XGraphML - A TOOL FOR TRANSFORMING UML TO GRAPHML

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Sandeep Raavi

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

April 2010

Fargo, North Dakota

North Dakota State University
Graduate School

Title

XGRAPHML – A TOOL FOR TRANSFORMING

UML TO GRAPHML

By

SANDEEP RAAVI

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

North Dakota State University Libraries Addendum

To protect the privacy of individuals associated with the document, signatures have been removed from the digital version of this document.

ABSTRACT

Raavi,Sandeep, M.S., Department of Computer Science, College of Science and Mathematics, North Dakota State University, April 2010. XGraphML-A Tool for Transforming UML to GraphML. Major Professor: Dr. Jun Kong.

This paper proposes and implements XGraphML, a tool which transforms a file from a UML format to a GraphML using XSLT and JAXP technologies. The tool connects ArgoUML with VEGGIE through this transformation. ArgoUML is an open-source tool used to model software artifacts through UML diagrams and uses UML as a standard saving mechanism. On the other hand, VEGGIE (i.e., the *Visual Environment for Graph Grammar Induction and Engineering*) utilizes GraphML as a standard saving mechanism. XGraphML takes a UML file as input and produces a corresponding GraphML file. Based on XGraphML, software engineers use ArgoUML to model software artifacts and then pass the designed model to VEGGIE for further analysis.

A stylesheet is used to transform UML format to a GraphML. The stylesheet is processed using the XGraphML tool that has been introduced through this paper. XGraphML provides a way to transform one major graph format to another. This tool is built on the base of Java, which is one of the most common programming languages and enhances extendibility for further use. Furthermore, a graphical user interface has been implemented which enables the users to view the transformation on the screen and then save it to the desired location on the system.

ACKNOWLEDGEMENTS

I thank Dr. Jun Kong for being my advisor, providing exceptional guidance and giving insight throughout the research, making this paper possible. I would like to thank Dr. Kendall Nygard, Dr. Gursimran Walia and Dr. Jin Li for serving on my committee.

I would like to thank my family, all my friends and everyone who encouraged me for their continuous support while completing this paper.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS iv

LIST OF FIGURES .... v iii

CHAPTER 1.INTRODUCTION 1

1.1. Background 3

1.1.1. XML 3

1.1.2. Types of XML based graph formats 4

1.1.3. VEGGIE- A tool for graph grammar 5

1.1.4. ArgoUML and UML diagrams 5

1.1.5. Class Diagrams 6

1.2. Approach 7

CHAPTER 2. RELATED WORK 9

2.1. XMI to HTML 9

2.2. Pamda tool 9

2.3. XMI2GXL Translator 10

2.4. JAXP 1.4 (Java API for XML parsing) 10

2.4.1. SAX (Simple API for XML) 10

2.4.2. DOM (Document Object Model) API 11

2.4.3. API for XSLT 11

2.5. <oXygen/> xml Editor 11

2.6. Eclipse IDE 12

TABLE OF CONTENTS (Continued)

CHAPTER 3.UNDERSTANDING XMI,PGML AND GRAPHML........................ 13

3.1. Core elements of XMI...13

3.1.1. Classes and Attributes... 14

3.1.2. Associations...18

3.2. Core elements of PGML...20

3.2.1. <Group>...20

3.2.2. <rectangle>...20

3.2.3. <text>...21

3.3. Core elements of GraphML..23

3.3.1. Nodes...24

3.3.2. Edges...24

3.3.3. Port...25

3.3.4. <desc>...25

3.3.5.<data>...25

3.4. Sample GraphML...26

3.5. Generated GraphML for a simple association..............................27

3.6. Comparison of UML format and GraphML format............................ 28

CHAPTER 4. IMPLEMENTATION OF XSLT.. 30

4.1. Elements used in the transformation...33

4.2. XGraphML.xsl...34

4.2.1. Initial Version Check and Validation 37

4.2.2. Classes or Nodes...39

TABLE OF CONTENTS (Continued)

4.2.3. Methods ... 39

4.2.4. Attributes ... 40

4.3. Association or Edges ... 40

4.3.1. Source ... 41

4.3.2. Sourceport ... 41

4.3.3. Target ... 42

4.3.4. Target Port ... 42

4.3.5. Directed or Undirected Edge ... 43

CHAPTER 5. IMPLEMENTATION OF THE TOOL-XGRAPHML ... 44

5.1. Block Diagram of the transformation ... 44

5.2. Basic Interface of the tool ... 45

5.3. Implementation of Transformer class ... 48

CHAPTER 6. CASE-STUDY ... 51

6.1. Class diagram for a Shopping Cart ... 51

6.2. Case Study ... 54

CHAPTER 7. CONCLUSIONS AND FUTURE WORK ... 56

7.1. FUTURE WORK ... 57

BIBILOGRAPHY ... 59

LIST OF FIGURES

Figure **Page**

1.1. Contents of an .uml format.. 2

1.2. ArgoUML..6

1.3. Simple Class diagram with an association between them................................7

3.1. Example of a class diagram...14

3.2. XMI Code generated for Fig3.1.. 17

3.3. XMI Code generated for an association..19

3.4. PGML file generated from two classes and a relation....................................23

3.5. Sample GraphML file..26

3.6. Generated GraphML code for Fig(3.5)...27

3.7. GraphML file for a simple graph with a single association...............................28

3.8. Element level comparison...29

4.1. Pattern matching of XSLT..32

4.2. Basic elements of a stylesheet and their description.......................................34

4.3. XgraphML.xsl...37

5.1. Block diagram of the transformation...44

5.2. Screenshot of XGraphML.. 45

5.3. Opening an UML file...46

5.4. With an UML file loaded..46

5.5. Transformation in progress...47

5.6. Progress indicator...47

5.7. Usage of Transformer class..48

LIST OF FIGURES (Continued)

6.1. Shopping Cart Example..51

6.2. Shoppingcart.graphml..53

6.3. List of classes in UML format and GraphML format...................................54

6.4. List of associations in UML format and GraphML format.............................55

CHAPTER 1

INTRODUCTION

In the current programming world where everything is moving from procedure oriented programming to a declarative way of programming, XML is one such language which has enhanced declarative way of programming to a greater extent [2]. XML is a mark-up language in which all the elements are self-declarative. These custom-made elements change depending upon the needs of user. Technologies like Web Services and Hibernate use XML to configure applications which are built using modern day technologies.

The Unified Modeling language has become a de-facto modeling standard in the present world of OOAD (Object oriented Analysis and design), and UML has made the life of a developer easier and simpler with all the various diagrams [6]. A model is required to create a user's visual abstraction of an application before it is built or coded. Most of the programmers prefer to create a blueprint and to visualize the outcome before an actual application is built. The blue print can be a class diagram, a visual model or any UML diagram [6]. There is a growing need in today's IT world to represent a complex UML diagram in the form of a graph, which provides an abstract view to the user.

This paper deals with two forms of XML graph formats, UML format [1] and GraphML(Graph Markup language) [14], which are based on the XML standard. A class diagram is saved with an extension of .uml, which is an XML based format with a combination of two different subsets of XML formats (XMI, PGML).

The .uml refers to a format in which a class diagram is generated within ArgoUML, but UML refers to the Unified Modeling Language, which is a standard for modeling language. The .uml format in Fig 1.1.has the first and last element as <uml> in the XML file, which is shown in the below XML.

```
<uml>
        <xmi>
                <......>
        </xmi>
        <pgml>
                <.....>
        </pgml>
</uml>
```

Fig 1.1. Contents of an .uml format

VEGGIE (*Visual Environment for Graph Grammar Induction and Engineering*), a tool for context sensitive parsing, takes .graphml as input and applies graph grammars for further analysis. As Veggie does not accept a UML diagram directly, so the UML diagram has to be converted to GraphML in order to retrieve the graph. This generates a need to have an interface that bridges the gap between UML and VEGGIE through the transformation. A graph that is generated using the designed tool can be fed into VEGGIE to process the graph further into a meaningful format. Veggie retains the semantics of a UML diagram using graph grammars and self-declared production rules. GraphML is also an XML based language which lays ground towards an exchange format for graph visualizations [14]. This format is designed to support all the tools which support XML and are saved with an extension of .graphml and .graph.

During the transformation of UML to GraphML, the semantics of a UML diagram has to be identified in order to further break down the complex structure and generate a host graph. This was my main motivation in constructing an open source tool that does this transformation and retains the semantics of a UML diagram.

The main objective of this paper is to design and implement an open source tool that can easily convert a UML diagram from XML to a graphml. This tool helps a user to break down the complexity of a UML diagram and better understand the abstract level of information through the transformation. The derived host graph is further analyzed in VEGGIE by custom graph grammars and production rules.

1.1. Background:

1.1.1. XML

With the evolution of internet and Web technologies, there is a wide need to process the dynamic data currently served in most websites. Processing this type of dynamic data in web pages can be easily achieved by injecting XML files [21]. If a company plans to upgrade its architecture, the entire data in an older version has to be transformed into a newer version, which involves a great investment of time and resources. As one format is not compatible to another, this demands a common language, or a bridge, which is compatible across different platforms and makes the transformation easy. Most companies use XML as a medium to exchange data between two incompatible systems.

Aspects like dynamic data processing and ease of use makes XML a user friendly language across cross platform architectures.Techniologies like Web-services and XHTML enhanced the usage of XML (Extensible Mark-up Language). XML is a subset of SGML

(Standard Graph Markup language) [8] and is a mark-up language created for documents containing structured information. In this language users can define custom tags depending on their usage, and this makes XML independent and application neutral language. Every application has its own document structure, and a DTD (Document Type Definition) is used to provide grammar to a particular structure and forms a base to XML [23]. Any XML based language would have a DTD predefined according to OMG, and this can further be modified depending upon application requirements.

1.1.2. Types of XML based graph formats

This paper is focused on graph transformation and graph grammars. Therefore, we take into account some of the core XML based graph formats like:

1. PGML (Precision Graphics Markup Language)
2. XMI (XML Meta data Interchange) (Refer to Chapter-3)
3. GraphML (Graph Mark-up language) (Refer to Chapter-3)

PGML is a XML based 2D scalable graphics language designed to meet vector graphics needed by novice and expert level graphics artists [25]. PGML uses the imaging model of the PostScript language that is used by most of the graphic designers and is the basis for some of the major graphing applications like Corel Draw, Adobe Illustrator, etc. A PGML drawing consists of the collection of one or more graphical objects-path objects, shape objects and text objects [27].The XML elements generated by this drawing have a certain standard defined by W3C. A user utilizing PGML can do all sorts of enhancements, such as adding color to the drawing and setting thickness to the diagrams..

1.1.3. VEGGIE- A tool for graph grammar

VEGGIE is a tool which provides context-sensitive parsing and induction environment [11]. This tool is used to produce complex graph grammars for visual languages by using a machine learning system and injecting production rules. There has been a wide range of usage for graph grammars in the visual modeling industry, and several tools have been developed to enhance this use. Most of the software engineers prefer to view their programs and workflows visually in the form of graphs or diagrams. There are several ways to express graphs in natural language processing.

VEGGIE motivates us to work further in the direction of the transformation from one graph format to another to further enhance interoperability between two graph formats. Graph grammar is generally achieved by specifying certain production rules and is represented as a graph in the form of nodes and edges. As most graph transformation tools consider GraphML format to be flexible and easy to generate, we consider going further in that direction with our research and proposing a tool that transforms UML format to GraphML. GraphML also employs almost similar kinds of formatting, such as nodes, edges and endpoints.

1.1.4. ArgoUML and UML diagrams

ArgoUML is an open source UML modeling tool which is built on the platform of Java, making it even more popular and compatible on multiple platforms. This tool shown in Fig 1.2. can be used to draw UML 1.4 diagrams like Class, State, Use case, Activity, and Collaboration diagrams, among others[8] .ArgoUML uses re-engineering to generate code [3] for all the diagrams mentioned above. We can also use the same tool to retrieve a model

from code, generated with the help of reverse engineering. Keeping all the features and scalability in mind, working with this tool is ideal and an efficient match.

There is a broad classification of UML diagrams in the current version of UML 2.0, which constitutes 13 different types of diagrams. In this paper we consider class diagrams, as these diagrams are the most efficient way of expressing classes and objects in object oriented programming. Programmers represent the entire application into classes, attributes and methods.

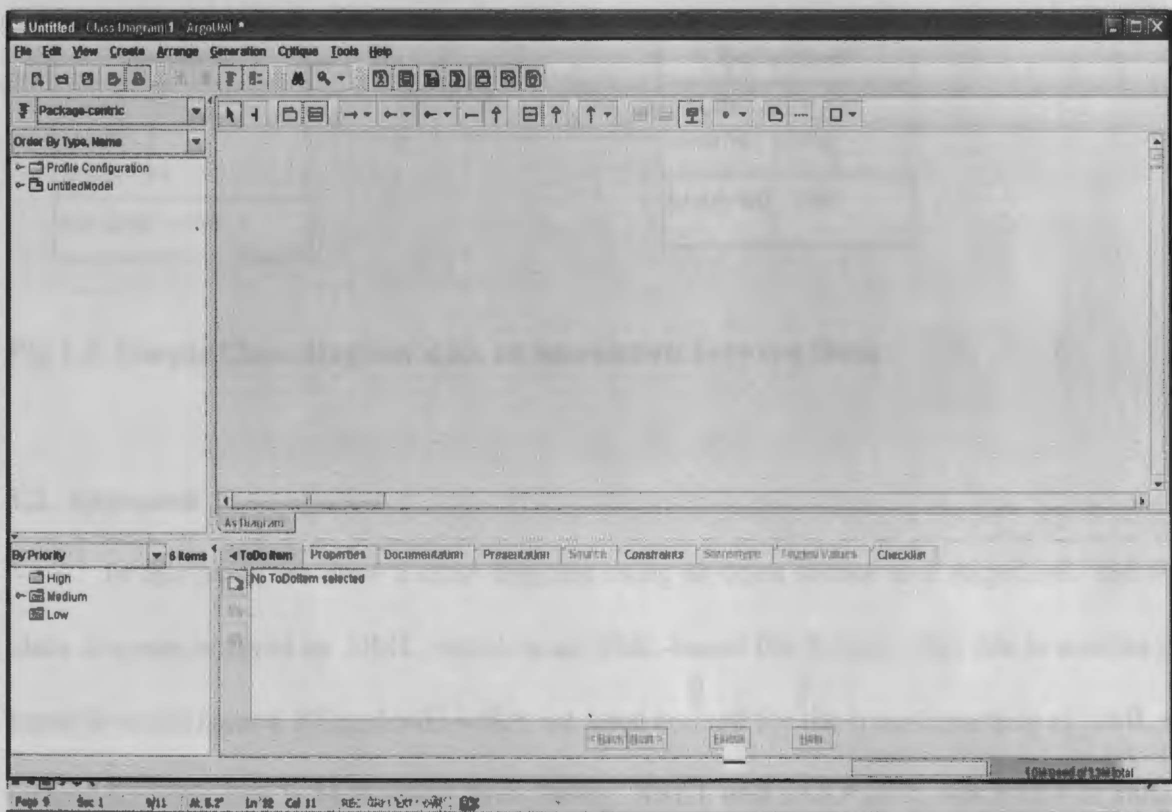

Fig 1.2. ArgoUML

1.1.5. Class Diagrams

Class diagrams are a visual representation of a detailed system design. These diagrams consist of classes, attributes, methods and relationships between classes. A class

is drawn as a rectangle with 3 partitions. The first partition has the name of the Class, the second partition consists of its attributes, and the third has all the methods. Due to a wide use of OOAD, class diagrams have become popular and are a de-facto standard for representing a visual static structure.

In Fig 1.3, Employee and Department are two simple classes, and the relationship between them is named "Works for". This class diagram generated form ArgoUML is saved in XML format with an extension of .uml.

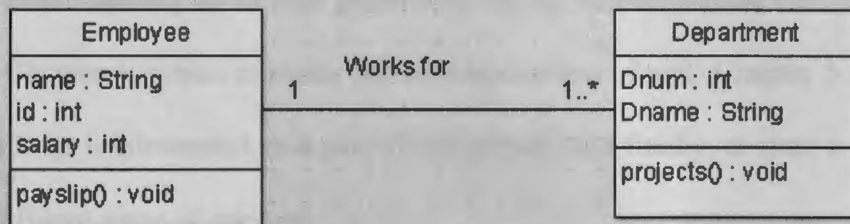

Fig 1.3. Simple Class diagram with an association between them

1.2. Approach

In this paper we draw a class diagram using an open source tool ArgoUML and the class diagram is saved as .UML, which is an XML-based file format. This file is used as an input to a tool named XGraphML which we have created for the transformation of UML to GraphML. This tool is built using Java Swing, XSLT and JAXP [17]. An XSLT is coded with the help of XPath [18], where in the source document, i.e., UML, is transformed into resultant graph format GraphML. We write certain rules in XPath to keep the source document the same but apply these rules to produce a resultant target document.

We propose and design the tool XGraphML, which consists of a user interface where users can input .UML files, selecting a corresponding XSLT required for

transformation and then get a resulting GraphML file on the other half of the display pane. The user has an option to edit the document and save it to a desired location on the system. This file is later used for further analysis as input to VEGGIE by using graph grammars and production rules. The XSLT can be customized based on the users' needs and achieve a desired output using XGraphML.

The rest of the paper is organized into 5 chapters. In chapter 2, we discuss some of the related work to this paper and work done previously in this direction. Chapter 3 offers an understanding of various graph formats for transformation used in this paper, followed by Chapter 4, which explains the implementation of xslt. Chapter 5 describes the tool that has been implemented as a part of this paper. And finally, chapter 6 states the conclusions and future areas of research.

CHAPTER 2

RELATED WORK

This chapter gives a brief introduction of some of the available tools in the market that take XML as input and then generate various outputs. Most of the tools use sophisticated technologies to achieve these transformations. We analyze tools which generate code from XMI using re-engineering and then try to implement the same idea of transformation while using pattern matching techniques.

2.1. XMI to HTML

In order to transform an XMI document to a browser enabled format like HTML, a coded stylesheet can be used to perform the task of transformation. The main purpose of this project is to display an object oriented design in a browser [23]. This project uses an XSLT processor to process the stylesheet along with XMI and generates a HTML file which can be viewed in a browser.

2.2. Pamda tool

This tool is an UML2code generator which supports AOM (Aspect oriented modeling). This tool takes input of any UML diagram, which supports Aspect oriented modeling in XMI format and generates corresponding code for it [10]. The tool also makes use of a sophisticated velocity template engine in order to perform this conversion.

2.3. XMI2GXL Translator

A stylesheet is written using XSLT where GXL is produced from XMI. A class diagram is saved as .xmi from ArgoUML, and then a corresponding XSLT is used to transform this XMI to GXL [9]. This tool is command based, which requires a file to be executed from a command prompt and does not have any user interface to do the processing.

As described in [13], xslt can be used to transform GXL to GraphML and vice versa. This paper uses xslt extensively to complete the task of transformation. In this paper, this methodology is used as a base to use xslt, but with a different set of graph formats.

2.4. JAXP (Java API for XML parsing)

JAXP uses its API (Application programming interface) to parse and validate XML documents [17]. The initial version of JAXP 1.3 was released with J2SE 5.0, but with the release of J2SE 6.0 JAXP 1.4, there has been a support of Streaming API for XML (StaX). JAXP consists of the following APIs

1. SAX (Simple API for XML)
2. DOM (Document Object Model)
3. Transformer (Refer to Chapter 5)

2.4.1. SAX (Simple API for XML)

SAX [17] is used for element by element processing in an XML document. In this method of parsing, a SAXParserFactory class helps us to generate an instance of a parser.

There are several interfaces already pre-defined in the API, such as ContentHandler, ErrorHandler, EntityResolver, and DTDHandler, which consist of some pre-defined methods.. These methods are used by SAXReader object, which are wrapped in parser. This kind of parsing is done on an element-by-element basis, which makes it useful on the server side, rather than the application front.

2.4.2. DOM (Document Object Model) API:

DOM parsing is basically done in a tree-structure format. The entire XML document is loaded into memory in the form of a tree structure, classified into nodes and parsing starts from root node to preceding children. This API is generally more inclined towards interactive applications as the entire XML tree structure is present in the memory, aiding the user to modify and access results. In this API a DocumentBuilderFactory class is used to create a DocumentBuilder instance, which is used to parse an XML document.

2.4.3. API for XSLT

JAXP has an API for XSLT processing which is defined in the javax.xml.transform package, which provides access to the XSLT created by this study, which reads as a stream of data and is processed with the transformation instructions. This API has been used in tool creation and to process the stylesheet, and more information on the API is provided in Chapter 4.

2.5. <oXygen/> xml Editor

In this paper we have used <oXygen> to validate and test stylesheet, using multiple

sources and target documents. This tool offers us with a platform of transformation by providing some core functionalities to experiment with xml, such as editing, parsing, transformation and to support the modern day needs of programming. This tool also provides the functionality of SOAP and WSDL testing [19]. We used this tool in debugging and testing the stylesheet that has been used for transformation in this paper. This tool has been used for the initial testing of stylesheet as this has almost all the xslt processors embedded in it. <oXygen/> tool is a commercial software and requires a purchased license; we used a trial version, which lasts for 30 days and is fully functional as a paid software, but only for a limited period of time.

2.6. Eclipse IDE

We used Eclipse, an open source IDE, to build XGraphML with the help of four different classes. The entire project is run as a package which can further be deployed on to jboss server to make it accessible for multiple users. This open source development environment provides various plug-ins by importing their respective libraries [7]. Eclipse incorporates a built-in incremental compiler and full model of java source files. Eclipse can be used to export the entire project into an executable jar file that makes XGraphML portable and platform independent, provided java is installed as a pre-requisite on the machine.

CHAPTER 3

UNDERSTANDING XMI, PGML AND GRAPHML

In order to transform one format to another there has to be a clear understanding between all the elements of source and the target document. The source elements are to be mapped to the target elements with the help of processing instructions written in XSLT. As this paper deals with the transformation of UML format to GraphML, we have to understand what elements in UML format correspond to elements in GraphML and then try to map most all of them. All the elements cannot be mapped with each other as they are in two different formats, so we try to extract all possible elements from the source document and then try to inject them in the target document, i.e., GraphML.

3.1. Core elements of XMI

XMI is an OMG standard that deals with the exchange of metadata via XML [16]. In this paper we use an open source tool to draw class diagrams which are saved in .uml format. All the data that is present in the class diagrams are saved in xmi and pgml elements that look similar to XML format. XMI has a specific DTD, which was defined by the OMG [4] when this format was proposed, but ArgoUML uses its customized tages. We give a brief overview of the XMI format and the elements used from the perspective of ArgoUML and how the data is stored in this tool.

3.1.1. Classes and Attributes

Fig 3.1 shows a simple class diagram with two classes Employee and Department. The attributes of the classes are name, id, salary, Dnum, Dname and has two methods- payslip() and projects().

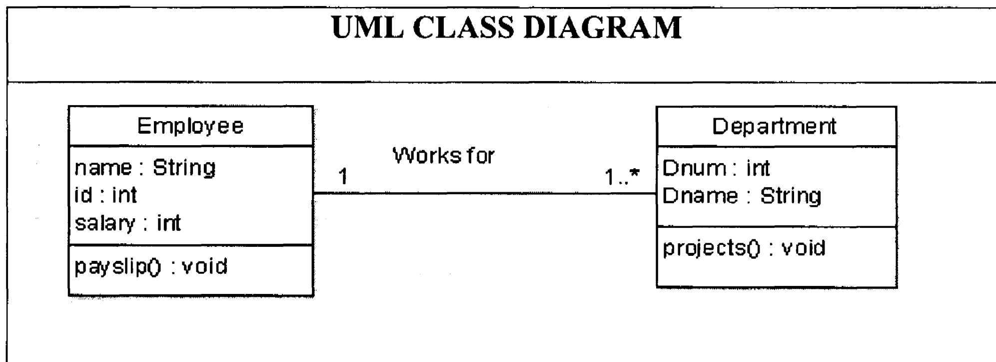

Fig 3.1. Example of a class diagram

XMI Code

```
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Thu Feb 25
13:02:45 CST 2010'>
  <XMI.header>   <XMI.documentation>
    <XMI.exporter>ArgoUML (using Netbeans XMI Writer version 1.0)</XMI.exporter>
    <XMI.exporterVersion>0.28.1(6) revised on $Date: 2007-05-12 08:08:08 +0200 (Sat, 12 May
2007) $ </XMI.exporterVersion>
   </XMI.documentation>
   <XMI.metamodel xmi.name="UML" xmi.version="1.4"/></XMI.header>
  <XMI.content>
   <UML:Model xmi.id = '10-22-2--73-324c2761:127066bd6dc:-8000:0000000000000DDB'name
= 'untitledModel' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
     <UML:Namespace.ownedElement>
      <UML:Class xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DDC'name = 'Employee' visibility = 'public' isSpecification = 'false' isRoot
= 'false' isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
        <UML:Classifier.feature>
         <UML:Attribute xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DDD'name = 'name' visibility = 'public' isSpecification = 'false' ownerScope
= 'instance'changeability = 'changeable' targetScope = 'instance'>
  <UML:StructuralFeature.multiplicity>
 <UML:Multiplicity xmi.id = '10-22-2--73-324c2761:127066bd6dc:-8000:0000000000000DDE'>
        <UML:Multiplicity.range>
     <UML:MultiplicityRange xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DDF'lower = '1' upper = '1'/>
```

```
          </UML:Multiplicity.range>
        </UML:Multiplicity>
      </UML:StructuralFeature.multiplicity>
      <UML:StructuralFeature.type>
        <UML:DataType href = 'http://argouml.org/profiles/uml14/default-uml14.xmi#-84-17--
56-5-43645a83:11466542d86:-8000:000000000000087E'/>
      </UML:StructuralFeature.type>
    </UML:Attribute>
    <UML:Attribute xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DE0' name = 'id' visibility = 'public' isSpecification = 'false' ownerScope =
'instance'changeability = 'changeable' targetScope = 'instance'>
      <UML:StructuralFeature.multiplicity>
        <UML:Multiplicity xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DE1'>
          <UML:Multiplicity.range>
            <UML:MultiplicityRange xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DE2'lower = '1' upper = '1'/>
          </UML:Multiplicity.range>
        </UML:Multiplicity>
      </UML:StructuralFeature.multiplicity>
      <UML:StructuralFeature.type>
        <UML:DataType href = 'http://argouml.org/profiles/uml14/default-
java.xmi#.:000000000000086C'/>
      </UML:StructuralFeature.type>
    </UML:Attribute>
    <UML:Attribute xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DE3'name = 'salary' visibility = 'public' isSpecification = 'false' ownerScope
= 'instance'changeability = 'changeable' targetScope = 'instance'>
      <UML:StructuralFeature.multiplicity>
        <UML:Multiplicity xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DE4'>
          <UML:Multiplicity.range>
            <UML:MultiplicityRange xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DE5'lower = '1' upper = '1'/>
          </UML:Multiplicity.range>
        </UML:Multiplicity>
      </UML:StructuralFeature.multiplicity>
      <UML:StructuralFeature.type>
        <UML:DataType href = 'http://argouml.org/profiles/uml14/default-uml14.xmi#-84-17--
56-5-43645a83:11466542d86:-8000:000000000000087E'/>
      </UML:StructuralFeature.type>
    </UML:Attribute>
    <UML:Operation xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DE6'name = 'payslip' visibility = 'public' isSpecification = 'false'
ownerScope = 'instance'isQuery = 'false' concurrency = 'sequential' isRoot = 'false' isLeaf = 'false'
isAbstract = 'false'>
      <UML:BehavioralFeature.parameter>
        <UML:Parameter xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DE7' name = 'return' isSpecification = 'false' kind = 'return'>
          <UML:Parameter.type>
```

```
              <UML:DataType href = 'http://argouml.org/profiles/uml14/default-
java.xmi#.:000000000000086B'/>
            </UML:Parameter.type>
          </UML:Parameter>
        </UML:BehavioralFeature.parameter>
      </UML:Operation>
    </UML:Classifier.feature>
  </UML:Class>
  <UML:Class xmi.id = '10-22-2--73-324c2761:127066bd6dc:-8000:0000000000000DE8'
    name = 'Department' visibility = 'public' isSpecification = 'false' isRoot = 'false'isLeaf =
'false' isAbstract = 'false' isActive = 'false'>
    <UML:Classifier.feature>
      <UML:Attribute xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DE9' name = 'Dnum' visibility = 'public' isSpecification = 'false' ownerScope
= 'instance'changeability = 'changeable' targetScope = 'instance'>
        <UML:StructuralFeature.multiplicity>
          <UML:Multiplicity xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DEA'>
            <UML:Multiplicity.range>
              <UML:MultiplicityRange xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DEB'lower = '1' upper = '1'/>
            </UML:Multiplicity.range>
          </UML:Multiplicity>
        </UML:StructuralFeature.multiplicity>
        <UML:StructuralFeature.type>
          <UML:DataType href = 'http://argouml.org/profiles/uml14/default-
java.xmi#.:000000000000086C'/>
        </UML:StructuralFeature.type>
      </UML:Attribute>
      <UML:Attribute xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DEC'name = 'Dname' visibility = 'public' isSpecification = 'false'
ownerScope = 'instance'changeability = 'changeable' targetScope = 'instance'>
        <UML:StructuralFeature.multiplicity>
          <UML:Multiplicity xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DED'>
            <UML:Multiplicity.range>
              <UML:MultiplicityRange xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DEE'lower = '1' upper = '1'/>
            </UML:Multiplicity.range>
          </UML:Multiplicity>
        </UML:StructuralFeature.multiplicity>
        <UML:StructuralFeature.type>
          <UML:DataType href = 'http://argouml.org/profiles/uml14/default-uml14.xmi#-84-17--
56-5-43645a83:11466542d86:-8000:000000000000087E'/>
        </UML:StructuralFeature.type>
      </UML:Attribute>
      <UML:Operation xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DEF'name = 'projects' visibility = 'public' isSpecification = 'false'
ownerScope = 'instance' isQuery = 'false' concurrency = 'sequential' isRoot = 'false' isLeaf =
'false'isAbstract = 'false'>
```

```
          <UML:BehavioralFeature.parameter>
           <UML:Parameter xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DF0'name = 'return' isSpecification = 'false' kind = 'return'>
             <UML:Parameter.type>
</UML:Parameter.type>
            </UML:Parameter>
          </UML:BehavioralFeature.parameter>
        </UML:Operation>
      </UML:Classifier.feature>
    </UML:Class>
```

Fig 3.2. XMI code generated for Fig3.1.

ArgoUML generates XMI format with numerous elements and attributes, but for our requirement we only considered core elements that are required for transformation. Every XMI code generated from the ArgoUML follows a specific DTD [2] (Document Type Definition).This format is divided into header **(<XMI.header>)** and content **(<XMI.content>)**. **(<XMI.header>)** consists of the exporter, i.e. ArgoUML in our case, and its version in **(<XMI.exporterVersion>)**. It also has the name of the metamodel and its version, from which this class diagram has been derived.

As metamodel is UML, we import UML namespace initially in the starting of the document given as "xmlns:UML = 'org.omg.xmi.namespace" .The name of the class with other properties of that class are stored in **(<UML:Class>)** and each class has an auto-generated xmi.id. In the example above, xmi.id is '10-22-2--73-324c2761:127066bd6dc:-8000:0000000000000DDC,which is a unique id. This id is used as a reference further in the entire XMI document to make associations. The class name is bold and marked in red to differentiate readable to the reader. **(<UML:Operation>)** consists of all the methods that have been specified in a class. **(<UML:Attribute>)** contains all the attributes of a particular class. **(<UML:MultiplicityRange>)** element has multiplicity like 0..*(zero-to-many), 1..*(one-to-many), 1..1(one-to-one) and are distinguished as upper limit and lower

limit and a value next to it. All the above mentioned elements are child elements to **(<UML:Class>)**, and every class has all these elements in common. If a class has no attributes, the name attribute is left empty, but a unique id is auto-generated and is referenced with the same id in the entire XML.

3.1.2. Associations

Taking Fig 3.1 into consideration, if there is an association between Employee and Department, the code in the Fig 3.3 is added up in the above XMI code. **(<UML:Association>)** element contains all the properties of the association. The name of the association is given in the name attribute inside the **(<UML:Association>)**, the origin and the target classes are referenced with their id in **(<UML:AssociationEnd.participant>)**. The diagram consists of only one association and thus there are two end participants which are referenced to both the classes, with their xmi.ids'**10-22-2--73-324c2761:127066bd6dc:-8000:0000000000000DDC**' for Employee and **'10-22-2--73-324c2761:127066bd6dc:-8000:0000000000000DE8'** for Department. The multiplicity of the association is given by **(<UML:MultiplicityRange>)**. The name of the association "Works for" is also given by a randomly auto-generated id "**10-22-2--73-324c2761:127066bd6dc:-8000:0000000000000DF1'**, which is highlighted in red for the reader's convenience.

XMI Code
<UML:Association xmi.id = **'10-22-2--73-324c2761:127066bd6dc:-8000:0000000000000DF1'** name = 'Works for' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'> <UML:Association.connection> <UML:AssociationEnd xmi.id = '10-22-2--73-324c2761:127066bd6dc:-

```
8000:0000000000000DF2'visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering
= 'unordered'aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
  <UML:AssociationEnd.multiplicity>
    <UML:Multiplicity xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DF9'>
      <UML:Multiplicity.range>
        <UML:MultiplicityRange xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DF8'lower = '1' upper = '1'/>
        </UML:Multiplicity.range>
      </UML:Multiplicity>
     </UML:AssociationEnd.multiplicity>
    <UML:AssociationEnd.participant>
     <UML:Class xmi.idref = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DDC'/>
     </UML:AssociationEnd.participant>
    </UML:AssociationEnd>
    <UML:AssociationEnd xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DF5'visibility = 'public' isSpecification = 'false' isNavigable = 'true' ordering
= 'unordered'aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
     <UML:AssociationEnd.multiplicity>
      <UML:Multiplicity xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DFB'>
       <UML:Multiplicity.range>
        <UML:MultiplicityRange xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DFA'lower = '1' upper = '-1'/>
       </UML:Multiplicity.range>
      </UML:Multiplicity>
     </UML:AssociationEnd.multiplicity>
     <UML:AssociationEnd.participant>
      <UML:Class xmi.idref = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DE8'/>
     </UML:AssociationEnd.participant>
    </UML:AssociationEnd>
   </UML:Association.connection>
  </UML:Association>
 </UML:Namespace.ownedElement>
</UML:Model>
<UML:Multiplicity xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DF3'>
  <UML:Multiplicity.range>
   <UML:MultiplicityRange xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DF4'lower = '1' upper = '1'/>
  </UML:Multiplicity.range>
</UML:Multiplicity>
<UML:Multiplicity xmi.id = '10-22-2--73-324c2761:127066bd6dc:-
8000:0000000000000DF6'>
```

Fig 3.3. XMI code generated for an association

3.2. Core elements of PGML

Precision graph markup language is a language which is used to generate 2D graphics for some pre-defined shapes like rectangle, circle, eclipse and pie wedge. ArgoUML uses PGML to define co-ordinate systems for each rectangle represented as a class in a class diagram. This graph format has some elements which represent advanced graphical properties like color, font and stroke.

3.2.1. <Group>

The **<Group>**element of the PGML file has attribute names, descriptions, and has a reference given by the attribute 'href' to each class of a class diagram, and attributes such as fill, fillcolor, stroke and stroke color are used to enhance the appearance of a class as a whole. The name attribute is given a specific reference number like "Fig 0.0", "Fig 0.1.0" etc. The XML code for the element <group> is given by

<group name="Fig0" description="org.argouml.uml.diagram.static_structure.ui.FigClass [168, 72, 112, 112] pathVisible=false; stereotypeView=0; operationsVisible=true; attributesVisible=true; "href="**10-22-2--73-324c2761:127066bd6dc:-8000:0000000000000DDC**" fill="1" fillcolor="white" stroke="1" strokecolor="black">

3.2.2. <rectangle>

In a class diagram, a class is represented as a rectangle with three partitions, and each partition is given a specific coordinates in terms of 'x' and 'y'. Attributes like height and width are used to specify the dimensions of that rectangle, and attributes like color are available to make a class look user friendly and interactive. We make use of the xy coordinates in the resultant graphml. Each class in a class diagram has 3 <rectangle> elements incorporated in to a <group> element.

```
<group>
	<rectangle name="Fig0.0" x="168" y="72" width="112" height="112" fill="1"
	fillcolor="white" stroke="0" strokecolor="0 255 255"/>
</group>
```

3.2.3. <text>

This element of PGML holds all the text entered into the class, such as the name, methods and the attributes. Attributes like font, font size and textcolor are added in addition to the previous attributes. Fig 3.4. shows the generated code in PGML.

```
<text name="Fig0.3.2" x="168" y="154" width="84" height="22" fill="0" fillcolor="
white" stroke="0" strokecolor="black" textcolor="black" font="Dialog" italic="false"
bold="false" textsize="12" justification="Left">payslip() : void</text>
```

Generated PGML

```
<pgml description="org.argouml.uml.diagram.static_structure.ui.UMLClassDiagram|10-22-2--
73-324c2761:127066bd6dc:-8000:0000000000000DDB"name="Class Diagram>
 <group
name="Fig0"description="org.argouml.uml.diagram.static_structure.ui.FigClass[16
8, 72, 112,
112]pathVisible=false;stereotypeView=0;operationsVisible=true;attributesVisible=t
rue;"href="10-22-2--73-324c2761:127066bd6dc:-8000:0000000000000DDC"
fill="1" fillcolor="white" stroke="1"  strokecolor="black">
   <private>
   </private>

   <rectangle name="Fig0.0" x="168" y="72" width="112"
height="112"fill="1"fillcolor="white"stroke="0"
    strokecolor="0 255 255"/>
   <group name="Fig0.1" description="org.argouml.uml.diagram.ui.FigStereotypesGroup[168,
72, 112, 0]"
      href="10-22-2--73-324c2761:127066bd6dc:-8000:0000000000000DDC" fill="1"
fillcolor="white" stroke="1"      strokecolor="black">
    <private>
    </private>

    <rectangle name="Fig0.1.0" x="168"  y="72" width="112"  height="0"
fill="1"fillcolor="white"stroke="1"      strokecolor="black" />
   </group>
   <text name="Fig0.2"  x="168"  y="72" width="112" height="23" fill="1" fillcolor="white"
stroke="0"   strokecolor="black" textcolor="black" font="Dialog"
italic="false"bold="false"textsize="12"justification="Center"
```

```
   >Employee</text>
   <group name="Fig0.3"
description="org.argouml.uml.diagram.ui.FigOperationsCompartment[168, 153, 112, 30]"
      href="10-22-2--73-324c2761:127066bd6dc:-8000:0000000000000DDC" fill="1"
fillcolor="white" stroke="0"
      strokecolor="black">
    <private>
    </private>
    <rectangle name="Fig0.3.0"  x="168"  y="153"  width="112"  height="30" fill="1"
fillcolor="white" stroke="0"strokecolor="black" />
    <path name="Fig0.3.1"
description="org.argouml.uml.diagram.ui.FigEditableCompartment$FigSeperator"
     fill="1" fillcolor="white"stroke="1"strokecolor="black" >
     <moveto x="168" y="153" />
    <lineto x="279" y="153" />
    </path>
    <text name="Fig0.3.2" x="168" y="154" width="84"
height="22"fill="0"fillcolor="white"stroke="0"          strokecolor="black" textcolor="black"
font="Dialog" italic="false"bold="false"textsize="12"
justification="Left">payslip() : void</text>
   </group>
   <group name="Fig0.4"
description="org.argouml.uml.diagram.ui.FigAttributesCompartment[168, 95, 112, 58]"
      href="10-22-2--73-324c2761:127066bd6dc:-8000:0000000000000DDC"
fill="1"fillcolor="white"stroke="0" strokecolor="black">
    <private>
    </private>
     <rectangle name="Fig0.4.0" x="168" y="95" width="112"
height="58"fill="1"fillcolor="white"stroke="0"
     strokecolor="black"/>
    <path name="Fig0.4.1"
description="org.argouml.uml.diagram.ui.FigEditableCompartment$FigSeperator"
     fill="1"fillcolor="white" stroke="1" strokecolor="black">
     <moveto x="168" y="95" />
     <lineto x="279"y="95" />
    </path>
    <text name="Fig0.4.2" x="168" y="96"width="79"
height="16"fill="0"fillcolor="white"stroke="0"
strokecolor="black"textcolor="black"font="Dialog" italic="false" bold="false"
textsize="12"justification="Left"
    >name : String</text>
<text name="Fig0.4.3" x="168"  y="112"  width="38"  height="16"
fill="0"fillcolor="white"stroke="0"   strokecolor="black" textcolor="black"font="Dialog"
italic="false"bold="false" textsize="12" justification="Left">id : int</text>
    <text name="Fig0.4.4"  x="168"  y="128" width="80" height="22"
fill="0"fillcolor="white"stroke="0"
strokecolor="black"textcolor="black"font="Dialog"italic="false"bold="false"justification="Left"
>salary : String</text>
   </group>
   <rectangle name="Fig0.5" x="168" y="72" width="112 height="112"
```

```
fill="0"fillcolor="white"stroke="1"    strokecolor="black"/>
 </group>
 <group name="Fig1" description="org.argouml.uml.diagram.static_structure.ui.FigClass[456,
80, 123,
96]pathVisible=false;stereotypeView=0;operationsVisible=true;attributesVisible=true;"href="10-
22-2--73-324c2761:127066bd6dc:-8000:0000000000000DE8" fill="1"
fillcolor="white"stroke="1"strokecolor="black">
   <private>
   </private>

   <rectangle name="Fig1.0" x="456" y="80" width="123" height="96"
fill="1"fillcolor="white"stroke="0"    strokecolor="0 255 255"/>
   <group name="Fig1.1"description="org.argouml.uml.diagram.ui.FigStereotypesGroup[456,
80, 123, 0]"
      href="10-22-2--73-324c2761:127066bd6dc:-8000:0000000000000DE8" fill="1"
fillcolor="white" stroke="1"
      strokecolor="black">
    <private>
    </private>

    <rectangle name="Fig1.1.0" x="456" y="80" width="123"
height="0"fill="1"fillcolor="white"stroke="1"    strokecolor="black"/>
 </group>
</pgml>
```

Fig 3.4. PGML file generated from two classes and a relation

3. 3. Core elements of GraphML

GraphML is a format which has been introduced to graph drawing community to ease the generation of graphs by a customized and user friendly format. Various graph drawing tools now implicitly use the GraphML format for saving a diagram drawn in those tools. This kind of graph format is very flexible, extendible with the injection of more packages and can be widely used for various purposes based on the users' needs. GraphML is also an XML based format and thus makes things simple for a user to understand the contents of the file format. GraphML provides some of the core features and properties of graphs, but does so in its own terminology, using terms such as node, edge etc.

3.3.1. Nodes

In GraphML or in basic graph theory, a node is a fundamental unit with which graphs are formed. The graphs can further be classified into directed or undirected based on the properties of the nodes and edges which connect them

```
<node id="1" type="Employee" pos="160 48">
        <port id="{Employee}"/>
        <data key="attrib">
                <attrib id="Terminal" type="2" bool="True"/>
        </data>
</node>
```

The node element's attributes include id, type and pos, and id is denoted by a number which can later be identified with the reference of "id". The 'type' attribute of a node specifies the characteristic of that node and the 'pos' attribute of a node reveals the position of a node on an xy plane.<port> is a child element to <node>, and this element gives a specific id on the node, which can be utilized to connect to a different node using an edge. The <data> element is customized depending on the user's requirement.

3.3.2. Edges

An edge is a line connecting two nodes that are described as a source node and a target node. The edge can still be classified as directed or undirected based on the direction of flow or a specific property mentioned in the relationship. The directed attribute is false if the edge has no specified direction and is true if there is a definite direction.

```
<edge type="{E}" directed="false" source="1" sourceport="{Employee}" target="2"
targetport="{Department}">
```

Here the source is the id of the node from which the edge is originated, and this edge is using the port {Employee} to connect to the target node through port {Department} of node2.

3.3.3. Port

A port is an attribute of a node and a node has numerous ports, such as North, South, East, West (in the example given below). A user can declare his/her own port names depending on the tool they use and requirement. A port name is mandatory for a node, and a <port> element has to be declared.

```
<!ELEMENT port (desc?,(data|port)*)>
<!ATTLIST port name    NMTOKEN  #REQUIRED>
```

The above DTD explains the usage of port in GraphML, and a name is given to the port when it is declared so that the edge can have them as an origin to the respective node

```
<node id="n0">
   <port name="North"/>
   <port name="South"/>
   <port name="East"/>
   <port name="West"/>
</node>
```

3.3.4. <desc>

The description element denoted by <desc> provides a brief description for the element by which it is declared. This element is not mandatory but adds some additional information about the element for a better understanding to a user. It is similar to a <comment> tag which provides some additional information to the user for better understanding.

3.3.5. <data>

The <data> element is used to declare an id which is used multiple times rather than using the entire id of the node or edge. For example, <keyid= “k1 for =”edge”>,which

means the id ‘k1’ is assigned for an edge and, irrespective of how many nodes a graph contains, the data for node is represented as

```
<edge source="n0" target="n1">
    <data key="k1">
<edge>
```

Here ‘k1’ is a reference to an edge and, similarly, a key can be assigned to all the elements, and a data element can be used to represent it as an id that we have declared in the key.

3.4. Sample GraphML

Fig 3.5. shows a simple graph that consists of seven nodes and eight edges, and the code generated for such a graph is given below. This graphml graph has been generated by graph-tool, a tool based on phyton [22].

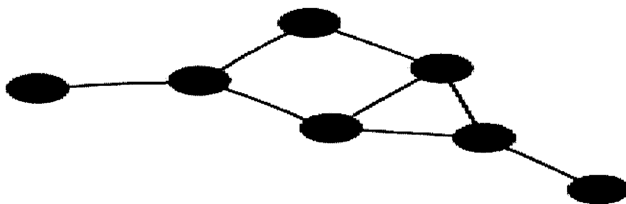

Fig 3.5. Sample GraphML file

The Fig 3.6 shows the code generated for a simple graph with 7 nodes and 8 edges. Each node and edge has a unique id and is referenced later with these ids.

GraphML Code

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml                                    xmlns=http://graphml.graphdrawing.org/xmlns
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <graph id="G" edgedefault="undirected">
    <node id="n0"/>
    <node id="n1"/>
```

```
  <node id="n2"/>
  <node id="n3"/>
  <node id="n4"/>
  <node id="n5"/>
  <node id="n6"/>
  <edge id="e0" source="n0" target="n1"/>
  <edge id="e1" source="n1" target="n2"/>
  <edge id="e2" source="n2" target="n6"/>
  <edge id="e3" source="n3" target="n4"/>
  <edge id="e4" source="n4" target="n5"/>
  <edge id="e5" source="n4" target="n2"/>
  <edge id="e6" source="n5" target="n6"/>
  <edge id="e7" source="n6" target="n1"/>
 </graph>
</graphml>
```

Fig 3.6. Generated GraphML code for Fig(3.5.)

3.5. Generated GraphML for a simple association

Fig 3.7. shows a customized GraphML code for the Employee-Department class diagram with a relationship 'Works for' between both the classes.

GraphML Code

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This GraphML document was generated from XMI by an XMI-to-GraphML conversion
style sheet.-->
<graphml xmlns:argouml="org.omg.xmi.namespace.UML"
xmlns:UML="org.omg.xmi.namespace.UML">
<graph id="untitledModel" edgedefault="undirected">
<desc>
<!--This graph is derived from a UML-class diagram-->
</desc>
<node id="1" type="Employee" pos="168 72">
<port id="{Employee}"/>
<data key="attrib">
<attrib id="Terminal" type="2" bool="True"/>
</data>
</node>
<node id="2" type="Department" pos="456 80">
<port id="{Department}"/>
<data key="attrib">
<attrib id="Terminal" type="2" bool="True"/>
</data>
```

```
</node>
<edge type="{E}" directed="false" source="1" sourceport="{Employee}" target="2"
targetport="{Department}">
<desc>
<!--This edge represents a relationship between classes in a UML Class diagram-->
</desc>
</edge>
</graph>
</graphml>
```

Fig 3.7. GraphML file for a simple graph with a single association.

3.6. Comparison of UML format and GraphML format

Fig 3.8. shows an element level comparison of both the formats that are used in this paper. Notes are provided with an explanation for each element, detailing the extent to which they can be substituted with resultant formats.

UML Format	Notes	GraphML Format
<XMI>	Substitutable, these are the root elements of both the formats	**<graphml>**
<UML:Model>	Substitutable, In GraphML all the keys to the corresponding <data> are declared in this element.	**<graph>**
<UML:Class>	Substitutable, Classes in UML class diagrams are represented as nodes in GraphML	**<node>**
<UML:Association>	Substitutable, Associations in Class-diagrams are known as edges in GraphML	**<edge>**
Class-name	For this transformation, we consider the class name,which is unique, to be the port name. Class diagrams cannot be declared with multiple ports	**<port>**
N/A	This has to be generated by the stylesheet that we have written, the stylesheet auto-generates <desc> for better understanding to the users.	**<desc>**
N/A	There is no element in XMI corresponding to	**<hyperedge>**

N/A	This element also has to be generated during transformation and we give a value for each <key>,and <data> is represented using this element.	**<key>**
<UML:Operation>	This element in XMI is used to store all methods inside a particular class.	**<key id =”method”>**
<UML:Attribute>	This element in XMI is used to store all the attributes and its properties.	**<key id =“attribute”>**
<UML:Multiplicity >	This element is used to store the multiplicity value of the corresponding Associations.	**<key id=”leftlimits”>** **<key id=”righlimits”>**
<PGML @x , @y>	This element gives the position of the class on an xy plane	**<node pos “ “>**

Fig 3.8. Element level comparison

As all the elements cannot be mapped between both formats, we try to extract most of the data possible from .uml and try to get a GraphML file .In this process, some of the elements are customized according to our needs and are generated by the stylesheet with processing instructions specified.

CHAPTER 4

IMPLEMENTATION OF XSLT

XML Stylesheet language transformation is a simple and the most extensively used method of transforming two XML files. As UML and GraphML are two formats based on XML, this paper uses XSLT as the mode of transformation. Transformation is easier using XSLT, and is simple to extend a stylesheet to incorporate multiple XML file formats at once.

There are a few alternatives to XSLT, such as XQuery, JSLT and Axgen. XQuery is a deterministic functional language which is used to transform one form of XML to another [26]. This language is designed for a large pool of XML data and retrieved using SQL like queries. So, for a user to use this language he/she has to be aware of SQL queries. This language feels natural for the users who are well acquainted with SQL. XQuery lacks document centric template processing, which makes it more difficult for formatting an XML document.

JSLT is language which is purely based on JavaScript and is considered an alternative to XSLT. This language does not use template based processing and any user who is aware of JavaScript can use this language to do XML transformations. In order to equip all the functionalities of XSLT into this language, JSON has to be passed instead of an XML document and the client side template must use AJAX to perform a transformation. These two features make this language even more complicated when compared to XSLT.

AXgen is a tool which also takes the input of XML but generates java classes for OJB by using Jakartas Velocity templates. This tool is built using some sophisticated

technologies and is used to take any input from an XMI generating tool like ArgoUML or Rational Rose and then generate its corresponding java classes. This tool is useful when its user want to understand a graph from a programmatic stand point, rather than a visual stand point.

In this paper, XSLT is used as a middleware for any kind of XML transformation in programming languages like Java, C# and .net, making it scalable. XSLT, when compared to other techniques of transformation, provides a clear, distinct way of mapping elements between two XML files. Irrespective of Java, C# or .net, a stylesheet can be embedded within an application to perform the transformation. One of the most important reasons to choose xslt for this paper is a need for template based transformation. Second, XML formatting can be achieved only by XSLT when compared to all the other alternatives discussed above. XSLT can be used to efficiently display the XML file on multiple web browsers when compared to other parsing techniques.

XSL (XML Stylesheet Language) is a language which is widely used to transform one XML format to a different XML format or some form of human readable format like HTML. With a wide variety of graph formats like GXL, XMI, GML, there is a need to transform one graph format to another for scalability, thereby reusing the same format with a minimal loss of data [15]. XSLT is a stylesheet written in XML which uses a basic paradigm of pattern matching. XSLT consists of XPath and XLink which together constitute XPointer and is considered a core part of transformation. XPath provides all the rules for transformation and XLink provides all the needed references from an external data source. XPath is a component of XSL which visualizes the XML document in a tree-structure and then distinguishes between different types of nodes in a document. XPath and

XLink together make XPointer [18], which is used in XSLT to specify certain rules of transformation in the form of a stylesheet.

As shown in the Fig4.1., XSLT applies a pattern matching technique wherein a custom template is called and a set of processing instructions are already pre-written in that template.

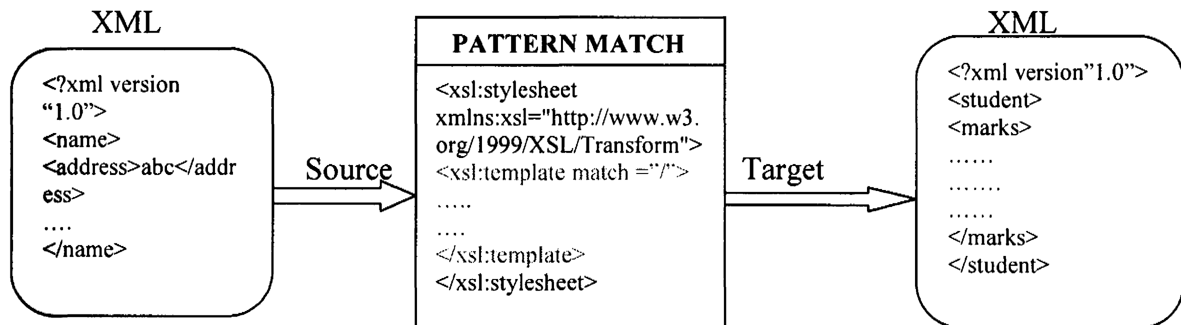

Fig 4.1. Pattern matching of XSLT

The processing instructions are written inside the <xsl:template> element, and the attribute match in this element checks for the corresponding match in the source document using a top-down approach, It then follows the processing instructions to give the desired target. XSLT, in other words, is also an XML document and contains normal processing instruction. In the block diagram above in Fig 4.1, template match performs a check from the root node of the source document and indicates a starting point of the transformation. If "/" is substituted by "address", an already existing template with the name "address" will be called and then applied.

XSLT is also based on XML and is used to transform XML to HTML, XHTML and a plain text document.XSL is a language with many elements. We will not use all of them, but with a few of the elements, we can make the transformation possible. More and more elements can be used for advanced users, but by keeping a novice user in mind, we

tend to make the stylesheet as simple as possible. In XSLT the transformations are described as rules which form a pattern, and this pattern is matched against the source tree and a resultant tree is achieved [5]. The resultant tree can have a similar structure to that of the source tree or can be completely different.

4.1. Elements used in the transformation

The Fig 4.2 below gives a description of some of the basic elements that have been used as a part of developing XGraphML. Column 1 lists all the elements and Column 2 gives a brief description of the respective elements.

ELEMENT	DESCRIPTION
xsl:stylesheet	This element stores the version of the stylesheet and the namespaces required for the stylesheet.
xsl:output	This element specifies the way the target document is supposed to be displayed as an output
xsl:variable	This element is used to specify variables with a pre-defined constant value.
xsl:choose	This element is used in XSLT as a switch case in any programming language. This element is supposed to be preceded by xsl:when and the condition is specified in this element
xsl:message	This element is used to specify a message to the output based upon a the value of the terminate attribute yes/no.
xsl:comment	This element is used to output a comment in the target document.

xsl:attribute	This element is used to create an attribute in the target document.
xsl:element	This element is used to create an element in the target document.
xsl:value-of	This element is used to get the value of an XML element in the source document and add it to the source document.
xsl:for-each	This element is similar to a for loop condition in any programming language.
xsl:if	This element is similar to an if clause in any other programming language.

Fig 4.2. Basic elements of a stylesheet and their description

4.2. XGraphML.xsl

Fig 4.3 outlines the actual stylesheet which is used for the transformation of UML to GraphML. We name the stylesheet as XGraphML, and all the XPath processing instructions are written within the stylesheet.

XGraphML.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
xmlns:UML="org.omg.xmi.namespace.UML"
xmlns:argouml="org.omg.xmi.namespace.UML">
        <xsl:output indent="yes" method="xml"/>
        <xsl:strip-space elements="*"/>
        <xsl:variable name="tool"
select="XMI/XMI.header/XMI.documentation/XMI.exporter/text()"></xsl:variable>
        <xsl:variable name="metamodel_name"
select="XMI/XMI.header/XMI.metamodel/@xmi.name"/>
        <xsl:variable name="metamodel_version"
select="XMI/XMI.header/XMI.metamodel/@xmi.version"/>

        <xsl:template match="/">
           <xsl:choose>
              <xsl:when test="$metamodel_name !='UML' ">
```

```
                <xsl:message terminate="yes">This source document is from an incorrect
model</xsl:message>
              </xsl:when>
              <xsl:when test="$metamodel_version !='1.4'">
                <xsl:message terminate="yes">This is an incorrect version</xsl:message>
              </xsl:when>
              <xsl:when test="$tool!='ArgoUML (using Netbeans XMI Writer version 1.0)'">
                <xsl:message terminate="yes"> The input is not derived from
ArgoUML</xsl:message>
              </xsl:when>
              <xsl:otherwise>
                <xsl:message terminate="no"> This is from a correct version, correct model and
correct tool</xsl:message>
              </xsl:otherwise>
            </xsl:choose>

            <xsl:comment> This GraphML document was generated from XMI by an XMI-to-
GraphML conversion style sheet.</xsl:comment>
            <graphml>
              <graph >
                <xsl:attribute name="id">
                  <xsl:value-of select="//XMI.content/UML:Model/@name"/>
                </xsl:attribute>
                <xsl:attribute name="edgedefault">undirected</xsl:attribute>
                <desc>
                  <xsl:comment>This graph is derived from a UML-class
diagram</xsl:comment>
                </desc>
                <xsl:for-each
select="//XMI.content/UML:Model/UML:Namespace.ownedElement/UML:Class">
                  <node>
                    <xsl:variable name="a" select="@xmi.id"/>
                    <xsl:attribute name="id">
               <!--        <xsl:variable name="cur" select='position()'/>
                      <xsl:value-of select="$cur" />
                      <xsl:variable name="pre" select=".">/xsl:variable> -->
                 <xsl:value-of select="$a"/>
                    </xsl:attribute>
                    <xsl:attribute name="type">
                      <xsl:value-of select="@name"/>
                    </xsl:attribute>
                    <xsl:attribute name="pos">
                      <xsl:for-each select="//pgml/group">
                        <xsl:if test="@href=$a">
                          <xsl:value-of select="rectangle[1]/@x"/>
                          <xsl:text> </xsl:text>
                          <xsl:value-of select="rectangle[1]/@y"/>
                        </xsl:if>
                      </xsl:for-each>
                    </xsl:attribute>
```

```
                    <port>
                        <xsl:attribute name="id">
                            <xsl:text>{</xsl:text>
                            <xsl:value-of select="@name"/>
                            <xsl:text>}</xsl:text>
                        </xsl:attribute>
                    </port>
                    <data key="attrib">
                        <attrib>
                            <xsl:attribute name="id">Terminal</xsl:attribute>
                            <xsl:attribute name="type">2</xsl:attribute>
                            <xsl:attribute name="bool">True</xsl:attribute>
                        </attrib>
                    </data>
                </node>
            </xsl:for-each>
              <xsl:for-each
select="//XMI.content/UML:Model/UML:Namespace.ownedElement/UML:Association">
                <edge>
                    <xsl:attribute name="type">{E}</xsl:attribute>
                    <xsl:attribute name="directed">false</xsl:attribute>

                    <xsl:for-each
select="UML:Association.connection/UML:AssociationEnd[1]/UML:AssociationEnd.participant
/UML:Class">
                        <xsl:variable name="classid" select="@xmi.idref"/>
                        <xsl:attribute name="source">
                            <xsl:value-of select="$classid"/>
                        </xsl:attribute>
                        <xsl:attribute name="sourceport">
                            <xsl:for-each
select="//XMI.content/UML:Model/UML:Namespace.ownedElement/UML:Class">
                                <xsl:if test=" @xmi.id=$classid">
                                    <xsl:text>{</xsl:text>
                                    <xsl:value-of select="@name"/>
                                    <xsl:text>}</xsl:text>
                                </xsl:if>
                            </xsl:for-each>
                        </xsl:attribute>
                    </xsl:for-each>
                    <xsl:for-each
select="UML:Association.connection/UML:AssociationEnd[2]/UML:AssociationEnd.participant
/UML:Class">
                        <xsl:variable name="classid" select="@xmi.idref"/>
                        <xsl:attribute name="target">
                            <xsl:value-of select="$classid"/>
                        </xsl:attribute>
                        <xsl:attribute name="targetport">
                            <xsl:for-each
select="//XMI.content/UML:Model/UML:Namespace.ownedElement/UML:Class">
```

```
                              <xsl:if test=" @xmi.id=$classid">
                                <xsl:text>{</xsl:text>
                                <xsl:value-of select="@name"/>
                                <xsl:text>}</xsl:text>
                              </xsl:if>
                            </xsl:for-each>
                          </xsl:attribute>
                          <desc>
                            <xsl:comment>This edge represents a relationship between classes in a
UML Class diagram</xsl:comment>
                          </desc>
                        </xsl:for-each>
                      </edge>
                    </xsl:for-each>
                  </graph>
                </graphml>
              </xsl:template>
</xsl:stylesheet>
```

Fig 4.3. XGraphML.xsl

In the stylesheet above, we declare three variables, tool, meta_model and metamodel_version along with some values, and we retrieve the values later in the stylesheet using a prefix' $'.

4.2.1. Initial Version Check and Validation

```
    <xsl:choose>
      <xsl:when test="$metamodel_name !='UML' ">
        <xsl:message terminate="yes">This source document is from an incorrect
model</xsl:message>
      </xsl:when>
      <xsl:when test="$metamodel_version !='1.4'">
        <xsl:message terminate="yes">This is an incorrect version</xsl:message>
      </xsl:when>
      <xsl:when test="$tool!='ArgoUML (using Netbeans XMI Writer version
1.0)'">
        <xsl:message terminate="yes"> The input is not derived from
ArgoUML</xsl:message>
      </xsl:when>
      <xsl:otherwise>
```

```
            <xsl:message terminate="no"> This is from a correct version, correct model
and correct tool</xsl:message>
        </xsl:otherwise>
     </xsl:choose>
```

This part of the stylesheet performs the initial validation of the UML file, which has been generated by ArgoUML or any other tool. If the file is generated by a different tool apart from ArgoUML, it terminates and gives an error message, "**generated from an incorrect tool**". <xsl:choose> is an XML version of switch case as in OOP(Object oriented programming). The first <xsl:when> checks for the metamodel of the XML. As the metamodel used by ArgoUML is "UML", the stylesheet checks for the metamodel and if incorrect, <xsl:message> is passed over the control, and the processing is terminated. A message is displayed saying that the source document is from an incorrect model. The second <xsl:when> checks for the version of metamodel if it is equal to 1.4, if not, <xsl:message> is executed and an error message is displayed to the user.

If all the initial validation is passed, then the last <xsl:otherwise> is executed where in the <xsl:message terminate="no"> will execute, which implies that all the previous validations were true. A message is still displayed to the user saying "**This is from a correct version, correct model and correct tool.**" This message informs the user that the input is from ArgoUML and is from a correct model. Further parts of the stylesheet are the actual transformation of the XMI document after validation.

As mentioned in the previous sections, <xsl:template match ="/"> is a rule which starts processing from the root node of the source document. In our case, <XMI> is the root node of the input document. <xsl:comment> is used to place a comment in the target tree as we use <!......> or /*....*/in other programming languages. There is a <key> element that is

assigned to either node or edge using an attribute “id” and has different property names, such as attribute and methods.

4.2.2. Classes or Nodes

```
<xsl:for-each
select="//XMI.content/UML:Model/UML:Namespace.ownedElement/UML:Class">
```

The above “for-each” statement extracts all the data required from within the <UML:Class> element of the XMI. This data is transformed as a node into the GraphML format. Irrespective of the number of classes in a class diagram, the iteration takes place until the last <UML:Class> element and the corresponding nodes are created in the target tree.

4.2.3. Methods

```
<xsl:for-each select="UML:Classifier.feature/UML:Operation">
    <data key ="method">
        <xsl:value-of select="@name"/>
    </data>
</xsl:for-each>
```

The above “for-each” loop in the stylesheet checks for the <UML:Operation> element in source document and then creates a <data> element with the <key> value “method.” The <xsl:value-of> extracts data corresponding to the name attribute inside the <UML:Operation> element. All the methods declared in a particular class are retrieved and stored with the key value “method” and placed in their respective classes.

4.2.4. Attributes

Every class can have many attributes or none depending upon the users' needs and the project's requirements. We took both these scenarios into account and then wrote a condition in the stylesheet that checks for data and retrieves it to a target document.

```
<xsl:for-each select="UML:Classifier.feature/UML:Attribute">
    <data key ="attribute">
          <xsl:value-of select="@name"/>
     </data>
 </xsl:for-each>
```

The value of the attribute is classified within <data key="attribute"> and the attributes of a class are extracted from the <UML:Atrribute> element of the source with an attribute name. If there are no attributes, the value will be left empty, but a random key is generated.

4.3. Association or Edges

Associations in a class diagram and edges in GraphML are the same. An edge in GraphML has many attributes, such as id, source, sourceport, target and targetport. In order to extract all the data from the class diagram, we use the class name as the port name for the transformation because Class diagrams do not have a concept of ports.

```
<xsl:for-each
select="//XMI.content/UML:Model/UML:Namespace.ownedElement/UML:Associati
on">
   <edge>
       <xsl:attribute name="id">
             <xsl:value-of select="@xmi.id"/>
        </xsl:attribute>
   <edge>
```

The "id" of an association is specified inside a <UML:Association> element with an auto-generated id by ArgoUML and is stored in xmi.id attribute. The <value-of>

extracts the value of xmi.id and then copies the value to the resultant graphml. The value extracted from this is classified into an 'id' attribute for edge.

4.3.1. Source

The source attribute in a class diagram is the class from which the association originates. In our example, Employee would be the source.

```
<xsl:for-each
select="UML:Association.connection/UML:AssociationEnd[1]/UML:AssociationEnd.
participant/UML:Class">
    <xsl:variable name="classid" select="@xmi.idref"/>
        <xsl:attribute name="source">
            <xsl:value-of select="$classid"/>
        </xsl:attribute>
</xsl:for-each>
```

The for-each condition checks for <UML:AssociationEnd.participant> and drills down to the <UML:Class> element where the id is retrieved using @xmi.idref.

4.3.2. Sourceport

As the UML Class diagram does not employ the concept of ports, for transformation we consider source port the name of the class.

```
<xsl:attribute name="sourceport">
    <xsl:for-each
select="//XMI.content/UML:Model/UML:Namespace.ownedElement/UML:Class">
        <xsl:if test=" @xmi.id=$classid">
            <xsl:text>{</xsl:text>
            <xsl:value-of select="@name"/>
            <xsl:text>}</xsl:text>
        </xsl:if>
    </xsl:for-each>
</xsl:attribute>
```

4.3.3. Target

Target is the target class in the class diagram or a node in GraphML to which an association or an edge is connected. The first occurring <UML:AssociationEnd.participant> element in the source document is the source of the association, and the second element corresponds to the target. Any Association or edge cannot have more than two <UML:AssociationEnd.participant> elements as their children to the<UML:Association> element.

```
<xsl:for-each
select="UML:Association.connection/UML:AssociationEnd[2]/UML:AssociationEnd.
participant/UML:Class">
      <xsl:variable name="classid" select="@xmi.idref"/>
            <xsl:attribute name="target">
                  <xsl:value-of select="$classid"/>
            </xsl:attribute>
</xsl:for-each>
```

4.3.4 Target Port

The target port is in the recipient class where the association ends, and it can be retrieved from the '@name' attribute value, which corresponds to the name of the class.

```
<xsl:attribute name="targetport">
    <xsl:for-each
select="//XMI.content/UML:Model/UML:Namespace.ownedElement/UML:Class">
              <xsl:if test=" @xmi.id=$classid">
                        <xsl:value-of select="@name"/>
              </xsl:if>
     </xsl:for-each>
 </xsl:attribute>
```

4.3.5 Directed or Undirected Edge

An edge can be either directed or undirected without any specific direction. We specify certain rules in the XSLT, which determines if the edge is directed or undirected. We create a <data key ="directededge"> element, and if the edge is directed, we output the text value as true. If the edge is undirected, the value is false.

```
<data key="directededge">
  <xsl:variable name="navigationcount" select="count(UML:Association.connection
                /UML:AssociationEnd[@isNavigable='true'])"/>
        <xsl:choose>
            <xsl:when test="$navigationcount>1">directedfalse</xsl:when>
            <xsl:otherwise>directedtrue</xsl:otherwise>
        </xsl:choose>
</data>
```

CHAPTER 5

IMPLEMENTATION OF THE TOOL-XGRAPHML

In this Chapter we give a brief overview of the XGraphML tool that has been designed for the transformation of UML to GraphML using the XSLT, JAXP and Java Swing. This tool provides a user interface where in the user can import an .uml file, select the corresponding stylesheet and transform to GraphML. The most important feature is that the user can see the actual transformation taking place in the other pane of the tool. The user can compare both the formats together and then save the transformed GraphML to his/her desired location on the computer.

5.1. Block Diagram of the transformation.

Fig 5.1. explains the transformation of UML to GraphML. ArgoUML is the tool used to draw a class diagram and is stored in .uml format.

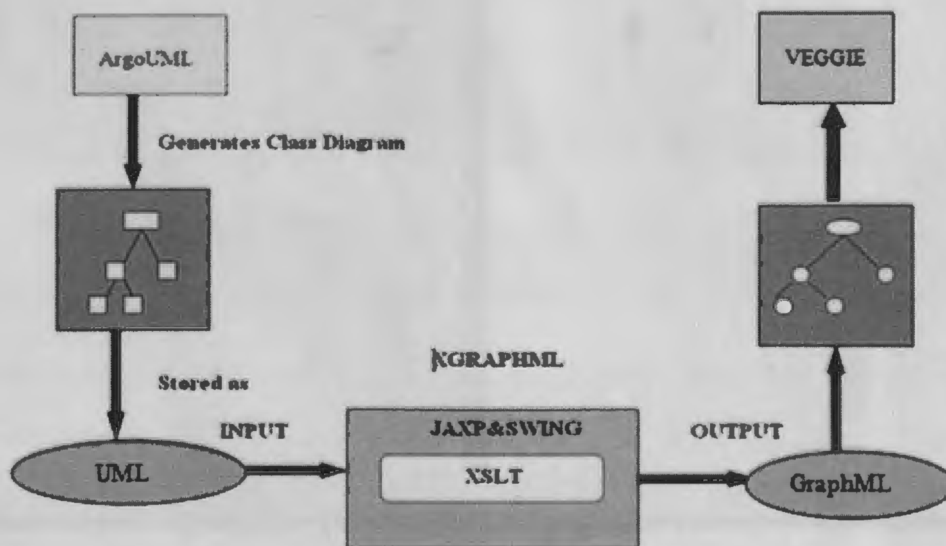

Fig 5.1. Block diagram of the transformation

This file is used as an input to XGraphML tool which has been developed to give us the targeted output, i.e., GraphML. This output file is further utilized by tools like Veggie for further analysis with graph grammars.

5.2. Basic Interface of the tool

This tool is built using Java Swing and JAXP, and this tool is still in the initial stages of development. Many enhancements can still be made to it, all of which will be mentioned in chapter 7 under future work.

Fig 5.2. Shows a basic interface of XGraphML, which is divided into 2 panes. The pane on the left side of the tool is used to load an .uml file generated from the ArgoUML tool. Loading this file can be either done by clicking file and then selecting open or directly clicking the open button from the toolbar as shown in Fig 5.3. and Fig 5.4.

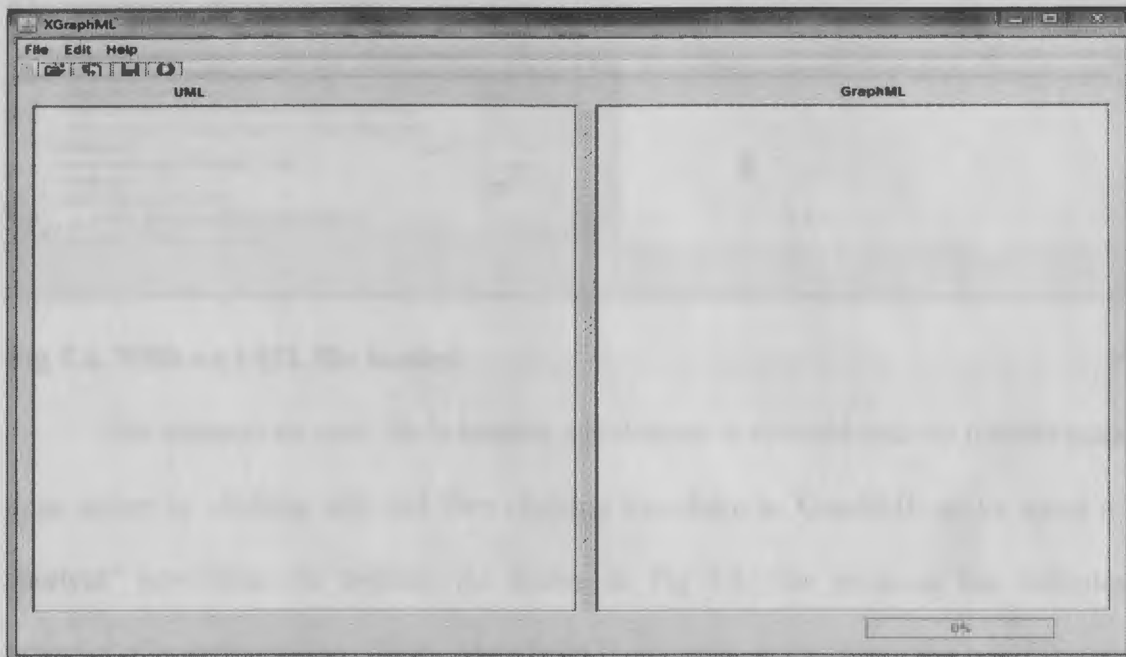

Fig 5.2. Screenshot of XGraphML

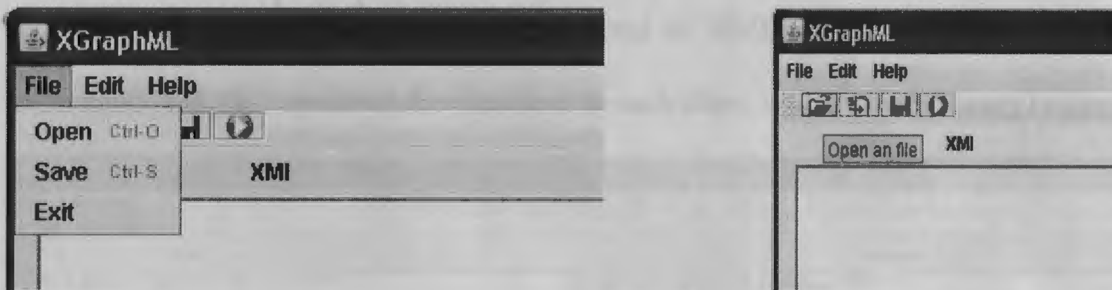

Fig 5.3. Opening an UML file

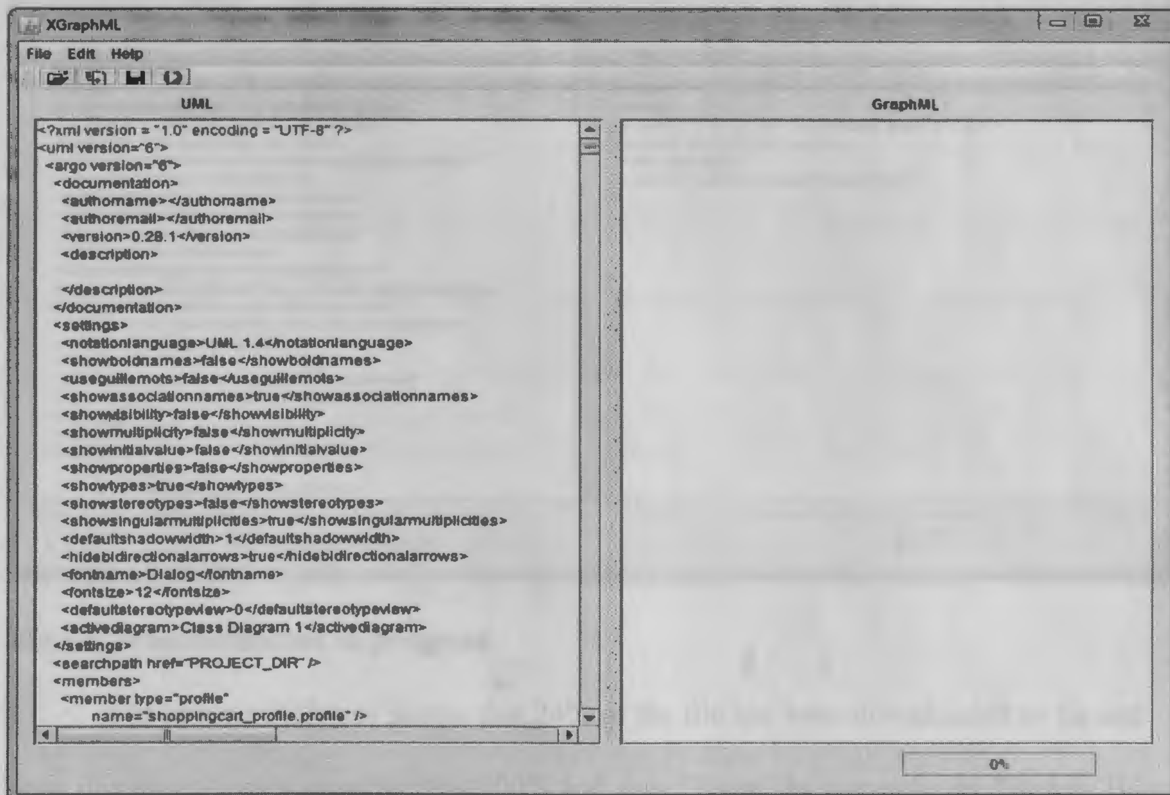

Fig 5.4. With an UML file loaded

The moment an .uml file is loaded, a stylesheet is selected and the transformation is done either by clicking edit and then clicking transform to GraphML or by using a "run shortcut" icon from the toolbar. As shown in Fig 5.5., the progress bar indicates the progress of transformation, which starts from 0 and ends at 100 when the transformation is done. A message is displayed on the screen that changes "Transformation in progress" to "Done," simultaneously. Anyone using this tool can edit on both the panes of the tool

depending on the user's needs and can be used as XML editors. The user will clearly understand both the formats as they are next to each other, which makes it easy to compare.

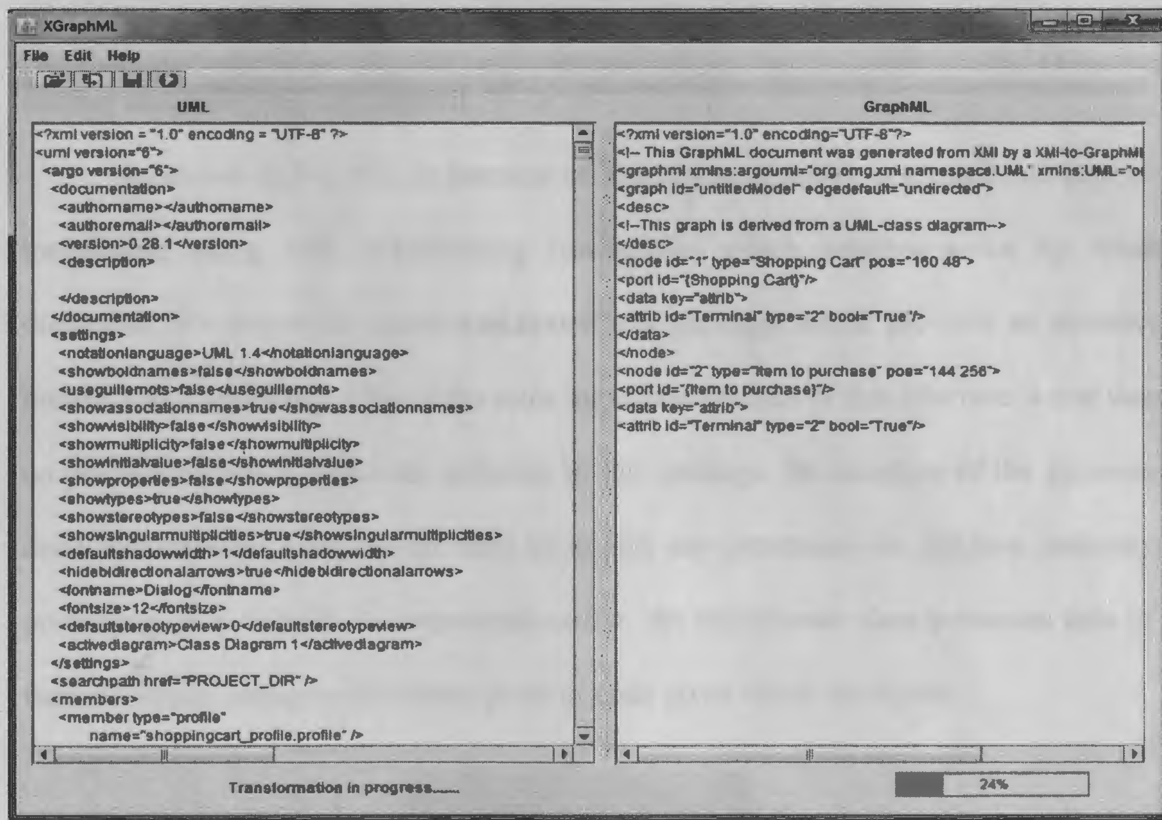

Fig 5.5.Transformation in progress

The process indicator shows that 24% of the file has been downloaded so far and once it is done, the progress bar hits 100% and says "Done" as shown in the Fig 5.6. This provides an interactive feature for the user wherein he/she can look at the progress and know when the entire transformation is done.

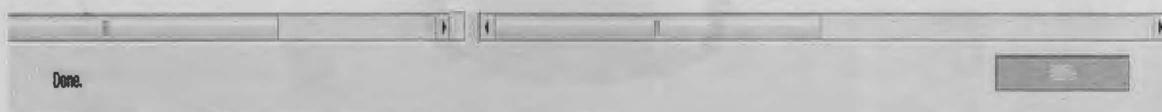

Fig 5.6. Progress indicator

5.3. Implementation of Transformer class

XGraphML uses JAXP to read the input of data i.e. .uml as a stream of data, apply the stylesheet that contains all the processing rules for transformation and then output another stream of data as GraphML.

As shown in Fig 5.7, an instance of TransformerFactory is created and sent to the transformer along with transforming instructions which together make up resultant document. We import the **javax.xml.transform** package, which provides an interface to invoke XSLT stylesheet. One of the most important features of this interface is that there is no predefined XSLT processor included in this package. Its selection of the processor is done dynamically. There is no need to import any processors or jar(Java Archive) for processing, it is in-built. As mentioned earlier, the transformer class processes data in the form of Stream using the following piece of code given below the figure.

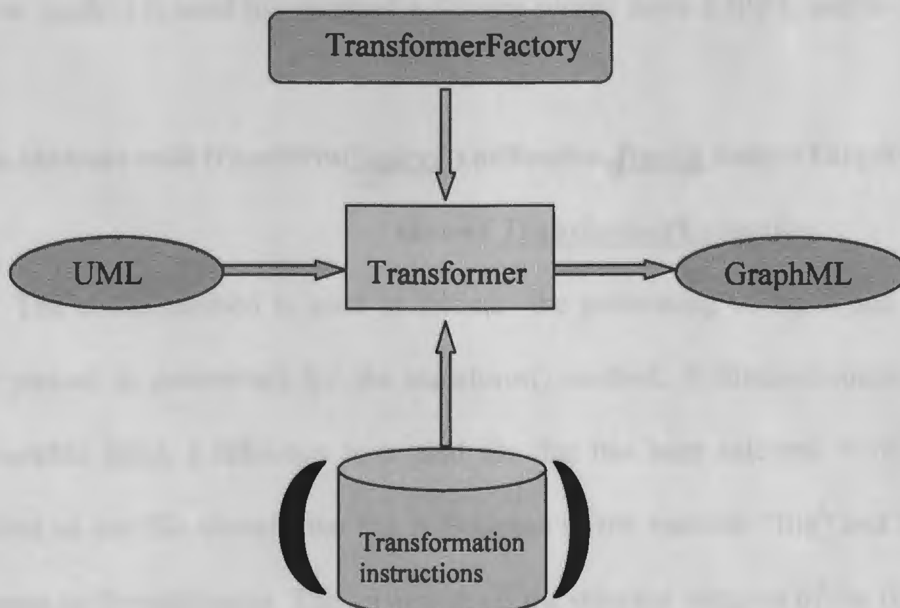

Fig 5.7. Usage of Transformer class.

```
StreamSource input = new StreamSource(file);
StreamSource xslt = new StreamSource(file1);
StreamResult output = new StreamResult("file:///c:/result.graphml");
TransformerFactory tf = TransformerFactory.newInstance();
  try {
            tf.newTransformer(xslt).transform(input, output);
      } catch (TransformerConfigurationException e1) {
            System.err.println("Exception in Transformer Configuration " +
e1.getMessage());
      e1.printStackTrace();
                  } catch (TransformerException e1) {
      System.err.println("Error in transforming" + e1.getMessage());
      e1.printStackTrace();
      }
```

Various methods of the transformer Class that are used in the transformation through XGraphML are

1. public StreamSource(java.io.File f)

The above method is used to construct a Stream source from a file f, and in our case, f is file.

2. public abstract void transform(Source xmlSource, Result outputTarget)

throws TransformerException

The above method is used to initiate the processing of input and output when both are passed as parameters for the transform() method. A StreamSource is initialized with a variable input, a reference to a .uml file that has been selected from Jfilechooser. Irrespective of any file chosen, the file is assigned to the variable "file" and file is sent as an argument to StreamSource. This in turn reads the selected instance of the file and creates a new object of StreamSource with input as a reference. A TransformerFactory instance is created to take processing instructions or the stylesheet required for transformation.

Multiple instances of TransformerFactory can be created to process multiple stylesheets in many threads; hence, it is considered thread-safe.

StreamResult is used to get the resultant or the transformed document and is initialize it to a variable 'result'. This result is stored at a temporary location "D:\result.graphml". The result.graphml is displayed on the screen and an option of save is provided in the toolbar of XGraphML, and users can save the file with a custom name and at a custom location.

CHAPTER 6

CASE STUDY

6.1. Class diagram for a Shopping Cart

In the Fig 6.1., a class diagram for a shopping cart has been drawn and the project has been saved as shoppingcart.uml [24]. The .uml file is imported into XGraphML, and a resultant output i.e. GraphML is achieved by transformation. The generated graphml is saved as "shoppingcart.graphml".

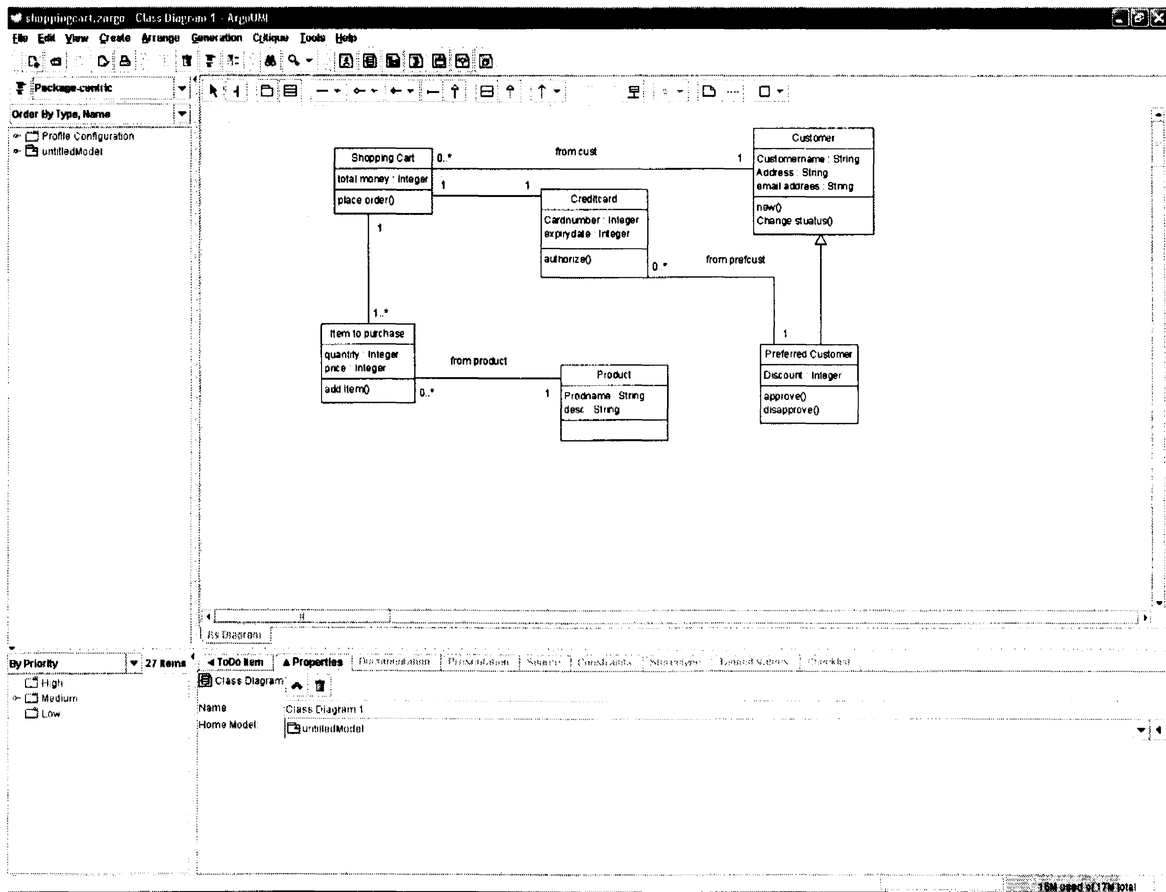

Fig 6.1. Shopping Cart example

We utilize as many elements as possible from the .uml format and then generate graphml based upon the DTD specified in the standard. The above class diagram consists of 6 classes, namely shopping cart, credit card, Item to purchase, customer and preferred

customer, along with six relations connecting these classes. Fig 6.2. below shows a brief listing of code produced in graphml format for the shoppingcart class diagram drawn in ArgoUML. This code has been produced using the XGraphML. This graphml file is further analyzed in Veggie using graph grammars.

Shoppingcart.graphml


```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This GraphML document was generated from XMI by an XMI-to-GraphML conversion style sheet.-->
<graphml xmlns:argouml="org.omg.xmi.namespace.UML" xmlns:UML="org.omg.xmi.namespace.UML">
<graph id="untitledModel" edgedefault="undirected">
<desc>
<!--This graph is derived from a UML-class diagram-->
</desc>
<node id="1" type="Shopping Cart" pos="160 48">
<port id="{Shopping Cart}"/>
<data key="attrib">
<attrib id="Terminal" type="2" bool="True"/>
</data>
</node>
<node id="2" type="Item to purchase" pos="144 256">
<port id="{Item to purchase}"/>
<data key="attrib">
<attrib id="Terminal" type="2" bool="True"/>
</data>
</node>
<node id="3" type="Customer" pos="664 24">
<port id="{Customer}"/>
<data key="attrib">
<attrib id="Terminal" type="2" bool="True"/>
</data>
</node>
<node id="4" type="Preferred Customer" pos="672 280">
<port id="{Preferred Customer}"/>
<data key="attrib">
<attrib id="Terminal" type="2" bool="True"/>
</data>
</node>
<node id="5" type="Creditcard" pos="408 96">
<port id="{Creditcard}"/>
<data key="attrib">
<attrib id="Terminal" type="2" bool="True"/>
</data>
</node>
```

```
<node id="6" type="Product" pos="432 304">
<port id="{Product}"/>
<data key="attrib">
<attrib id="Terminal" type="2" bool="True"/>
</data>
</node>
<edge type="{E}" directed="false" source="1" sourceport="{Shopping Cart}" target="2"
targetport="{Item to purchase}">
<desc>
<!--This edge represents a relationship between classes in a UML Class diagram-->
</desc>
</edge>
<edge type="{E}" directed="false" source="1" sourceport="{Shopping Cart}" target="5"
targetport="{Creditcard}">
<desc>
<!--This edge represents a relationship between classes in a UML Class diagram-->
</desc>
</edge>
<edge type="{E}" directed="false" source="1" sourceport="{Shopping Cart}" target="3"
targetport="{Customer}">
<desc>
<!--This edge represents a relationship between classes in a UML Class diagram-->
</desc>
</edge>
<edge type="{E}" directed="true" source="4" sourceport="{PrefferedCustomer}" target="3"
targetport="{Customer}">
<desc>
<!--This edge represents a relationship between classes in a UML Class diagram-->
</desc>
</edge>
<edge type="{E}" directed="false" source="5" sourceport="{Creditcard}" target="4"
targetport="{Preferred Customer}">
<desc>
<!--This edge represents a relationship between classes in a UML Class diagram-->
</desc>
</edge>
<edge type="{E}" directed="false" source="6" sourceport="{Product}" target="2"
targetport="{Item to purchase}">
<desc>
<!--This edge represents a relationship between classes in a UML Class diagram-->
</desc>
</edge>
</graph>
</graphml>
```


Fig 6.2. Shoppingcart.graphml

6.2. Case Study

Based upon the completed transformation, a case study has been performed to analyze the level of transformation achieved using XGraphML. In the Class diagram we have 6 classes and 6 relationships between them. Fig 6.3. lists all classes in the shopping cart class diagram along with their respective nodes from graphml.

Classes in Class Diagrams	Nodes in GraphML
Shopping Cart	1
Item to purchase	2
Customer	3
Preferred Customer	4
Creditcard	5
Product	6

Fig 6.3. List of classes in UML format and GraphML format

As listed in Fig 6.4.,there are six associations in the shoppingcart class diagram, column 1 shows a list of all relations in the class diagram and column 2 list both source and target. As mentioned earlier, an edge can be both directed and un-directed, and in our case we have five undirected edges and one directed edge.

A directed edge in graphml is represented by an attribute directed="false", which means it is not a directed edge. An edge from preferred customer to customer is directed, so graphml generates a value of the attribute as 'true'.

Associations in Class Diagram	Source and Targets in GraphML
From cust	Source="1" Target="1"
Shoppingcart→creditcard	Source="1" Target="5"
Shoppingcart→Item to purchase	Source="1" Target="2"
Preferedcustomer→customer	Source="4" Target="3"
From prefcust	Source="5" Target="4"
Product→Item to purchase	Source="6" Target="2"

Fig 6.4. List of associations in UML format and GraphML format

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this chapter we have outlined some of the advantages of using XGraphML when compared to other traditional tools used for transformations. We have eliminated all core limitations, having very few exceptions and have proceeded further with the transformation. We have listed all possibilities that would enhance and improve the efficiency of the tool towards the end of this chapter.

XGraphML provides a very simple interface with only needed options which a novice user can operate without undergoing any kind of training or referring to any technical documentation. If a user has a UML file, transformation from .uml to GraphML is done by selecting 2 buttons. This tool provides a two-pane interface where a user can `compare both source and target together and list all the discrepancies between both formats. The results generated by XGraphML can be validated against the source document. Uml, the source file in this transformation, is uploaded into ArgoUML and the corresponding transformed graphml file is uploaded into Veggie. Both these tools display the exact structure of the graph, if the transformation is performed as expected. The user can visually analyze the results and then modify the stylesheet as required to match both the source and the target. In order to test the transformation, a class is removed from the uml file, and then the corresponding node in the graphml file would be eliminated. Similarly, a relationship in a class diagram corresponds to an edge in graphml. The results are further analyzed with respect to the position of the classes on the x-y plane, and if the position of class is changed in the uml file and transformed using XGraphML, the resulting graphml file would reflect the changed position of the class, but as a node.

All the commercial parsers have processors installed in them, and a user has to configure a scenario with a source stylesheet and a processor required for the transformation. In order for a user to perform this operation, he/she has to be technically well versed with xslt processors for him/her to differentiate between processors. In XGraphML the processor functionality is provided by the transformer class. JAXP provides an API for transformation that eliminates the need of an XSLT processor.

7.1. FUTURE WORK

Due to the increasingly extensive use of XML and visual representation models in software engineering, there is a large scope of development in the area of graph transformation [12]. This tool can be further enhanced to incorporate all the graph formats and can be used for transformations with only a few minor changes to the stylesheet. This tool is built only for GraphML, but can be enhanced to get any desired output if the stylesheet required for processing is written correctly with all appropriate rules embedded in it.

Only class diagrams were taken into account while designing XGraphML, but this can be extended to other UML diagrams, such as the activity and State Chart diagrams, along with others.. This would make a tool scalable to all the UML diagrams present on board. This tool can be further enhanced by generating a visual model in the form of a graph simultaneously while the transformation is in progress. This visual model can be generated in the form of a 3D diagram that offers humongous opportunities for human computer interfaces [20].

Presently, we consider any class-diagrams generated only from ArgoUML, and the initial validation is purely w.r.t. ArgoUML. If a UML is generated using a different modeling tool, it would not pass the validation, and the processing would terminate with an exception displayed on the console. If the user is utilizing a different modeling tool to generate .uml files, then the stylesheet that has been used in XGraphML may need a minute manipulation. Otherwise, the user can write his or her stylesheet pertaining to the tool which is being used.

Some of the elements had to be ignored from source documents, as all the elements of the .uml format do not match the GraphML specification, and some of elements had to be auto-generated by the stylesheet solely for this paper.

BIBLIOGRAPY

[1] ArgoUML, retrieved from http://argouml.tigris.org/ on (07/15/2008).

[2] XML, retrieved from http://www.w3.org/XML on (02/22/2008).

[3]Linus Tolke and Markus Klink. An introduction to developing ArgoUML. http://argouml-downloads.tigris.org/nonav/argouml-0.24/cookbook-0.24.pdf retrieved on (5/17/2009).

[4]XMI,retrieved from http://www.omg.org/technology/documents/formal/xmi.htm on (10/20/2009).

[5] XSLT 1.0., retrieved from http://www.w3.org/TR/xslt on (10/20/2009).

[6] UML, retrieved from http://www.uml.org/ on (04/25/08).

[7] Eclipse, retrieved from http://www.eclipse.org/ on (05/15/2009).

[8] Alejandro Ramirez, Philippe Vanpeperstraete, Andreas Rueckert, Kunle Odutola, Jeremy Bennett, Linus Tolke, and Michiel van der Wulp. ArgoUml user manual, retrieved from http://argouml-stats.tigris.org/documentation/manual-0.24-single/argomanual.html on (05/05/2008).

[9] Daniel Volk, "XIG-An XSLT-based XMI2GXL-Translator". Ph.D. *Thesis,* Department of Computer Science, University of the Federal Armed Forces, 2001.

[10] Sundeep.V, "A tool for Aspect Oriented Modeling using State-Charts". *Master's Paper*, Department of Computer Science, University of NorthDakota, 2007.

[11] K.L. Ates and K. Zhang, "Constructing VEGGIE: Machine learning for Context-Sensitive Graph Grammars"*Proc. 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'07)*, October 2007, IEEE CS Press, 456-463.

[12] AlHakami.H, "From Object-Oriented to Components: Generating Component-Based

Software from UML Diagrams". *Master's project*, Department of Computer Science, King Saud University, 2007.

[13] U Brandes, J Lerner, C Pich,"GXL to GraphML and vice versa with XSLT", *Proc. 2nd Intl. Workshop Graph-Based Tools (GraBaTs '04),* 2004.

[14] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, M. Marshall, GraphML, Retrieved from http://graphml.graphdrawing.org/ on (6/12/2007).

[15] R. Holt, A. Schurr, S. Elliott, A. Winter, Graph Exchange Language, retrieved from http://www.gupro.de/GXL/ on (6/12/2007).

[16] OMG Unified Modeling Language Specification, Version 2.0, retrieved from http://www.omg.org on (08/15/2009).

[17] JAXP: Java API for XML Processing (JAXP) 1.4., retrieved from http://java.sun.com/ /webservices/jaxp/index.jsp on (06/18/08).

[18] XML Path Language (XPath) 2.0., retrived from http://www.w3.org/TR/xpath20 on (06/18/08).

[19] Timothy J. Grose, Gary C.Doney, Stephen A. Brodsky. Mastering XMI: Java Programming with XMI, XML and UML, NY: John Wiley & Sons, Inc, April 2001.

[20] Brad Myers , Scott E. Hudson , Randy Pausch, "Past, present, and future of user interface software tools", *ACM Transactions on Computer-Human Interaction (TOCHI)*, v.7 n.1, p.3-28, March 2000.

[21] Rainer Conrad, Dieter Scheffner, J. Christoph Freytag, "XML Conceptual Modelling using UML, *In Proceedings of the Nineteenth International Conference on Conceptual Modeling (ER2000),* October 2000.

[22] Graph tool, retrieved from http://projects.forked.de/graph-tool/ (05/15/2010).

[23] XMItoHTML, retrieved from http://www.objectsbydesign.com/projects/ /xmi_to_html.html on (05/15/2010).

[24] Shopping-cart class diagram, Retrieved from http://www.databaseanswers.org/data_models/uml_class_diagram_for_shopping_cart/index.htm on (05/05/2009).

[25] PGML, retrieved from http://www.w3.org/TR/1998/NOTE-PGML-19980410.html on (08/12/2009).

[26] A Critical Analysis of XSLT technology for XML Translation, retrieved from http://ruleml.org/papers/CriticalAnalysisXSLT.pdf on (08/12/2009).