A Model-Driven Approach for Graph Visualization

Celal Çığır, Alptuğ Dilek, Akif Burak Tosun

Department of Computer Engineering, Bilkent University

06800, Bilkent, Ankara

{cigir, alptug, tosun}@cs.bilkent.edu.tr

Abstract

Graphs are data models, which are used in many areas from networking to biology to computer science. There are many commercial and non-commercial graph visualization tools. Creating a graph metamodel with graph visualization capability should provide interoperability between different graph visualization tools; moreover it should be the core to visualize graphs from different domains. A detailed and comprehensive research study is required to construct a fully model driven graph visualization software. However according to the study explained in this paper, MDSD steps are successfully applied and interoperability is achieved between CHISIO and Graphviz visualization tools.

Keywords

Graph visualization, model-driven software development, UML, Ecore, Model-to-Text transformation, Model-to-Model transformation.

1. Introduction

Graphs are simply a set of vertices or nodes plus a set of edges that connect these nodes to each other. They are useful data structures to capture the relations (represented by edges) between the objects (represented by nodes). From network to data flow charts, from bioinformatics to computer science, many real problems can be modeled, simulated and visualized by using this simple but powerful data structure.

Graph visualization is a research area for producing visual representations of any data using graphs by deriving (or using) the topological information lying under the model. Understandability of these visual drawings must be high for easy interpretation and analysis. Geometrical information of the produced graph holds important measures for better graph visualization. Locations, sizes and relative positions of the nodes are part of the geometrical information and they should be adjusted either manually or automatically in order to produce an understandable and clear graph. This operation is called "graph layout". Many complex graphs can be laid out in seconds using automatic graph layouts.

Many different graph visualization tools are available either commercially or freely and they present a variety of features. Among one of these, CHISIO [1] is general purpose graph visualization and editing tool for proper creation, layout and modification of graphs. CHISIO includes compound graph visualization support among with different styles of layouts that can also work on compound graphs. Another example for graph visualization software is Graphviz which consists of a graph description language named the DOT language and a set of tool that can generate and process DOT files [2] . Like CHISIO, Graphviz also supports different graph layouts such as circular layouts, radial layouts and etc.

However, each graph visualization tool uses its own model to represent the graph data structure and these models not always have one-to-one correspondence with each other. Moreover nearly each tool has its own file format in order to store and process the graphs. For instance CHISIO has support for files formatted with respect to GraphML [3] specification, but cannot open *.dot* files. In the same way, Graphviz can open the files that have *.dot* file extensions but cannot process *.graphml* files. This problem is also valid among different graph visualization tools.

In order to solve this problem and provide a bridge between models of graph visualizations tools, there should be a common extended graph model with graph visualization capability. In order to create this extended graph model, following the model-driven software engineering concepts is a good choice in terms of identifying the domain concepts easily and making the transformations from one domain to another domain without difficulty and much effort.

Model-driven software development (MDSD) is an evolving technology in software development, which aims to generate code automatically from the models (documentation) hence providing the lacking synchronization between the two. The starting point in the approach is to determine the domain to apply MDSD methodologies, to perform the domain analysis and to identify the domain concepts. The metamodel, the heart of MDSD process, will have been constructed at the end of the phases stated.

In this study, we aim to construct an extended graph model with graph visualization capability by following the MDSD concepts. Therefore, our model can be used as a bridge between different graph visualization tools. Since MDSD provides model-to-model and model-to-text transformations, one can easily transform its graph or graph visualization model to another graph visualization model. Therefore, interoperability [4] among different graph visualization tools can be possible with the help of MDSD.

The remainder of this paper is organized as follows: Domain analysis is stated in Section 2. Grammar for Domain Specific Language (DSL) is presented in Section 3. Definition of metamodel based on MOF from scratch including abstract syntax, concrete syntax, static semantics and basic example models derived from the constructed metamodel can be found in Section 4. Section 5 presents definition of metamodel using UML profiling. Model-to-text and model-to-model transformations including example transformations can be found in Section 6 and 7. Discussions and lessons learned are presented in Section 8. Finally, conclusion and future work can be found in Section 9.

2. Domain Model and Domain Concepts

As stated in Section 1, graphs are useful data structures used in many different domains for different purposes. They are used in many theoretical areas to perform graph specific algorithms and many practical areas due to being a good way of representation inherently. The visualization of graphs is important since it is often helpful in revealing the infrastructure of the underlying model. Graph visualization software tools are especially used in visualizing social networking, computer networks, pathways in bioinformatics, etc. The graph visualization domain itself has applications in different domains, thus it is crucial to construct a metamodel of the domain to fulfill the needs of different application domains. The concepts derived will be explained in order below.

The basic elements of graph visualization domain are graph, node and edge. Before going into the details of these terms, it is important to identify the commonalities of them in practical use and make an abstraction. The abstraction containing the common properties is called the *GraphObject* in the domain terminology. Thus a graph object can either be a graph, node or edge and it has properties such as name (just a String literal), type (to be explained in detail in the section dedicated to concrete syntax) and label (to be explained in this section).

The formal and simple definition of a *Graph* is a kind of data structure that consists of a set of nodes and a set of edges that establish relationships (connections) between the nodes [5]. The graph concept in our model has a children node list. The children edge list is not to be hold as a parameter, but rather to be constructed on demand. For an edge to be a child of a graph, both of its end nodes must be in that graph. A graph can have either one owner compound node (will be explained in this section) or none, which is the case if the graph is the top most graph called root graph.

Node is the basic unit of a graph to represent entities. With the domain model, we would like to model both directed (in which the edges has separate source and target node ends) and undirected graphs so the node has out-edges (for which the node is the source for the edge) and in-edges list (for which the node is the target for the edge). Every node has an owner graph in our domain. Node has node properties, which determines its appearance and it will be explained below. Other properties of node are the top-left corner point, height and the width.

Edge is the unit of graph that represents the relationship between the entities. Every edge has one and only one source and target node, the owner graph of an edge is the common top-most graph containing its source and target nodes. However, this owner is not kept as property and might be calculated if needed. One other property of an edge is the bend point list, which determines the routing of the edge from the source node to the target node.

Compound node is a specialized node (extends from the node), which has a subgraph and hence other nodes within itself. The compound nodes provide nesting property for graphs, which are very useful for many application domains. For instance local area networks in computer networks or complexes in bioinformatics domain are instances of compound nodes.

Label is kind of a utility object in the domain, which cannot exist by its own but must be associated with a graph object. A label contains text, which is used to display relevant information about the associated graph object. The label can hold properties of the link connecting itself to the owner graph object. A label also has a point to determine its top-left corner within the graph coordinates.

Point is the simple x-y coordinate representation of to position different concepts such as node, edge, etc.

Node properties concept is a collection of properties to determine the appearance of a node such as the shape, transparency, line width, background, border and text colors. Such a concept is needed to differentiate the appearance of different node types modeled in M1 level. More about this issue will be mentioned in Section 4.2.

Edge properties concept is similar to the node properties and it clarifies the appearance of an edge such as the line width, arrow type and color.

Color is the representation of RGB channels used by node and edge properties.

Shape is used by the node properties to determine the shape of a node. This concept is to be represented as an enumeration in the metamodel.

Arrow type is used by the edge properties to determine the arrow head appearance of an edge. Similar to the shape concept, this is also to be represented as an enumeration.

3. DSL Grammar

There are different ways to visualize and express domain concepts. One of them is representing the domain concepts by using grammar such as BNF (Backus-Naur Form), EBNF [6] (Extended Backus-Naur Form) etc. Here is the mapping of domain concepts to grammar in terms of EBNF can be seen in Table 1.

As in the example shown in Table 1; our metamodel is rooted at a GraphObject which can be Graph, Node, Edge or Label. Each of these has its own representation; Graph has zero or more Nodes as well as it can be a CompoundNode; Node has INEDGES that is a list of incoming edges and OUTEDGES that is a list of outgoing edges and references to NodeProperties and Point. An Edge also has reference to its own EdgeProperties and etc.

```
1. GraphObject ::= Label | Graph | Node | Edge
2. Label ::= PointRef
3. PointRef ::= Identifier
4. Graph ::= {Node}* {CompoundNode}?
5. Node ::= INEDGES OUTEDGES
   NodePropertiesRef PointRef
6. INEDGES ::= {Edge}*
7. OUTEDGES ::= {Edge}*
8. NodePropertiesRef ::= Identifier
9. PointRef ::= Identifier
10. Edge ::= EdgePropertiesRef PointRef
    {PointRef}+
11. EdgePropertiesRef ::= Identifier
12. CompoundNode ::= Graph
13. NodeProperties ::= BORDER BACKGROUND
    TEXT
14. BORDER ::= ColorRef
15. BACKGROUND ::= ColorRef
16. TEXT ::= ColorRef
17. ColorRef ::= Identifier
18. EdgeProperties ::= ColorRef
19. LabelRef ::= Identifier
20. GraphObjectRef ::= Identifier
```

Table 1 – Grammar of DSL in EBNF.

4. Metamodeling from Scratch

Metamodels are models that make statements about modeling. As a simple example, for defining metamodels, MOF plays exactly the role that EBNF plays for defining programming language grammars. In this section we will present how we construct our metamodel based on a predefined meta-metamodel ECore.

4.1. Abstract Syntax

We develop our own metamodel using ECore, which is very similar to MOF. We used TOPCASED [7] plug-in of Eclipse [8] to construct and visualize our metamodel.

First we define metamodel GraphObject as an instance of ECore::EClass. Since GraphObject is an instance of Ecore::EClass, then the created children of GraphObject, which are Node, Edge, and Graph, are also instances of ECore::EClass.

Then we added enumerations ArrowType and NodeShape as instances of ECore::EEnum. Finally we created Label, NodeProperties, EdgeProperties, Point, and Color as instances of ECore::EClass. The resulted metamodel M2 level is shown in Figure 1.

In the figure the relationship of these classes with meta-metalevel are not shown for simplicity.

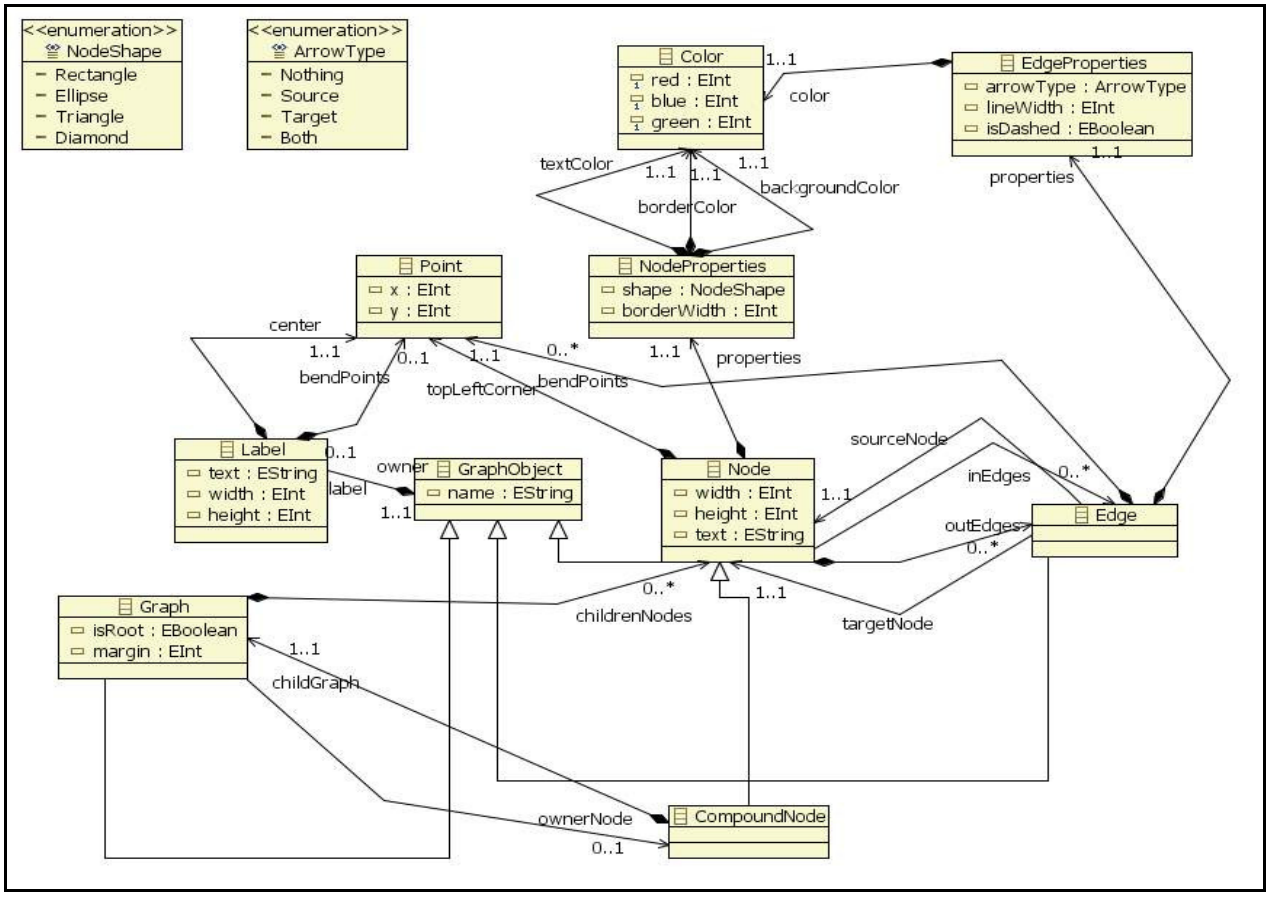

Figure 1 – Metamodeling from scratch (ECore)

4.2. Concrete Syntax

If we were to provide a basic and simple concrete syntax we should have provided a rectangle shape for node and a compound node, a line with a filled-arrow at the target node end for an edge, an ellipse shape for a label and so on. The color for all the concrete syntax elements should have been black. Such a sample concrete syntax might be as in Figure 2.

However, we would like to assign different appearances for different nodes, edges, etc. for the models in M1 level, which conforms to our metamodel in M2 level. This is a situation which is not stated in MDE specifications. The problem is that, the actual concrete syntax for models in M1 level is to be determined differently with respect to the properties of nodes and edges. However, the situation is that, only the default concrete syntax will be defined in M2 level and conditional checks in M1 level will determine the actual appearances of the domain concepts.

The graph concept in the abstract syntax does not have a corresponding concrete syntax, since it is generally the case the graph itself is not visualized but the owner compound node is the one visualized. The graph's concrete syntax is the union of the concrete syntax of the children nodes and the edges attached to these nodes.

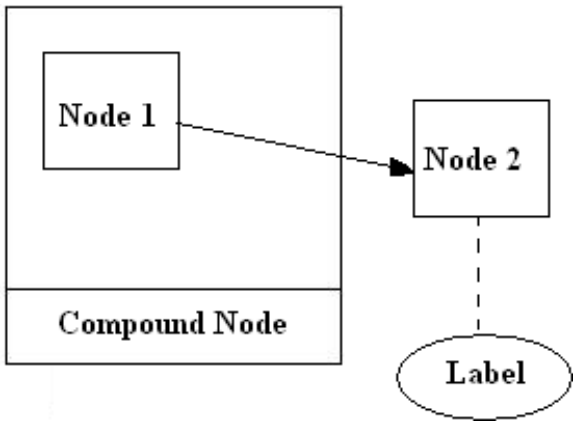

Figure 2 – Sample Concrete Syntax for Node, Edge, Compound Node and Label

4.3. Static Semantics

The OCL (Object Constraint Language) [9][10] is a textual, side-effect free, declarative language to define constraints on UML models. In our project we write our static semantics in OCL-like language named Check which is provided by openArchitectureWare, we did not introduce them in the figures of our metamodels to keep figures as clear as possible. Some of the Checks constructed for our metamodel can be summarized as follows; for example (1) means that every node must have a unique name, (4) every node must have width and height greater than zero, (9,10,11) color object's red, blue, and green attributes must have values between 0 and 255, etc. More can be seen in Table 2.

```
(1) context Graph ERROR "Duplicate node names
detected!" :

        isRoot ?
nodesUnderGraph().forAll(n|n.withSameName(

                nodesUnderGraph())) : true;

(2) context Graph ERROR "The root graph must
not have an owner " +

        " compound node!":

        isRoot ? ownerNode == null : true;

(3) context Node ERROR "Zero length node name
not allowed!" :

        name.length > 0;

(4) context Node ERROR "Width and Height
should be greater than zero "

        +name :

        width > 0 && height > 0;

(5) context Node ERROR "Node Properties should
not be null!":

        this.properties != null;

(6) context NodeProperties ERROR
"BackgroundColor in NodeProperties should not

 be null!":

        this.backgroundColor != null;

(7) context NodeProperties ERROR "BorderColor
in NodeProperties should not be null!":

        this.borderColor != null;

(8) context NodeProperties ERROR "TextColor in
NodeProperties should not be null!":

        this.textColor != null;

(9) context Color ERROR "Red value of Color
should be between 0 an 255!":

        this.red >= 0 && this.red <=255;

(10) context Color ERROR "Blue value of Color
should be between 0 an 255!":

        this.blue >= 0 && this.blue <=255;
```

```
(11) context Color ERROR "Green value of Color
should be between 0 an 255!":

        this.green >= 0 && this.green <=255;

(12) context CompoundNode ERROR "No graph
defined for "+name :

        childGraph != null;

(13) context CompoundNode ERROR "Child graph's
owner does not match "

        + " this compound node " + name + "!":

        childGraph.ownerNode == this;

(14) context Edge ERROR "Source and target
node should not be null!":

        sourceNode != null && targetNode !=
null;

(15) context Edge ERROR "Edge Properties
should not be null!":

        this.properties != null;

(16) context Edge ERROR "Source and target
node of an edge cannot" +

        " be same":

        sourceNode != targetNode;

(17) context Edge ERROR "Edge must exist in
source and target " +

        " nodes' corresponding lists " + name
+ "!":

        sourceNode.outEdges.contains(this) &&

        targetNode.inEdges.contains(this);

(18) context Graph ERROR "The graph's isRoot
must    be set to false; since it has an owner
compound node! " + name:

        ownerNode != null ?isRoot == false :
true;
```

Table 2 – Constraints of metamodel in Check language.

4.4. Example Model

This section includes examples model conforming to our metamodel. The appearance of the nodes, edges and the compound node in the example model are not shown in the model. The properties of the node and edge are to be specified also in the model.

These examples are for showing that our metamodel is capable of modeling such graph oriented representations.

The first example is from the bioinformatics field. In Figure 3, M2 level corresponds to our metamodel and M1 level corresponds to model of biological pathway domain. You can see in the figure that MacroMolecule is a Node. Protein, RNA and DNA are MacroMolecules. SmallMolecule is a Node. Complex is of type CompoundNode in that sample model, whereas the Activation, Inhibition, Product and Substrate are of type

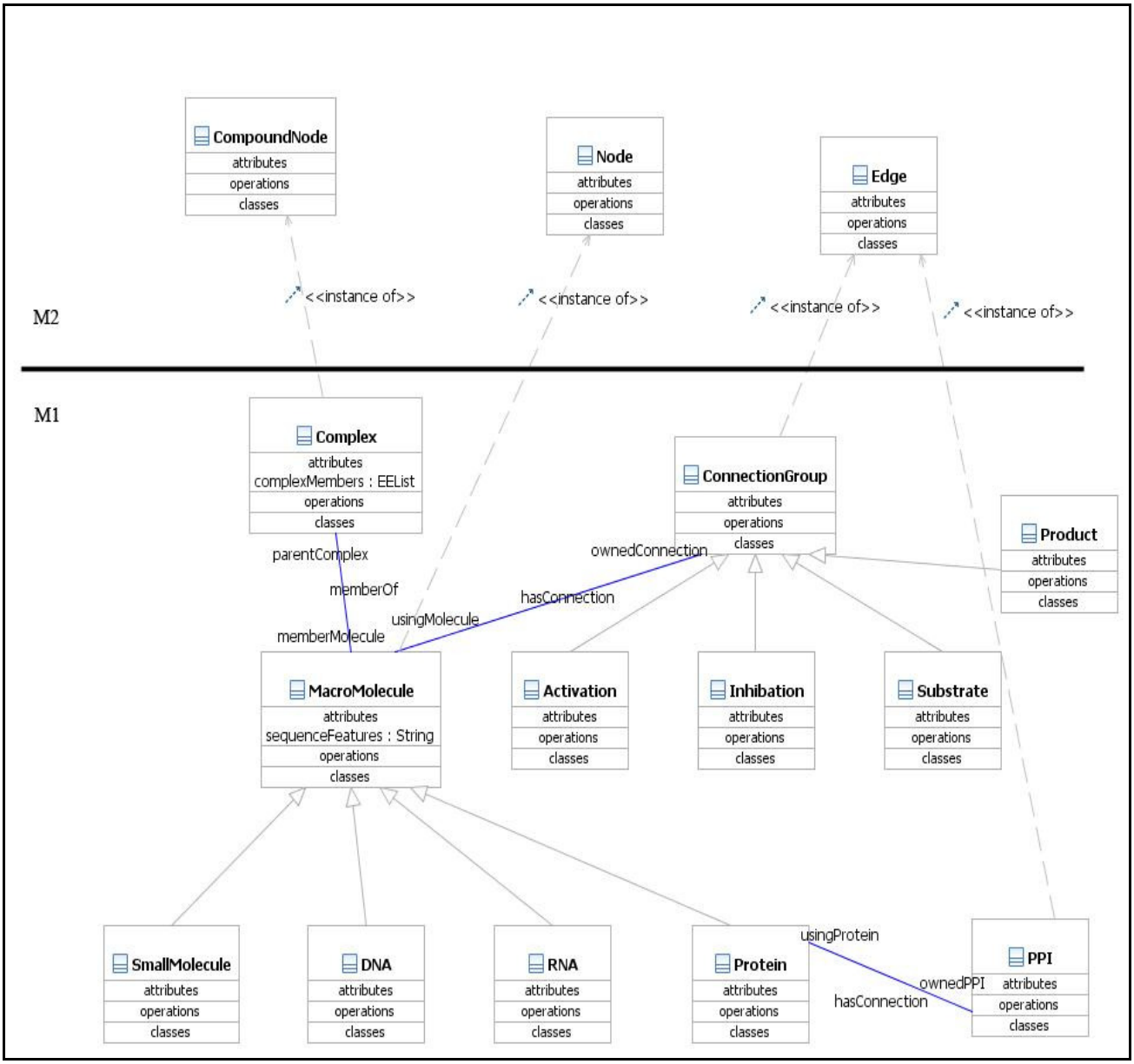

Figure 3 – Biological Pathway example using Graph metamodel.

Edge. It can be seen that Complex molecules correspond to CompoundNode in our metamodel since Complex molecules are consist of more than one MacroMolecule, as CompoundNode consists of more than one Node.

The second example is from computer networks. In Figure 4, you can see in the figure that Computer and Printer is a Node. PC and Server are Computers. LAN is of type CompoundNode in that sample model, whereas the Link is of type Edge. EthernetCable, USBConnection and WirelessLink are Links.

In these two examples, although the domains are totally different from each other, they can be modeled using our graph metamodel.

5. Alternative Metamodel Using UML Profiling

In this study, we also create our metamodel using UML profiling. A profile in the Unified Modeling Language (UML) [11] provides a generic extension mechanism for customizing UML models for particular domains and platforms. Profiles are defined using stereotypes, tagged values, and constraints that are applied to specific model elements, such as Classes, Attributes, Operations, and Activities.

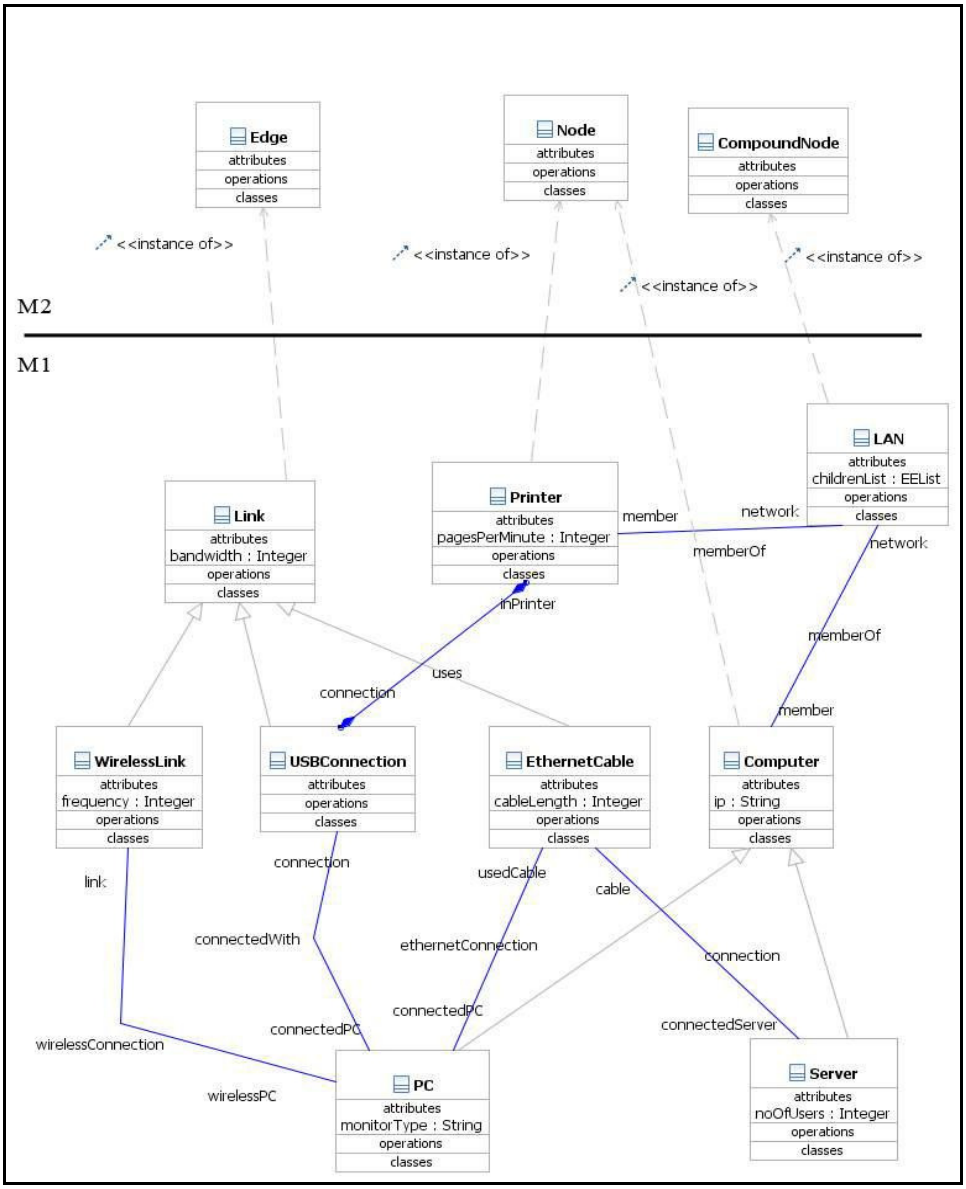

Figure 4 – Computer Network example using Graph metamodel

Lightweight extension of UML2 is used for our profiling schema. UML2 is an EMF [12]-based implementation of the UML 2.x [15] OMG metamodel for the Eclipse platform. UML2 provides

- a useable implementation of the UML metamodel to support the development of modeling tools,
- validation as a means of defining and enforcing levels of compliance,
- support for defining and interchanging profiles

We completed our lightweight extensions by creating profiles/stereotypes and applying them to model elements. Stereotypes can be used to add keywords, constraints, images, and properties (tagged values) to model elements. UML2 stores stereotype applications as dynamic EMF objects.

The metamodel created by using UML2 is shown in Figure 5. In the figure the classes GraphObject, Point, Label, LabelLink, NodeProperties, EdgeProperties and Color are all stereotypes and have extension from UML::Class. Enumeration classes ArrowTypes and NodeShape have extension from UML::Attribute.

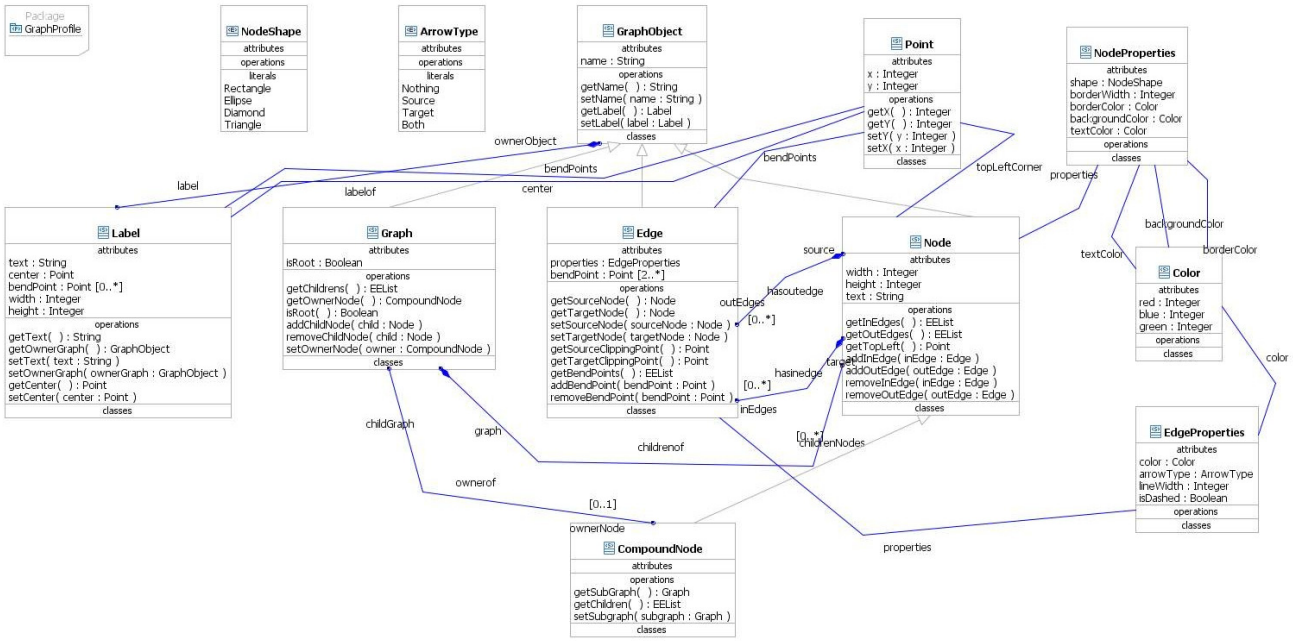

Figure 5 – Metamodeling via UML profiling (UML2 Lightweight Extension)

6. Model-to-Text Transformation

Model-to-text transformation is a 'special' case of model-to-model transformations. It is useful for generating codes as well as non-code documents such as xml, doc etc. In our project we focused on generating a file for representing our graph model. The file format we want to transform our model to is GraphML.

GraphML is an XML-based file format for graphs. It consists of a language core to describe the structural properties of a graph and a flexible extension mechanism to add application-specific data. Unlike many other file formats for graphs, GraphML does not use a custom syntax. Instead, it is based on XML and hence ideally suited as a common denominator for all kinds of services [3]. As a result if we can transform any instance of our metamodel to a GraphML file, we can use our metamodel for any kind of graph domain.

Another reason of choosing GraphML format is that CHISIO is capable of handling GraphML files. So we take the GraphML format for testing our metamodel created for CHISIO.

We set the rules of transformation and template files according to the file system of CHISIO as a result we are able to test our generated GraphML files on CHISIO. There are two examples of transformation in section 6.1.

Our objective in this part is to validate the correctness of our metamodel by transforming our models to GraphML format so that we can test the expressiveness power by visualizing the models in CHISIO.

6.1. Example Transformation

First example is a pathway model. It consists of proteins, small molecules, complexes, physical entities and their relations, which can be seen in Figure 6. We modeled the highlighted graph elements with our metamodel for the sake of visual simplicity.

The model of selected objects and relations are constructed according to our metamodel. Since our metamodel is capable of defining such models, this construction was successful and resulted model is validated with the help of "check" files.

The objects in the model are instances of the concepts defined in our Graph metamodel. For example SmallMolecule is a "Node" which has properties as "NodeProperties" saying that this node is triangle shaped, and has pink color on background; Protein is a "Node" which has properties as "NodeProperties" saying that this node is rectangle shaped, and has green color on background; Cytoplasm (Cellular Location) is a "CompoundNode", Activation link is an "Edge" which has properties as "EdgeProperties" saying that this edge has arrow at target, and has green color, and dashed, etc.

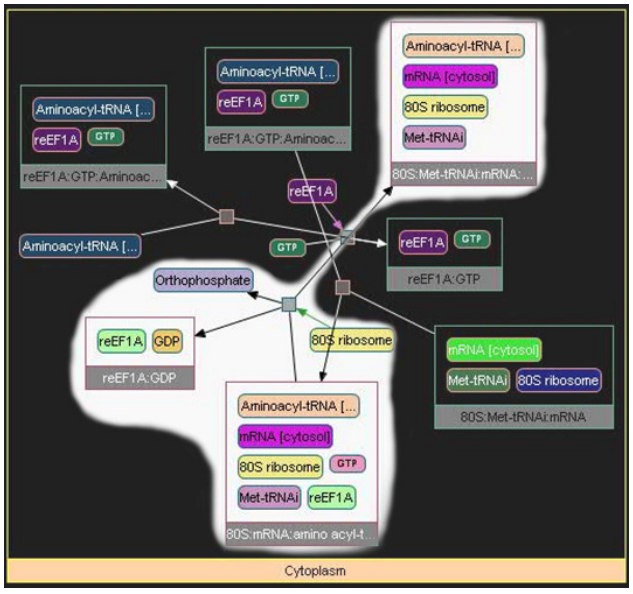

Figure 6 – The Example Pathway Graph

Then we generate the GraphML file according to the example model using a "template" and a "workflow" file. The template is written in Expand language (defined by openArchitectureWare) and it is consisting of any number of IMPORT statements, followed by any number of EXTENSION statements, followed by one or more DEFINE blocks (called definitions).

The metacode in the example in Figure 7 can be seen below. It has facilities to iterate over the elements of the input model (FOREACH), access the properties of the elements, and call other templates (EXPAND).

The workflow controls which steps (loading models, checking them, generating code) the generator executes. Our workflow first loads the model (Pathway Model) in to memory through XmiReader, and generates the code according to the model through xpand2.generator. Finally the GraphML format of the Pathway Model is generated and saved as a GraphML file. We can now open this file with CHISIO to see how our model can be visualized; it can be seen in Figure 8.

```
«DEFINE handleGraph FOR Graph»
    <graph  «IF !isRoot»id="graph«ownerNode.name»«name»">
            «ELSE»id="graph«name»">
        «ENDIF»
        «EXPAND dataWriter(this.margin.toString()) FOR "margin"»
        «EXPAND handleCompoundNode FOREACH childrenNodes»
        «IF isRoot»
            «EXPAND handleEdge FOREACH getAllEdges()»
        «ENDIF»
    </graph>
«ENDDEFINE»
```

Figure 7 – Example Expand code in Template.

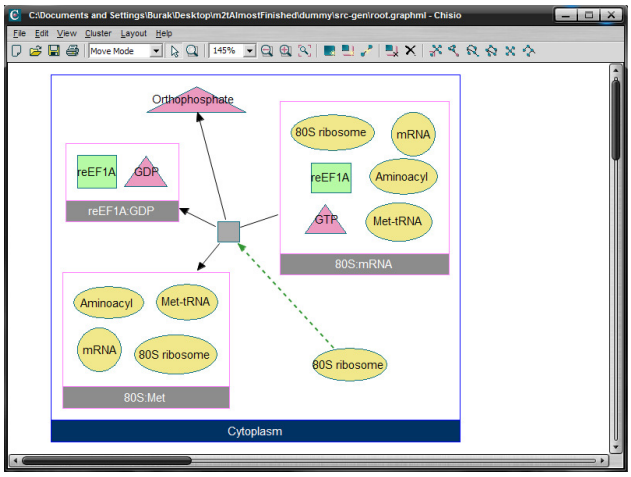

Figure 8 – Generated GraphML Visualization of Pathway Model with CHISIO

The second example is a computer network representing our computers on Bilkent network. The model is again constructed according to our metamodel. Computer and Switch objects are instances of "Node" where EthernetCables, WirelessLinks are instances of "Edge". After constructing the model we generated the GraphML file. The result can be seen in Figure 9, which is showing the visualization of Network model in CHISIO.

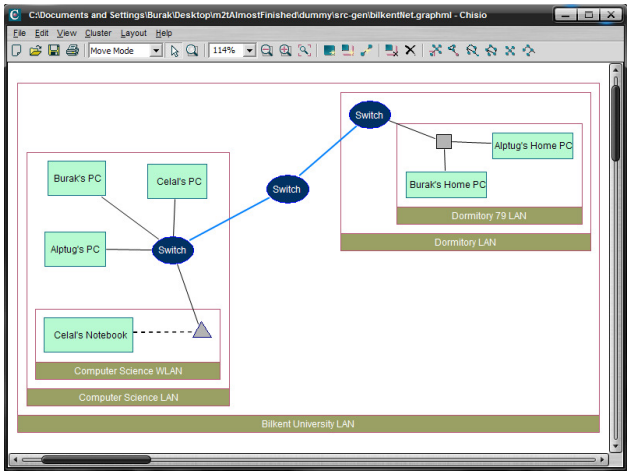

Figure 9 – Generated GraphML Visualization of Network Model with CHISIO

7. Model-to-Model Transformation

Model-to-model transformation, referred as M2M, is one of the key technologies in MDSD and the transformations that can improve the quality of models are important. Since different domains have different models, which may or may not conform to same metamodel,

transformation from one model to another model is usually needed. M2M transformation increases the reusability since developers use the existing models and make little changes on them. There are four types of M2M transformations: Platform Independent Model (PIM) to PIM, PIM to Platform Specific Model (PSM), PSM-to-PIM and PSM-to-PSM [13]. Here, our M2M transformation corresponds to PIM-to-PSM because we transform our generic metamodel to DOT metamodel used in Graphviz.

As a M2M transformation language, we used Atlas Transformation Language (ATL) in this study [14]. Operational context of ATL is given in Figure 10. Here *Ma* is the source model and *Mb* is the target model. *Ma* conforms to *MMa* which is metamodel of source model. Mb conforms to MMb which is the target metamodel. Both MMa and MMb conform to MOF (ECore) metametamodel. Here ATL can be thought as a bridge that provides transformation from source model to target model. In order to make M2M transformation, set of rules needs to be defined in ATL. These rules define the scope of the transformation.

Our motivation for model transformation is to provide a comprehensive enough metamodel to satisfy interoperability between different graph visualization tools.

7.1. Example Transformation

As a M2M transformation, we made transformation from our metamodel to DOT metamodel which is used in Graphviz. Here in Figure 11, the simplified metamodel of DOT can be seen. The basic transformation from our metamodel to DOT model is as follows: We create a DOT Graph for the graph that has isRoot set to true. For each compound node in our model, we create a DOT Subgraph.

We initialize the nodes list of the subgraph created with the childrenNodes of the childGraph of the compound node. For every node we create a DOT Node and for every edge we create a DOT DirectedArc. From NodeProperties and EdgeProperties in our metamodel, we create PolygonNodeShape and ArrowType along with setting the colors of elements in DOT with Color in our metamodel.

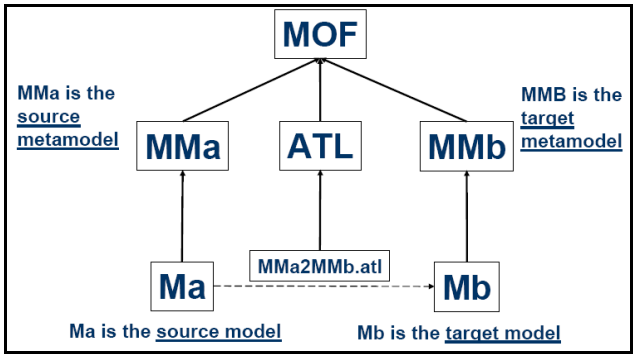

Figure 10 – ATL operational context.

The detail of the model transformation in terms of attribute matching is omitted here, but a sample transformation example written in the corresponding .atl file can be seen in Figure 12.

After our sample model is transformed with respect to the transformation described here, the target DOT model is created. By another M2T transformation we transform the DOT model to a .dot file. As a result, the file can be opened with the Graphviz tool. The DOT model to text transformation is performed via benefiting from a simple ATL Query and using helpers to output the .dot file properly. A sample helper example can be seen in Figure 13, where the properties of a DOT Node are written in to a .dot file.

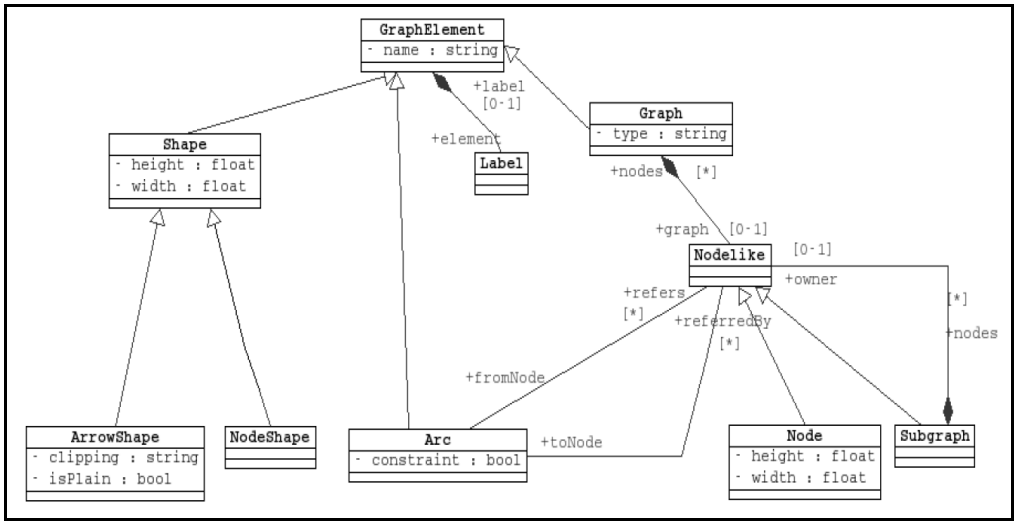

Figure 11 – Simplified DOT metamodel

```
rule Graph2Graph {
    from
        gIn: graph!Graph(gIn.isRoot=true)
    to
        gOut: DOT!Graph (
            compound <- true,
            type <- 'digraph',
            name <- gIn.name,
            nodes <- gIn.childrenNodes)
}

rule CompoundNodeToSubGraph{
    from
        cNode: graph!CompoundNode
    to
        subGraph : DOT!SubGraph (
            name <- 'cluster_' + cNode.name,
            label <- SubGraphLabel,
            labelloc <- 'b',
            color <- '"'+cNode.getColor()+'"',
            style <- 'rounded',
            nodes <- cNode.getChildrenNodes())
        SubGraphLabel: DOT!SimpleLabel (
            content <- cNode.name
        )
}
```

Figure 12 – Sample ATL Transformation Rules

In the examples you can see how we visualize the previously created models by using Graphviz and DOT file format; for example in Figure 14, the visualized model is the same as in Figure 8, which is viewed in CHISIO. This model is first transformed to DOT model and then transformed to DOT file for visualizing in Graphviz application.

The other example is for computer network model that is mentioned before. In Figure 15, the visualized model is the same network model whose CHISIO visualization can be seen in Figure 9. This model is also transformed to DOT model first, and transformed to DOT file for visualizing in Graphviz application.

```
helper context DOT!Node def: toString2() : String =
    'node "' + self.name +  '"[shape=' + self.shape.name +
    ' color="' + self.color +
    '" style ="' + self.style +
    if self.label.oclIsUndefined() then
        ''
    else
        '" label=' + self.label.toString2() + '];\r\n'
    endif + ';\r\n';
```

Figure 13 – Sample Helper for ATL Transformation

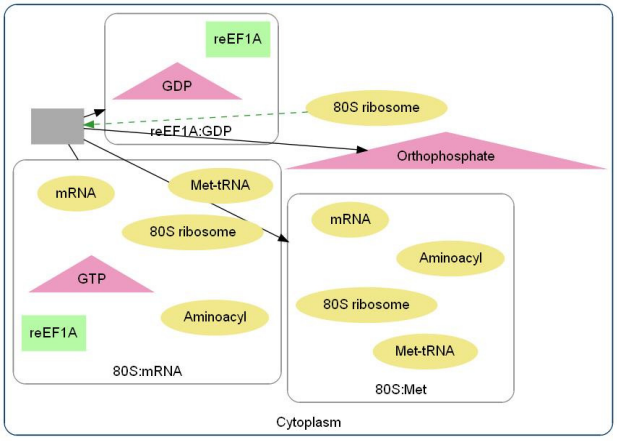

Figure 14 – Example Pathway model in Graphviz.

8. Discussion and Lesson Learned

While creating a metamodel for graph visualization, we gained experience about the benefits of model-driven software engineering. At the beginning of the project, we had some difficulties in determining the metamodel, since the discrimination between model and metamodel levels was not that apparent. This was due to the reason; MDSD concepts were new for us, whereas we had quite enough experience with OO-Software. Although most of the concepts in the domain are derived from M1 level during domain analysis process, with the help of reflection the concepts in meta-level are derived.

In order to construct a metamodel, one has to have enough and sufficient domain knowledge about the domain in consideration. In fact the situation is more or less the same even in modeling in M1 level. However, from our experiences, we claim better and broader domain

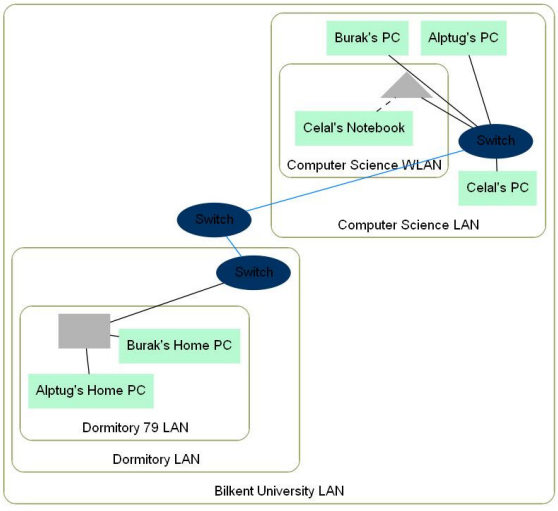

Figure 15 – Example Network visualized in Graphviz

knowledge is necessary in order to perform modeling in M2 level. The reason is due the fact that metamodel to be constructed must be expressive and comprehensive enough for the models to be constructed in M1. One can create a model of a domain in M1 level with respect to his own needs, however modeling in M2 must provide kind of a template for others to use. For instance, constructing a metamodel to represent the relations of different entities in chemistry without any knowledge about the domain will be a failure.

From our experiences in constructing the graph metamodel with graph visualization capability, a lesson we learned is the following: The effort spent in the discrimination phase between M1 and M2 levels is crucial for the validity and correctness of the metamodel to be constructed, thus enough time must be spent with caution at this phase.

We had the chance to compare the grammar and metamodel expressiveness power during the project. Although mapping of domain concepts to the grammar can be seen in Section 3, the validity or the correctness of the grammar is not tested with sample models. It is not only hard to perform the mapping but also hard to test the validity of the grammar. The grammar is the harder way for software designers to identify the relationships between the domain concepts when compared to the metamodeling explained in the section. It is without doubt that the expressiveness of the grammar is far more less than that of the metamodel.

It was surprising to see that designing the metamodel from scratch was easier than applying UML-profiling. Although UML profiling sounds much easier, we experienced difficulties in applying profiling to our domain. This is due to the reasons; the UML profiling examples provided are just simple cases where no comprehensive real world problems are expressed and the software tool used for profiling was still an incubating product.

Other than the conceptual model driven experiences described in the preceding paragraphs, we gained experience with many different tools used in different phases of the project. For the construction of the metamodel, Eclipse's EMF was used. Different plug-ins are used along with the core EMF structure, such as TOPCASED plug-in to draw the metamodel elements visually. The current version of EMF is immature. There are lots of problems encountered related with the tool, there were many exceptions thrown while interacting with menus provided, the profiling tool was incapable of drawing the associations properly, there were inconsistencies between the model and the diagrams, etc.

For the model-to-text transformation "open-ArchirectureWare" is preferred. The tutorials of the tool are useful, although full coverage is not provided. One has to search through the forums to find complicated expressions. The syntactical differences existing in different files used for this part increases the learning curve.

For the model-to-model transformation "ATL" is preferred. The syntax for ATL is a mixture of Java and OCL, which makes it easy to use and learn. However, the exception handling of the ATL is really poor, that is if there is a problem either with the input model or the actual transformation, sometimes no exception is shown. Sometimes an exception without clear statement of the actual problem is displayed, which is not that helpful.

In overall, most of the tools developed for MDSD are still incubating (they are still under development), since MDSD is relatively a new concept. As the experience and interest about the field increases, the tools and libraries will surely be more stable and powerful.

9. Conclusion and Future Work

In this paper, the interoperability issues between different graph visualization tools are stated as the main focus. In order to tackle these problems, a study composing of two main steps is performed.

As a first phase of this study, we constructed a graph metamodel which has capability of graph visualization. This construction is done in two different ways: from scratch and by using UML profiling. We give abstract syntax, concrete syntax and static semantics of the constructed metamodel. Also we generated two different M1 level models from our constructed metamodel: a biological pathway graph instances and a computer network graph instances.

We performed Model-to-Text and Model-to-Model transformations in order to conclude this study. For M2T transformation our study is focused on two different transformations. The first one is transforming our model to GraphML file format. By doing so we provided a way of visualizing our models via CHISIO. As a second M2T transformation we transform DOT model to .dot file format to visualize our models in Graphviz, this transformation is done as a part of M2M transformation.

In M2M transformation, we transform our model (based on our metamodel) to DOT model which is also a graph visualization model used in Graphviz. This transformation is done by using ATL.

To sum up we applied MDSD steps successfully and provided a way to achieve interoperability between CHISIO and Graphviz visualization tools.

There are many possible extensions as future work of this study. The first one is to perform reverse engineering and provide support for transformations between GraphML and DOT file formats. This transformation can be done via using our metamodel. If we can transform

GraphML-to-Graph metamodel then by following our current study we can transform Graph metamodel to DOT and .dot file consequently. On the opposite direction if we can transform .dot file-to-DOT metamodel we can transform DOT to our Graph metamodel and GraphML file consequently. This would be a complete bridge between DOT and GraphML formats.

Our graph metamodel has graph visualization capability. Therefore, it is possible to generate CHISIO's model package by using our metamodel as a future work. If so, application domain specific graph visualization tools such as computer network monitoring tools can be customized from CHISIO without much effort.

10. Acknowledgements

We would like to thank Asst. Prof. Bedir Tekinerdoğan for organizing the TMODELS symposium and giving us the opportunity to publish our work in this paper.

11. References

[1] http://www.cs.bilkent.edu.tr/~ivis/chisio.html. CHISIO website.

[2] http://www.graphviz.org/doc/info/lang.html. DOT file specification.

[3] http://graphml.graphdrawing.org/. GraphML file specification.

[4] Benguria, G.; Larrucea, X., "Data Model Transformation for Supporting Interoperability," *Commercial-off-the-Shelf (COTS)-Based Software Systems, 2007. ICCBSS '07. Sixth International IEEE Conference on* , vol., no., pp.172-181, Feb. 26 2007-March 2 2007

[5] http://en.wikipedia.org/wiki/Graph_(data_structure)

[6] http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf EBNF specification

[7] http://www.topcased.org/. Topcased offical website.

[8] http://www.eclipse.org/ Eclipse offical website.

[9] OCL1.5 Specification: http://www.omg.org/cgi-bin/apps/doc?formal/03-03-13.

[10] OCL 2.0 Specification: http://www.omg.org/cgi-bin/apps/doc?ptc/2003-10-14.

[11] Unified Modeling Language (UML), Version 1.5, OMG document formal/03-03-01, Object Management Group (2003), http://www.omg.org/cgi-bin/doc?formal/03-03-01.

[12] http://www.eclipse.org/modeling/emf/

[13] Kovse J. *'Generic Model-to-Model Transformations in MDA: Why and How?'* Workshop Generative Techniques in the Context od Model Driven Architecture, OOPSLA 2002.

[14] http://www.eclipse.org/m2m/atl/ Atlas model transformation language.

[15] UML 2.0 Superstructure Specification, OMG document formal/05-07-04, Object Management Group, Inc. (2005), http://www.omg.org/cgi-bin/doc?formal/05-07-04.