is it well suited for tools that are purely graphically oriented. Therefore a transformation into a graphical format is needed. The format most promising for this purpose is SVG. The format proposed in this submission is well prepared 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 is such a mechanism that is designed to transform one XML format into another. For proof of concept, this proposal includes a set of XSLT scripts to transform a model given in XMI[UML+DI] into SVG. At the current stage this is applied to information regarding class diagrams only.

As set out in the following paragraphs, these stylesheets extract information from an XMI file with model and DI data and build a new SVG document out of this. The resulting SVG file contains the representation of a single class diagram in UML notation.

The next section describes the transformation concept for class diagrams. This is followed by a section containing the discussion about further extensions of the stylesheets. This includes, for instance, a suggestion how to deal with DI files which store several diagrams and a discussion about transformation of other diagram types. The stylesheets are designed to generate different notations with small modifications. More about this issue can be found in the last section.

8.1 General transformation concept

A transformation using XSLT scripts is based on a set of templates. In our implementation we need one template which matches the first class diagram in a diagram interchange file. All other information such as model data is used later on to obtain single values such as class or attribute names (script: xmi2svg.xsl).

Once a class diagram is matched, a named template is implemented to generate the SVG header on some standard elements which can be reused for the diagram elements to be created later on (script: build_diagramm.xsl).

With the usage of <apply-templates> the XSLT processor will iterate over all child elements of the diagram. These children can be node elements such as a class or an association-end or edge elements such as associations or generalizations. Templates which match these elements can be found in build_nodeelem.xsl and build_edgeelem.xsl.

The templates of build_nodeelem.xsl and build_edgeelem.xsl need to know about the kind of model element which is associated with the node or edge element matched. To obtain this information, we need to follow the link attribute. This is done by means of a couple of named templates in util_modellookup.xsl.

The model element type (e.g. class or generalization) is used as condition for calling other named templates for generating SVG elements. These named templates can be found in build_class.xsl, build_assozend.xsl, and build_edge.xsl. Each of them uses util_modellookup to get the text elements, e.g. class or operation name, which are to be represented.

Another kind of information which needs to be extracted is the model information on the association ends if an edge is represented in the diagram. In this case the scripts call a lookup routine for each of the edge ends and generate an SVG element based on the attributes returned. For instance, a navigable flag might be set for an edge end. This will lead to a generation of an solid arrow if the other edge end does not have a navigation flag set.

The representation of an edge is very simple when using the <path> element of SVG. The coordinates of the points of an edge need to be written into an attribute of the path element. To do so, we created a template which matches all points and generates a string (util_createpath.xsl).

For example, the XMI code for the waypoints of a single edge is as follows:

```
<UML:GraphEdge.waypoints>
  <!-- waypoints[0] -->
  <XMI.field>    <!-- base -->
    <XMI.field>597.5</XMI.field>    <!-- x -->
    <XMI.field>304.0</XMI.field>    <!-- y -->
  </XMI.field>
  <XMI.field>    <!-- control1 -->
    <XMI.field>0.0</XMI.field>
    <XMI.field>0.0</XMI.field>
  </XMI.field>
  <XMI.field>    <!-- control2 -->
    <XMI.field>0.0</XMI.field>
    <XMI.field>0.0</XMI.field>
  </XMI.field>
  <!-- waypoints[1] -->
  <XMI.field>    <!-- base -->
    <XMI.field>597.5</XMI.field>    <!-- x -->
    <XMI.field>340.0</XMI.field>    <!-- y -->
  </XMI.field>
  <XMI.field>    <!-- control1 -->
    <XMI.field>0.0</XMI.field>
    <XMI.field>0.0</XMI.field>
  </XMI.field>
  <XMI.field>    <!-- control2 -->
    <XMI.field>0.0</XMI.field>
    <XMI.field>0.0</XMI.field>
  </XMI.field>
  <!-- waypoints[2] -->
  <XMI.field>    <!-- base -->
    <XMI.field>152.84375</XMI.field>    <!-- x -->
    <XMI.field>705.0</XMI.field>            <!-- y -->
   </XMI.field>
   <XMI.field>   <!-- control1 -->
    <XMI.field>0.0</XMI.field>
    <XMI.field>0.0</XMI.field>
   </XMI.field>
   <XMI.field>   <!-- control2 -->
    <XMI.field>0.0</XMI.field>
    <XMI.field>0.0</XMI.field>
   </XMI.field>
  </UML:GraphEdge.waypoints>
```

Each waypoint is a BezierPoint. Since this edge is not curved, the x and y values of the control points are zero. This edge is transformed to the following SVG commands:

"M597.5,304.0 L597.5,340.0 L152.84375,705.0"

This string includes draw commands for a path. These commands are MoveTo-Point(597.5, 304.0) then DrawLineTo-Point(597.5, 340.0) then DrawLineTo-Point(152.84375, 705.0).

As set out in the paragraphs above, all information for generating the SVG can be taken from diagram interchange or model interchange data. The linkage of corresponding elements makes it easy to write XSLT scripts for a general transformation to SVG.

8.2 Extensions to the style sheets

In the process of developing the XSLT scripts for this proof of concept we thought about extensibility of the scripts to support the following features:

(1) Support of other diagram types and further elements such as interfaces, packages, parameterized classes, association classes, and dependencies (a complete list of all the elements which need to be supported can be found in the UML Notation Guide).

This means that the model lookup routines would need to be extended for the new model element types and for the handling routines for these types. Also additional named templates will have to be written for generation of a graphical representation in SVG for these new elements.

(2) Support of XLink. In the current implementation a linkage between model and diagram interchange elements is only possible through usage of simple IDRefs. For this reason, only XMI files which contain both model interchange and diagram interchange data can be transformed. This is not acceptable for a final solution.

XSLT allows access of different files using the <xsl:for-each> command (<xsl:for-each select="document('other-document.xml')">*). For a support of special cases of XLinks where only a document and an IDRef is used, an implementation could employ the string functions of XPath to extract link information. A general solution of XLink should be possible with the usage of a XLink library by an XSLT extension. All templates which need the XLink support can be found in util_modellookup.xsl.*

(3) Last but not least there is a need for support of transformation for files which contain more than one diagram—which is the most common case.

To accomplish this, one of the following solutions might be implemented:

- *A parameterized set of scripts which still only extract one diagram is a viable alternative. An external application controls the calls of these scripts for the different diagrams. One parameter selects the extracted diagram.*
- *With the use of XSLT extensions, a generation of several output files can be implemented. With this feature an SVG file for each diagram could be generated.*
- *A solution could be a dynamic SVG which contains all diagrams and a user interface to switch between the diagrams in the SVG browser.*

8.3 Transformation to other notations

For the proof of concept we implemented only the transformation to one graphical representation as defined in UML Notation Guide. Generally any notation is possible as long as no renderer algorithm is required. This allows modeling tools to provide XSLT scripts which generate SVG with their own notation. For example, some tools draw shadows for classes and objects or make use of colors.

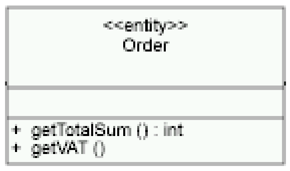
For a notation where position of graphical elements change, a renderer is needed. This renderer might be connected to the stylesheets via XSLT extensions. However, in this case, XSLT might not be the best solution.

With the following example we wish to show how simple a change to the scripts for a different notation might be. In the example given we change the representation of a class element.

For the generation of the class element the script *build_class* is responsible. We extracted the lines of interest from this stylesheet:

```
    <g style="fill:none;stroke:black;stroke-width:1">
      <rect height="{$height}" width="{$width}" style="fill:white"/>
      <xsl:if test="$draw_attributes">
        <line y1="{$Attrib_PartStart}"  y2="{$Attrib_PartStart}" x2="{$width}"
style="fill:none;stroke:black;stroke-width:1"/>
      </xsl:if>
      <xsl:if test="$draw_operations">
        <line y1="{$Operation_PartStart}" y2="{$Operation_PartStart}" x2="{$width}"
  style="fill:none;stroke:black;stroke-width:1"/>
      </xsl:if>
    </g>
```

With this code, a class looks the way it is defined in the UML Notation Guide. A solid-outline rectangle with three compartments separated by horizontal lines

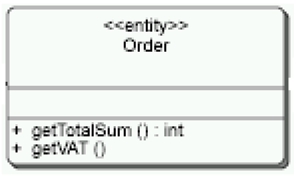


will be viewed in a SVG browser.

With only a small change in this code, a class with shadow and round edges can be viewed.

```
    <g style="fill:none;stroke:black;stroke-width:1">
```

```
      <rect x="2" y="2" rx="4" ry="4" height="{$height}" width="{$width}" style="fill:grey"/>
      <rect rx="4" ry="4" height="{$height}" width="{$width}" style="fill:white"/>
      <xsl:if test="$draw_attributes">
        <line y1="{$Attrib_PartStart}"  y2="{$Attrib_PartStart}" x2="{$width}"
style="fill:none;stroke:black;stroke-width:1"/>
      </xsl:if>
      <xsl:if test="$draw_operations">
<line y1="{$Operation_PartStart}" y2="{$Operation_PartStart}" x2="{$width}"
  style="fill:none;stroke:black;stroke-width:1"/>
      </xsl:if>
    </g>
```



9 Representation in SVG packaging meta-information into SVG diagrams

This chapter provides a short overview on how the meta-information for diagram interchange is represented in a generated SVG file.

As defined in the UML Notation Guide, a class diagram is a graph of *Classifier elements* connected by their various static relationships (see UML 1.4 - Chapter 3 - Notation Guide – Section 3.19.2). In our implementation, this includes classes, associations, and generalizations. Each of these elements are represented by a group in the generated SVG file. A group has no graphical representation but is used to embrace the graphical elements of a model element. For instance, a class group contains rectangles, text elements, and lines.

Following the notation guide, a class is drawn as a solid-outline rectangle with three compartments separated by horizontal lines (see Notation Guide 3.22.2).

SVG

```
<!--Class Position-->
<g style="font-size:9;font-family:dialog;" onmousemove="removePopup()"
      onclick="class_click(evt,'Class Position')"
      transform="translate(184,458)">
      <g style="fill:none;stroke:black;stroke-width:1">
          <rect style="fill:white" width="88" height="61"/>
          <line style="fill:none;stroke:black;stroke-width:1" x2="88" y2="20"
              y1="20"/>
          <line style="fill:none;stroke:black;stroke-width:1" x2="88" y2="41"
              y1="41"/>
      </g>
      <!-- top name compartment -->
      <text style="text-anchor:middle" dx="88" dy="9">Position</text>
      <!-- attribute compartment -->
      <g transform="translate(0,20)">
          <text dx="2" dy="9">#<tspan x="12">posnum</tspan> : int</text>
      </g>
      <!-- operation compartment -->
      <g transform="translate(0,41)">
          <text class="standardfont" dx="2" dy="9">+
          <tspan x="12">getPosnum</tspan>()
          </text>
      </g>
</g>
```

The top name compartment holds the class name and stereotype. Class name and stereotype are aligned in the middle of the name compartment. The stereotype value is extracted from the model and enclosed in guillemets «». Additional icons for special stereotypes and a list of properties for the class are not supported by our current XSLT scripts, extracting this information from an external icon file and the model is straightforward.

In our implementation only attribute and operation compartments are displayed, yet there is no restriction to extend the representation with compartments for exceptions or requirements. If the compartment attribute *hidden* is set, no elements of this compartment will be generated into the SVG document.

An attribute compartment is defined in Notation Guide Section 3.25.2 as represented as a list of attributes. We decided not to display a compartment name or group properties. This representation can be found in most UML tools.

Not only the order but also the graphical positioning and visibility of the attributes are taken from the diagram interchange data. Attribute visibility symbol, name, type, and initial value are represented in the SVG document. Each attribute is encapsulated in separate SVG text elements.

Example: `#posnum: int`

`SVG:` `<text dx="2" dy="9">#<tspan x="12">posnum</tspan> : int</text>`

The SVG representation of the operation compartment is effected similar to the attribute compartment as a list of text elements. Each text element contains operation visibility, name, a list of parameters in braces, and the return type as defined in Notation Guide Section 3.26.2.

Example: `getPosnum( )`

`SVG:` `<text dx="2" dy="9">+ <tspan x="12">getPosnum</tspan>() </text>`

Apart from the class element we decided, for the proof of concept, to represent associations and generalizations to show how path elements are handled. As defined in the Notation Guide, paths consists of a series of line segments. Associations and generalizations are paths with an optional attached name and special symbols at the ends of the path.

For associations in our current implementation of XSLT scripts we allow the representation of navigability and aggregation indicators following the Notation Guide Section *3.43.2*. Textual elements such as role name and multiplicity are encapsulated in a group for each association end. This group contains only text elements as given by the diagram interchange data.

We define and make use of SVG elements for the aggregation symbol, navigability (an arrow), and generalization (large hollow triangle). These elements can be reused at any association or generalization end, necessitating only that a rotation angle be calculated according to the direction of the last segment they are placed on. To do so, we needed an XSLT extension to calculate the square root, which is not possible in XPath expressions and XSLT functions.

In this chapter we described how class diagrams are represented in SVG as defined in the UML Notation Guide. We have employed the grouping mechanism to embrace graphical elements which belong to the same model element. This allows us to add features such as a tooltip on classes which are shown in an SVG browser.

Part III

10 Full example

An excerpt from a saved file is attached in Appendix B.

A full example can be downloaded from the web page cited below:

http://www.gentleware.com/projects/diagraminterchange/index.php3

11 Summary of optional versus mandatory interfaces

This proposal does not suggest an interface but a metamodel for diagram information and a saving format of the information. For this reason, this point is

Not applicable.

12 Proposed compliance points

13 Changes or extensions required to adopted OMG specification

No extensions to other adopted OMG specifications are needed.

14 Complete IDL definitions

This proposal does not suggest an interface but a metamodel for diagram information and a saving format of the information. For this reason, this point is

Not applicable.

Appendix A

15 Assignment of diagram elements

The following table is based on the current work of the U2P consortium for the superstructure of UML 2.0. It describes which element is represented by a GraphNode or by a GraphEdge. For some elements with a more complex representation, the nested elements are set in brackets '[]'. This table will be updated during the finalization process.

Element	DiagramElement	Diagram
ActionOccurrence	GraphNode	Interaction
Actor	GraphNode	Use Case
Aggregation	GraphEdge	Class/Package/Object
Artifact	GraphNode	Deployment
Association	GraphEdge	Class/Package/Object
Class	GraphNode	Class/Package/Object
Class template	GraphNode	Class/Package/Object
Collaboration	GraphNode	Composite
CollaborationOccurrence	GraphNode	Composite
CombinedFragment	GraphNode	Interaction
Component	GraphNode	Component
Component has Port	GraphNode[GraphNode]	Component
Composition	GraphEdge	Class/Package/Object
Connector	GraphEdge	Composite
Connector (Assembly)	GraphEdge[GraphNode]	Component
Coregion	GraphNode	Interaction
Dependency	GraphEdge	Class/Package/Object
Dependency	GraphEdge	Deployment
DeploymentSpecification	GraphNode	Deployment
Extend	GraphEdge	Use Case
ExtensionPoint	GraphNode	Use Case
FinalState	GraphNode	State
Frame	GraphNode	Interaction
Generalization	GraphEdge	Class/Package/Object
Generalization	GraphEdge	Deployment
GeneralOrdering	GraphEdge	Interaction
Include	GraphNode	Use Case
InstanceSpecification	GraphNode	Class/Package/Object
Instantiation	GraphEdge	Deployment
InteractionOccurrence	GraphNode	Interaction
Interface	GraphNode	Class/Package/Object
Lifeline	GraphNode	Interaction
Message	GraphEdge	Interaction
Node	GraphNode	Deployment
Package	GraphNode	Class/Package/Object
PackageExtension	GraphEdge	Class/Package/Object
PackageImport (private/public)	GraphEdge	Class/Package/Object
Part	GraphNode	Composite
Port	GraphNode	Component
Realization	GraphEdge	Class/Package/Object
Role binding	GraphEdge	Composite

State	GraphNode	State
Stop	GraphNode	Interaction
Transition	GraphEdge	State
Use Case	GraphNode	Use Case

The table below lists diagram elements which are not modeled in the UML metamodel but have a explicit representation in diagrams. They are identified by their typeInfo in the SimpleSemanticModelElement.

SemanticModelElement.typeInfo	**DiagramElement**
NameCompartment	GraphNode
AttributeCompartment	GraphNode
OperationCompartment	GraphNode
ClassDiagram, StateDiagram, …	Diagram
Name	GraphNode
Visibility	GraphNode
TypeSeparator	GraphNode
InitialValue	GraphNode
Multiplicity	GraphNode
Ordering	GraphNode
InterfaceCircle	GraphNode
InterfaceLine	GraphEdge

Appendix B

16 Excerpt from an XMI[DI] example

To get an impression of the saving format suggested in this proposal, this appendix contains an excerpt from an example file with the XMI[DI] information. As example, the UML model of the diagram interchange metamodel itself is used as depicted in Figure 7.1. Due to differences between MOF and UML, the structure types of the metamodel are modeled as classes in this example.

```
<?xml version = '1.0' encoding = 'ISO-8859-1' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Thu Sep 05 18:53:29 CEST 2002'>
  <XMI.header>
```

To be provided…

This example is cut off at this point. The overall length of the XMI file would fill several dozen pages. If desired, the full example file can be downloaded from the site given below:

http://www.gentleware.com/projects/diagraminterchange/index.php3.