GRAPH DATA CONVERSION AND TREE VISUALIZATION

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

By

Srinivas Reddy Guduru

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

Major Department:
Computer Science

December 2011

Fargo, North Dakota

North Dakota State University
Graduate School

Title

GRAPH DATA CONVERSION AND TREE VISUALIZATION

By

SRINIVAS REDDY GUDURU

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

Department of Computer Science

SUPERVISORY COMMITTEE:

Dr. Jun Kong	Dr. Jun Kong
Chair (typed)	Chair (signature)
Dr. Kendall Nygard	Dr. Kendall Nygard
Dr. Simone Ludwig	Dr. Simone Ludwig
Dr. Jerry Gao	Dr. Jerry Gao

Approved by Department Chair:

	Brian M. Slator
Date	Signature

ABSTRACT

The first part of the paper is converting graph data format from GXL to GraphML. GXL and GraphML are both XML-based languages for representing graph data. The HP page segmentation tool produces output graph data in GXL format, and the Visual Environment for Graph Grammar Induction and Engineering (VEGGIE) tool takes input graph data in GraphML format. To make these two tools interoperable, the Graph Data converter changes a GXL graph structure data input into a GraphML graph structure output. XSLT is used by this tool for conversion; XSLT is a simple, functional language that is often used to convert an input XML document into an output XML document.

The second part of the paper focuses on tree visualization to generate additional views in VEGGIE for a given graph. Present VEGGIE tree visualization is limited to JTree View which makes it hard to analyze a complex user interface. For in-depth analysis of user interfaces, we need more user-friendly and interactive tree views. Moreover, for further analysis, it is useful if the generated views can be saved. Prefuse libraries are used to enhance the VEGGIE views. Prefuse is a Java-based visualization toolkit: it provides rich, interactive, and user-friendly visualizations. VEGGIE is enhanced with GraphView, TreeView, and Simplified Tree view. HTML is also generated to view the tree structure in HTML browsers.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my adviser, Dr. Jun Kong, for giving me the opportunity to participate in his research work. His advice, guidance, and support greatly helped in making this work possible. Also, I would like to sincerely thank Drs. Kendall Nygard, Simone Ludwig, and Jerry Gao for serving on my committee.

I would like to thank all the professors and faculty of the computer science department for their continued support and encouragement through my master's studies. I would also like to thank HP Labs, Israel, for providing funds for this research project. Special thanks to my brother, Ramakrishna, and cousin, Vasumathi, for their encouragement and support. Finally, I would like to thank my family members for their support. Without them, it would have been impossible for me to pursue my graduation.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

CHAPTER 1. INTRODUCTION

This paper solves two challenging issues: Graph Data Conversion and Tree Visualization. Graph Data Conversion changes the graph data represented in GXL format to GraphML format. Tree Visualization provides interactive visualizations for the graph data.

1.1. Graph Data Conversion

Graphical user interfaces can have different layouts, and they contain information, navigation links, images, etc. To better serve end users, a typical user's interaction patterns and their effectiveness need to captured and semantically analyzed. To analyze user interfaces, the interface components first have to be extracted, and then, the generated components have to be analyzed. HP Labs in Israel has developed an image segmentation tool that segments a user interface [1] and extracts the graphical user interface components in the form of a GXL graph data structure. Visual Environment for Graph Grammar Induction and Engineering (VEGGIE) is a tool which supports graph parsing to analyze the graphical user interface [2]. Combining those two tools can efficiently analyze the semantics of a web interface. However, VEGGIE takes the GraphML format as input; the HP image segmentation tool produces the result in GXL format. To make these tools inter-operable, we need to convert data from the GXL format to the GraphML format. It is tedious and error prone to manually convert the data. In this paper, a tool has been developed to efficiently convert graph data from GXL to GraphML.

GXL and GraphML both follow the XML syntax and are capable of representing all types of graph formats [3, 4, 5], but the graph structure representation is different from one to the other. Different techniques have been proposed to convert one XML format to the other XML format. One common technique is based on programming languages. With the programming-language technique, the input XML document will be mapped to a programming class. Next, this

class is converted to a class that resembles output XML. Finally, this output class is mapped to an output XML document. This two-step process takes more time and programming efforts. Another technique is based on eXtensible Stylesheet Language Transformations (XSLT), a widely accepted XML transformation language. XSLT is declarative language and is, therefore, more resilient to changes than traditional programming languages [6]. The XSLT style sheet contains transformation rules to convert an input document to an output XML document; therefore, it is easier to understand than traditional programming languages. Due to the advantages of XSLT, this paper uses the XSLT processor for converting an XML document from GXL to GraphML.

To convert graph data from GXL to GraphML using XSLT, we have to address some challenging issues. First, we have to analyze two graph representations and summarize their syntactical differences. Second, we need to develop a mapping style sheet that represents these syntactical differences. Third, we convert GXL to GraphML using the XSLT style sheet. Finally, a user-friendly interface has to be developed.

A detailed comparison has been made between the two languages, and a list of structural and syntactical differences has been summarized. An XSLT style sheet is created based on this list. Because the Java language has the best support for XSLT and User Interface (UI) design, a Java-based Xerces XSLT processor with the Java Swing environment is used to design the conversion tool.

The Graph Data Conversion tool provides the functions of cut, copy, paste, redo, and undo for both the input document and the output document. This tool is generic. When a style sheet is changed, this tool can be applied to convert other XML formats.

1.2. Tree Visualization

"A picture is worth ten thousand words" as said by Frederick R. Barnard [7]. Today's web user interfaces become complex because they are providing more information to the user. Analyzing the structural and syntax relationship for the elements of a complex webpage is not easy. VEGGIE provides a mechanism for analyzing the structural and syntactical relationship of the elements on a complex webpage. VEGGIE tree visualization is limited to only JTree View, and also due to the recursive nature of the graph grammars used in the VEGGIE system, there will be some repeating child and parent nodes in the generated tree. The existing tree view has the following limitations.

1. Due to the recursive nature of graph grammars, there are some repeated child and parent nodes in the tree.
2. All nodes may not be displayed within one screen. Users have to expand the display to view children.
3. Tree view is not interactive.
4. You cannot save the tree view to an external disk for future reference.

Due to the limitations of tree view, an in-depth analysis of a graphical user interface is not easy. For an in-depth and easy analysis of the graphical user interfaces, more user-friendly and interactive visualization tools are needed.

Various visualization tools exist for visualizing large and complex network graphs, each having its own advantages and disadvantages. This paper uses one of the existing toolkits and integrates it into VEGGIE. The advantage of using an existing toolkit instead of designing a tool from scratch is that existing tools have been widely used and accepted in the community for many years and have been proven to be efficient. Prefuse [8], Java Universal Network/Graph

Framework (JUNG) [9], Piccolo [10], and Graphviz [11] are a few of the visualization tools that will perform tree visualization. Picolo and Prefuse generates better visualization for a networked graph. Picolo and Prefuse both provide similar visualizations, but due to Picolo's design, it is not easy to extend and integrate it into VEGGIE, whereas the Prefuse library is easily extensible and inferable into VEGGIE. GraphView and Treeview in Prefuse provide the visualization for the tree, and Prefuse also has zooming, panning, and dragging capabilities, which make it more interactive for users. In the future, we can add more visualization features to this implementation. Due to these features, this paper has chosen Prefuse for visualization.

There are a few challenges to extend Prefuse functionality into VEGGIE. The first one is identifying and using Prefuse methods in VEGGIE, and identifying Prefuse inputs to extend Prefuse functionality. The second one is converting VEGGIE tree data into Prefuse's data structures. The last one is making the generated Prefuse views more interactive.

The approach this paper has taken to solve the above-noted challenges is to analyze the Prefuse library and to identify methods and inputs. VEGGIE is updated to generate TreeML [12], GraphML, and HTML data structures. Tree View and Graph View is generated using Prefuse library, user interactive elements such as zooming and panning are added.

The additional views for the VEGGIE parser give more user interactivity and provide an easy way to analyze the user-interface structure. For future reference, tree data can be saved to a local machine. The remainder of the paper is organized as follows. Chapter 2 provides a complete Background of related topics for this paper. Chapter 3 gives the overall architecture of the application. Chapter 4 provides design, implementation, and results for the graph data conversion tool. Chapter 5 provides design, implementation, and results for the tree visualization tool. Chapter 6 discusses the advantages and disadvantages in this paper and future work.

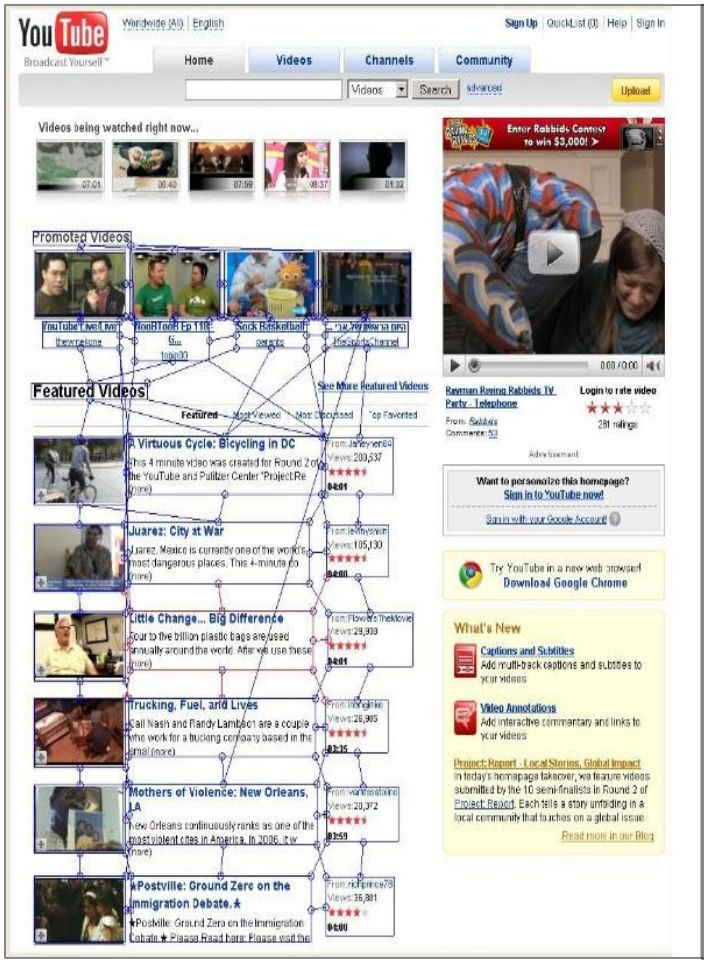
CHAPTER 2. BACKGROUND

This chapter provides information about HPLI's page segmentation tool, VEGGIE, GXL, GraphML, XSLT, and Prefuse. The HPLI's page segmentation tool extracts user interface semantics from an image of a webpage. HP pahe segmentation tool captures interface semantics in GXL graph data format. VEGGIE analyzes the user interface semantics. VEGGIE accepts user interface graph data in GraphML format. Prefuse is a java based visualization API, which provides interactive visualizations.

2.1. HPLI's Page Segmentation Tool

The HP Labs' page segmentation tool is based on computer vision methods to identify atomic visual elements from the image [1]. This tool uses segmentation and object recognition techniques [1]. The segmentation technique divides the graphical user interface into smaller segments to extract the interface's visual element semantics. The object recognition technique analyzes interface semantics. The tool takes a screen shot of a user interface as input and generates a fully classified, hierarchical graph in a graph data format [1].

Figure 1 shows an image of the YouTube segmented user interface image and sample output GXL graph data. The segmentation process divides the YouTube user interface image into multiple, small segments (top, down, bottom, etc.), and the object recognition process analyzes each small segment, and it recognizes images, texts, and the relationships. These images, texts, and the relationships among them are captured in GXL graph data format. Visual elements in a graphical user interface are captured as nodes and the relationship captured as an edge in the GXL.

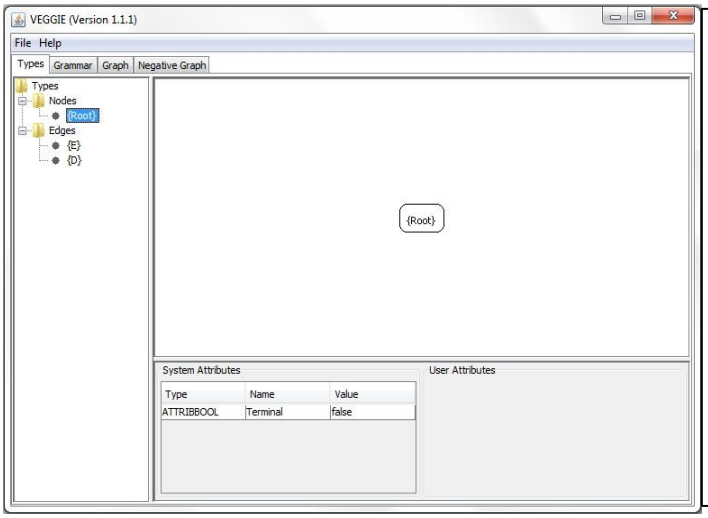



```
<node id="text58">
        <attr name="type">
                <string>text</string>
         </attr>
         <attr name="layout">
                <seq>
                    <int>428</int>    <int>344</int>
                     <int>540</int>   <int>374</int>
                </seq>
          </attr>
   </node>
 <edge from="text57" to="text58“ id="edge57-58" >
                <attr name="direction">
                        <string>E</string>
                </attr>
                <attr name="distance">
                        <string>26</string>
                 </attr>
   </edge>
```

Figure 1. You Tube Segmented Image and Sample Output GXL [1].

2.2. VEGGIE Tool

Visual Environment for Graph Grammar Induction and Engineering (VEGGIE) is a unified graph grammar construction, parsing, and inference tool [2]. VEGGIE uses GraphML as the standard formalism to represent graphs. Figure 2 shows the VEGGIE user interface and GraphML input. VEGGIE's user interface provides three types of editors: Node Type editor, Graph editor, and Grammar editor. The Node Type editor supports node types and edge types. The Graph editor contains the input graph data. The Grammar editor contains grammar production rules.




```
<node id="58" type="text" pos="428
344">
</node>

……

  <edge type="{E}" source="57"
target="58" directed="false"/>
  ……
```

Figure 2. VEGGIE Tool and Sample Input GraphML [2].

2.3. GXL

Graph eXchange Language (GXL) is designed to be a standard language to exchange graphs across different graph-based tools [4]. GXL is an XML-based language which can be extensible and customizable for unique applications. GXL was proposed to exchange graphs between software reengineering tools, and it included ideas from common exchange formats used in the reengineering. GXL can be used to describe typed, attributed, directed, and undirected graphs. GXL uses these basic graph formats to handle additional graphs such as hierarchical graphs and hyper graphs.

The GXL document below observes the XML syntax. All GXL documents share a name space, and they should contain a header element, i.e., <xml>, and a root element, i.e., <gxl>. Under the root element, there can be any number of graph elements, i.e., <graph>. Each graph element represents a single graph structure which is made of nodes, edges, and attributes. A node

is a point or object in the graph, and it represents a visual element. An edge represents a relationship between nodes.

```
<?xml version=”1.0” ?>
<!DOCTYPE gxl SYSTEM “gxl1.0.dtd” >
<gxl>
   <graph id=”example1”>
                <attr name = “edgemode”> <string> undirected </string> </attr>
                <node id = “A”>
                        <attr name = “label”> <string>First Base</string> </attr>
                </node>
                <node id = “B”>
                        <attr name = “label”> <string>Second Base</string> </attr>
                </node>
                <edge from = “A” to = “B” isdirected = “false”/>
    </graph>
    <graph id=”example2”>
    </graph>
</gxl>
```

2.4. GraphML

Similar to GXL, GraphML is also an XML-based representation of graph structures. The GraphML file format resulted from the joint effort of the Graph Drawing Steering Committee to define a common format for exchanging graph structure data [5]. GraphML is designed, in particular, for data transfer between graph drawing tools and other applications. It consists of a language core to describe the structural properties of a graph and a flexible extension mechanism to add application-specific data, such as layout and graphic information. In particular, such extensions can be freely combined or stripped without affecting the graph structure itself. Thus, GraphML provides a simple format for adding application-specific drawing information in a well-defined way. GraphML supports all the possible graph structures, including typed, attributed, directed, undirected, nested, and hyper graphs.

```
<?xml version=”1.0” encoding="UTF-8”?>
<graphml>
   <graph id="G" edgedefault="undirected">
        <key id="e0" for="edge" attr.name="distance" attr.type="int">
             <default>10</default>
        </key>
        <node id="n0""></node>
        <node id="n1"/>
        <edge id="e1" source="n0" target="n1" directed="true"/>
        <edge id="e2" source="n0" target="n2">
             <data key="e0">445</data>
        </edge>
    </graph>
</graphml>
```

The GraphML above observes XML syntax. All GraphML documents should have a root element, i.e., <graphml>. Similar to GXL, GraphML contains any number of graph elements, i.e., <graph>.

2.5. XSLT

XSLT is a language to transform XML documents. It is a functional language based on recursive pattern matching, value substitution, and template substitution. The XSLT style sheet contains a collection of template rules. Style sheets are convenient to use, portable, and easy to customize [6]. There are three essential components in an XSLT transformation: an XML document, an XSLT style sheet, and an XSLT processor. The XSLT processor applies template rules on an input XML document instance to produce a new XML document or HTML document.

```
<?xml version=”1.0” standalone="no”?>
  <xsl:stylesheet version="1.0"   xmlns:xsl="http://www.w3.org/1999/XSL/Transform”>

            <xsl:template match="paragraph">
                <p>
                <h1> I am starting new paragraph </h1>
                </p>
            </xsl:template>

</xsl:stylesheet>
```

The above document is an example XSLT style sheet. The XSLT style sheet is an XML document, and begins with the XML declaration, i.e., <?xml version="1.0" standalone="no"?>. An XSLT style sheet contains template <xsl:template> elements to select matching nodes from an input XML document. The match attribute associates the template with an element of the input XML.

2.6. Prefuse

Prefuse is an information visualization toolkit [8]. It supports interactive information visualization applications and visualizes tables, graphs, and trees. Prefuse is developed in the Java language, and allows users to adapt and extend its components through sub-classing and substitution. This paper incorporates Prefuse Tree View and Graph View into VEGGIE.

Prefuse Tree View is fully interactive and supports some advanced features, such as zooming, searching, and Degree-of-Interest (DOI). The DOI layout and scaling algorithm produces a display of the entire tree in the fixed-size area. When the user clicks on a node, it becomes the focus of interest, and the rest of the tree grows or shrinks, as appropriate, to display the viewport of the tree. Additional details for each node are available by using hypertext links when the node is in focus.

Graph view is a re-implementation of Yee implementation [8]. Yee implementation is an animated visualization of radial graphs. Complex graphical user interfaces can have thousands of nodes with edges between them. It is desirable to see all the nodes in a single view. Graph view is capable of visualizing a large number of nodes in a single view and provides an interactive exploration of the graph's sub-regions. In this view, the graph nodes are scattered all over the display and connected with arcs. This view provides an effective way of exploring large graphs by focusing on selected nodes.

CHAPTER 3. SYSTEM ARCHITECTURE

This chapter discusses the system's architecture. There are two modules in the system: Graph Data Conversion and Tree Visualization. Figure 3 shows the architecture of the system.

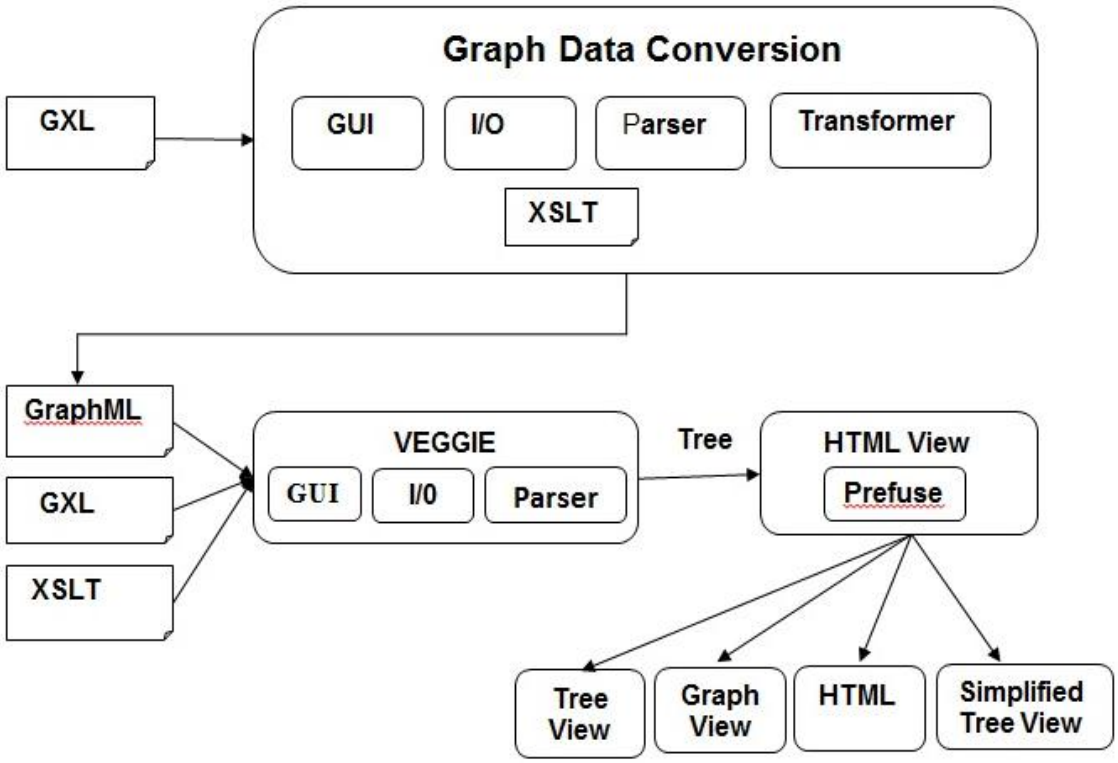


Figure 3. System Architecture.

The Graph Data Conversion takes a GXL file as input and produces a GraphML output. The Graph Data Conversion sub-system has four modules: Graphucal User Interface (GUI), Input/Output (I/O), Parser, and Transformer. The GUI module provides an option for the user to open, edit, save, and transform a GXL document. The I/O module reads the input GXL file, and Parser validates the syntax; upon successful validation, the transformer applies transformation rules to produce output GraphML. More details about XSLT and the conversion process will be

discussed in the System Implementation section. The tool's user interface is developed using Java Swing. GraphML generation is handled by a set of service methods utilizing a Java XML parser and an XSLT style sheet.

The generated GraphML file from the Graph Data Converter is given as an input to VEGGIE. Along with the GraphML input, VEGGIE requires a type file and a grammar file. VEGGIE generates three different types of views: Tree View, Graph View, and Simplified Tree View.

The tree visualization module generates different types of views from the parsing tree generated by VEGGIE. Visualization provides four additional views: Graph View, Tree View, HTML view, and Simplified Tree View. The Prefuse library is extended into VEGGIE to generate Graph View and Tree View. HTML view can be viewed on a web browser. Simplified Tree View shortens the existing tree view by removing repeated nodes. The individual module designs and implementations will be discussed in the next chapters.

CHAPTER 4. GRAPH DATA CONVERTER

This chapter discusses the design and implementation of the Graph Data Converter. The Graph Data Converter converts the GXL graph data into GraphML graph data. Figure 4 use case diagram illustrates the requirements.

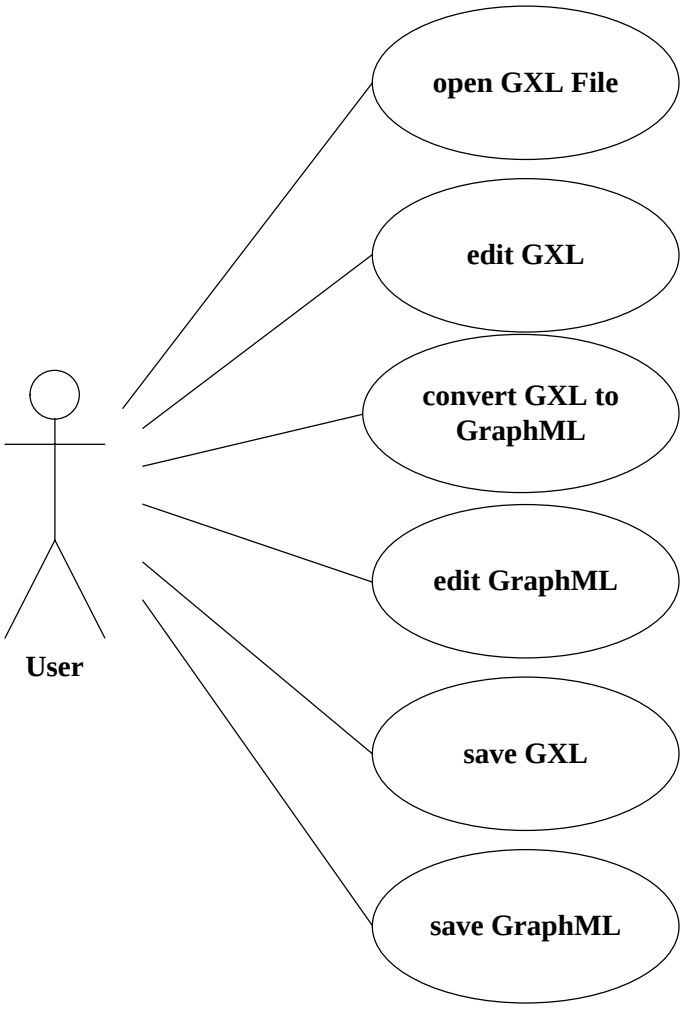


Figure 4. Graph Data Converter: Use Case Diagram.

The Graph Data Converter GUI provides the following functions:

1. Open GXL
2. Edit GXL
3. Save GXL
4. Transform GXL to GraphML
5. Edit GraphML
6. Save GraphML

The first use case ("open GXL") represents the requirement to select and open a GXL file from the local machine. The second and third use cases consist of requirements to edit the GXL file in the user interface and to save the GXL file to a local machine. These features are useful when there are inconsistencies in the input GXL. The fourth use case represents the transformation of GXL to GraphML. The fifth and sixth use cases specify requirements for editing and saving the generated GraphML file.

4.1. Design and Architecture

The application's system requirements are modeled with the sequence diagram shown in Figure 5. The sequence diagram shows all objects and the sequence of messages between objects to complete the requirements. The UI serves as the front end to the application, and all user interactions are handled by the UI.

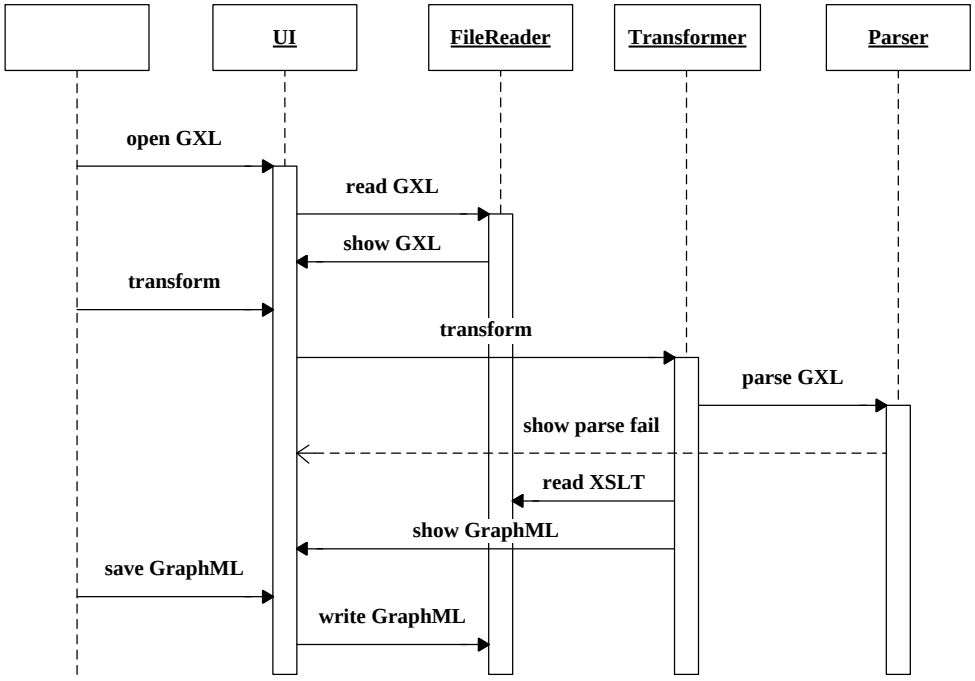


Figure 5. Graph Data Converter: Sequence Diagram.

There are four objects in this sequence diagram: UI, File Reader, Transformer, and Parser. First, the user opens the GXL file from the UI. The UI sends a message to the file reader to read the file from the system. The UI opens the file and shows it for editing. Users can edit the file in the UI. The transform menu selection in the UI sends a message to the transformer object to transform the GXL document. The Transformer sends a message to the parser object to validate and parse the GXL document. If parsing fails, the parser returns an error message to the UI. If parsing is a success, the Transformer sends a message to the File Reader to read XSLT. Based on the XSLT mappings defined in the style sheet, the transformer converts the GXL structure of the graph to GraphML structure. The generated GraphML graph will be displayed in the UI. Users can edit or save GraphML data.

4.2. System Implementation

The Graph Data Converter's graphical user interface (GUI) is developed using the Java Swing library, and the backend is implemented using Java language. Eclipse IDE is used for the tool development.

Figure 6 shows the class diagram of the Graph Data Converter. The class diagram lists the classes, attributes, and operations in the system.

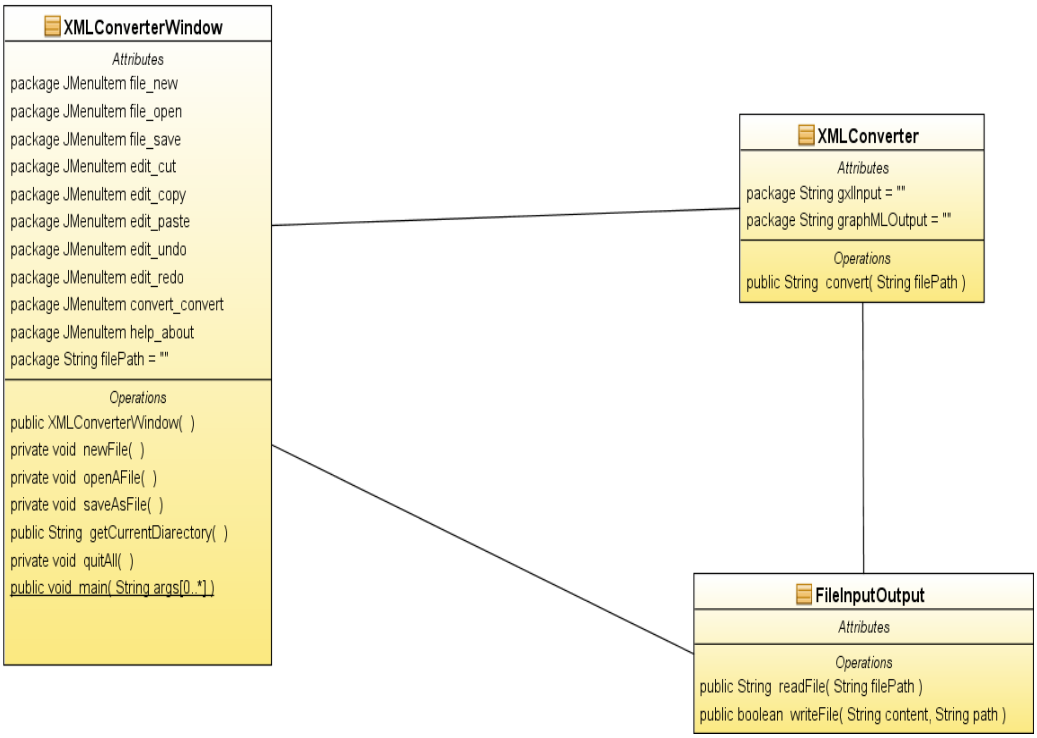


Figure 6. Graph Data Converter: Class Diagram.

The class diagram has three main classes: XML converter window, XML converter, and File Input Output. The XML converter window class creates a user interface and provides options: open GXL, edit GXL, convert GXL to GraphML, and save GXL or GraphML. The File

Input Output class reads files from the local machine or writes files to the local machine. The XML converter class converts a GXL document to a GraphML document using an XSLT processor.

4.2.1. User Interface

Figure 7 shows the user interface of the Graph Data Converter. The user interface consists of three tabs implemented using the tabbed pane layout of Java Swings.

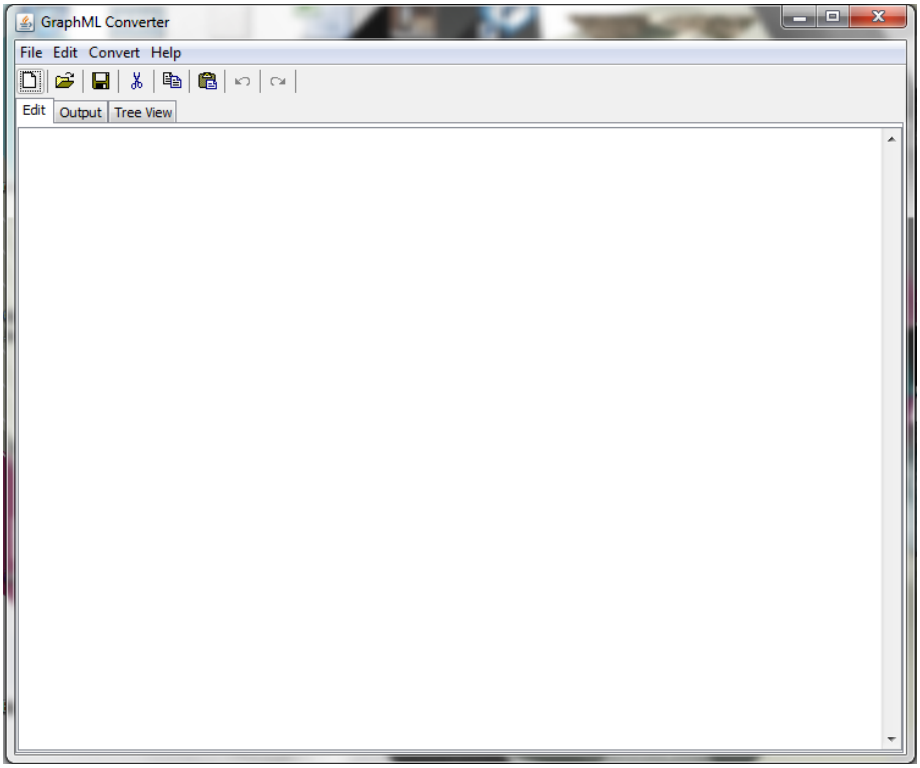


Figure 7. Graph Data Converter: User Interface.

4.2.1.1. Tabbed Pane Layout

Tabbed pane layout consists of three tabs. The first tab displays the input GXL data, and the second tab displays the output GraphML. These two tabs contain rich text editors to edit the data. The third tab displays the DOM structure of the GraphML data.

4.2.1.2. Menu and Tool Box

The graphical user interface provides four menus: File, Edit, Convert, and Help. The File menu has Open, Save, and Exit menu items. The Edit menu has Cut, Copy, Paste, Redo, and Undo menu items. The Convert menu has a GXL to GraphML Convert menu item. The Help menu has an About menu item. The user interface tool box provides Open, Save, Cut, Copy, Paste, Redo, and Undo menu items.

4.2.2. XSLT Style Sheet

The XSLT style sheet contains rules to transform GXL to GraphML. Each transformation rule has two parts: a pattern which is matched against nodes in the GXL and a template which can be instantiated to form part of the resulting GraphML. The transformation rules are developed by comparing GXL and GraphML. The key differences between GXL and GraphML are discussed as follows.

1. **Header:** The GXL document starts with <gxl>, whereas the GraphML document starts with <graphml>. Both can have an optional name space attribute.
2. **Graph:** With both GXL and GraphML, the graph element is declared with a <graph> tag. A GXL graph element contains an edge mode attribute, whereas a GraphML graph contains an edge default attribute. An edge mode value can be directed, undirected,

default directed, or default undirected. An edge default attribute value can be directed or undirected.

3. **Node:** A GraphML node may contain port information, but GXL does not support port information.
4. **Edge:** An edge in GXL is represented with from and to attributes, whereas an edge in GraphML is represented with source and target attributes. Both GXL and GraphML edge attributes refer to nodes using an identifier. GXL edge incidences can be ordered, but GraphML will have element <port>.
5. **Attributes:** A GXL attributed element is declared with <attr>, whereas a GraphML attribute is declared with key and data elements. A key element defines the type and scope of an attribute. The data element specifies an actual occurrence of the element.
6. **Port:** The ordering of incidences in a GXL edge is specified using fromorder and toorder attributes. In GraphML, the ordering of incidences is specified using a port. The GraphML port ordinal number specifies the ordering attribute of a GXL edge.
7. **Nested Graphs:** Nested graphs will contain a graph inside another graph. Nested graphs in GXL and GraphML have no differences.
8. **Hyper Graphs:** GXL hyper graphs are declared with a rel element, and end points of a hyper graph are declared with a relend element. GraphML hyper graphs are declared with a hyper graph element, and endpoints are declared with an endpoint element.

The XSLT style sheet is developed based on the above comparisons. The GXL file starts with a <gxl> header element, whereas a GraphML file starts with <graphml>. Therefore, the style sheet replaces <gxl> with <graphml>.

```
<gxl xmlns:xlink="http://www.w3.org/1999/xlink">
    ……
</gxl>
```

The document above shows an example GXL start element, i.e., <gxl>. The style sheet for transforming the GXL element to the corresponding GraphML element contains one xsl:template element with match and body. The match rule is declared with a string to match the gxl element, and the body contains content to generate output. The following document shows the XSLT template to convert a GXL root element to a GraphML root element.

```
<xsl:template match="gxl">

<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
<xsl:apply-templates/>
</graphml>

</xsl:template>
```

The following document shows the generated GraphML output. GraphML starts with <graphml> element. It has a name space element "http://graphml.graphdrawing.org/xmlns”.

```
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
……
</graphml>
```

XSLT transformation mappings for the <graph>, <node>, and <edge> elements are similar to the mapping for the root element. A recursive pattern-matching process for child elements of a root element is initiated by placing <xsl:apply-templates/> inside the parent <xsl:template> template. The document below shows an example XSLT template that converts a GXL node to the corresponding GraphML node. GXL node id is alpha-numeric whereas

GraphML node is numeric, so node id will be converted to numeric. Type and position attributes will be converted from GXL to GraphML.

```
<xsl:template match="node">

<xsl:variable name="character"
select="'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_$'" />
<xsl:variable name="id">
<xsl:value-of select='translate(@id,$character,"")'/>

<node id="{$id}">

<xsl:attribute name="type">
<xsl:if test="attr[@name='type']">
    <xsl:value-of select="attr[@name]/string"/>
</xsl:if>

<xsl:attribute name="pos">
   <xsl:value-of select="concat(attr/seq/int[1],' ',attr/seq/int[2])"/>
</xsl:attribute>

<data key="attrib">
    <attrib id="Terminal" type="2" bool="true"/>
</data>

<xsl:apply-templates/>

</node>

</xsl:template>
```

The following XSLT template converts a GXL edge to the corresponding GraphML edge. The XSLT template match rule matches GXL edge element, and code inside the template generates the GraphML edge. The “<xsl:template>” element will match with the GXL edge element with GraphML edge. Just like node id, edge id is numeric in GraphML, so edge id will be converted from alpha-numeric to numeric. GXL to and from attributes will be converted to source and target attributes.

```
<xsl:template match="edge">

<xsl:variable name="character"
select="'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_$'" />

<xsl:if test="(attr[@name='topology'] and attr[@name]/string !='disjoint') or
((attr[@name='topology'] and attr[@name]/string='disjoint') and (attr[@name='direction']
and attr[@name]/string ='E' or attr[@name]/string ='N'))">
<xsl:if test="(attr[@name='distance'] and attr[@name]/string &lt;= //@distance)">

<edge>

<xsl:attribute name="source">
<xsl:value-of select='translate(@from,$character,"")'/>
</xsl:if>

<xsl:attribute name="target">
   <xsl:value-of select= 'translate(@to,$character,"")'/>
</xsl:attribute>

<xsl:attribute name="type">
   {E}
</xsl:attribute>

<xsl:attribute name="directed">

            <xsl:choose>
                <xsl:when test="contains(@edgemode,un)">false</xsl:when>
                <xsl:otherwise>true</xsl:otherwise>
            </xsl:choose>

</xsl:attribute>

<xsl:apply-templates/>

</edge>
</xsl:if>
</xsl:if>
</xsl:template>
```

4.2.3. Code Explanation

The core of the converter is the transformer. The transformer changes the GXL document into the GraphML document by applying XSLT transformation rules. The transformer makes use of an XSLT engine to process the GXL document. The XSLT engine transforms the GXL document to a tree structure and applies the transformation rules on the tree structure to produce an output tree structure. The output tree structure is converted to GraphML data. XSLT transformer libraries are not part of the Java Framework standard distribution. An open-source Apache Xalan-J library xerces.jar is used in the tool.

The FileReader class reads a file from the system and writes the file to the system. The File open command in the GUI executes the following code for the file reader to read a file from the system. The File Reader reads the file, and the GUI displays the file content in the Editor Panel.

```
String input = “”;
StringBuilder body = new StringBuilder();

FileInputStream fin = new FileInputStream(filePath);;
BufferedReader myInput = new BufferedReader(new InputStreamReader(fin));

while ((thisLine = myInput.readLine()) != null){
body=body.append(thisLine).append("\n");
}
Input = body.toString();
```

The code above shows the File Reader operation. A file path is given as input to the reader. The reader opens the file content using the FileInputStream object. InputStreamReader and BufferedReader read the content which can be stored in the memory.

The “in memory” file content is available in the tool for further processing. The parse option selection in the user interface applies XSLT transformation rules on the file content and produces output. The TransformerFactory class of the Xalan-J library applies the XSLT transformation rules on the file content. To apply the TransformerFactory transformation rules, an object of the TransformerFactory class is created. Next, the transformer object is created from the TransformerFactory object for the XSLT style sheet.

```
TransformerFactory tFactory = TransformerFactory.newInstance();

Source xsltSource = new StreamSource("./src/xsltfile/converter.1.xsl");

Transformer transformer = tFactory.newTransformer(xsltSource);
```

The first line of the code above creates a TransformerFactory object. The next line reads the XSLT file. The third line creates a Transformer object for the given XSLT style sheet. A GXL document is converted to GraphML by calling the transform method of the Transformer object with the GXL source and result object. The following code shows the transformation of the input GXL content string to the output GraphML string and displays the GraphML in the GUI.

```
// transform inputXML using transformer – save output to outputXML
tranformer.transform(inputXML, outputXML);

//convert Result to String
String outputString = xmlOutput.getWriter().toString();

// get GUI output textarea
JTextArea output = new JTextArea("", 30, 100);

// write output GraphML String to Output Editor
output.setText(outputString);
```

4.3. Results

```
<gxl xmlns:xlink="http://www.w3.org/1999/xlink">
<?xml-stylesheet type="text/xsl" href="converter.xsl"?>

 <graph id="youtube.xml.bmp">
  <node id="text58">
   <attr name="type">
    <string>text</string>
   </attr>
   <attr name="layout">
    <seq>
     <int>428</int>
     <int>344</int>
     <int>540</int>
     <int>374</int>
    </seq>
   </attr>
  </node>
  <node id="text57">
   <attr name="type">
    <string>text</string>
   </attr>
   <attr name="layout">
    <seq>
     <int>307</int>
     <int>344</int>
     <int>402</int>
     <int>374</int>
    </seq>
   </attr>
  </node>

  <edge from="text57" id="edge57-58" to="text58">
   <attr name="topology">
    <string>disjoint</string>
   </attr>
   <attr name="direction">
    <string>E</string>
   </attr>
   <attr name="distance">
    <string>26</string>
   </attr>
```

```
      <attr name="vertical alignment">
      <string>equal</string>
    </attr>
  </edge>
  <edge from="text58" id="edge58-57" to="text57">
    <attr name="topology">
      <string>disjoint</string>
    </attr>
    <attr name="direction">
      <string>W</string>
    </attr>
    <attr name="distance">
      <string>26</string>
    </attr>
    <attr name="vertical alignment">
      <string>equal</string>
    </attr>
    </edge>
  ……
</graph>

</gxl>
```

The above document shows the YouTube user interface graph data generated using the HP page segmentation tool. The document contains one graph with the "youtube.xml.bmp" ID. The actual YouTube graph contains many nodes and edges, but for simplicity, this document contains two nodes and two edges. The first node is "text58," and the second node is "text57." Nodes have attributes to define type and layout. The first edge is "edge57-58," and the second edge is "edge58-57.. Both edges have attributes to define topology, direction, distance, size, alignment, etc. Figure 8 shows the Graph Data Converter tool. The tools editor panel shows the input GXL file content.

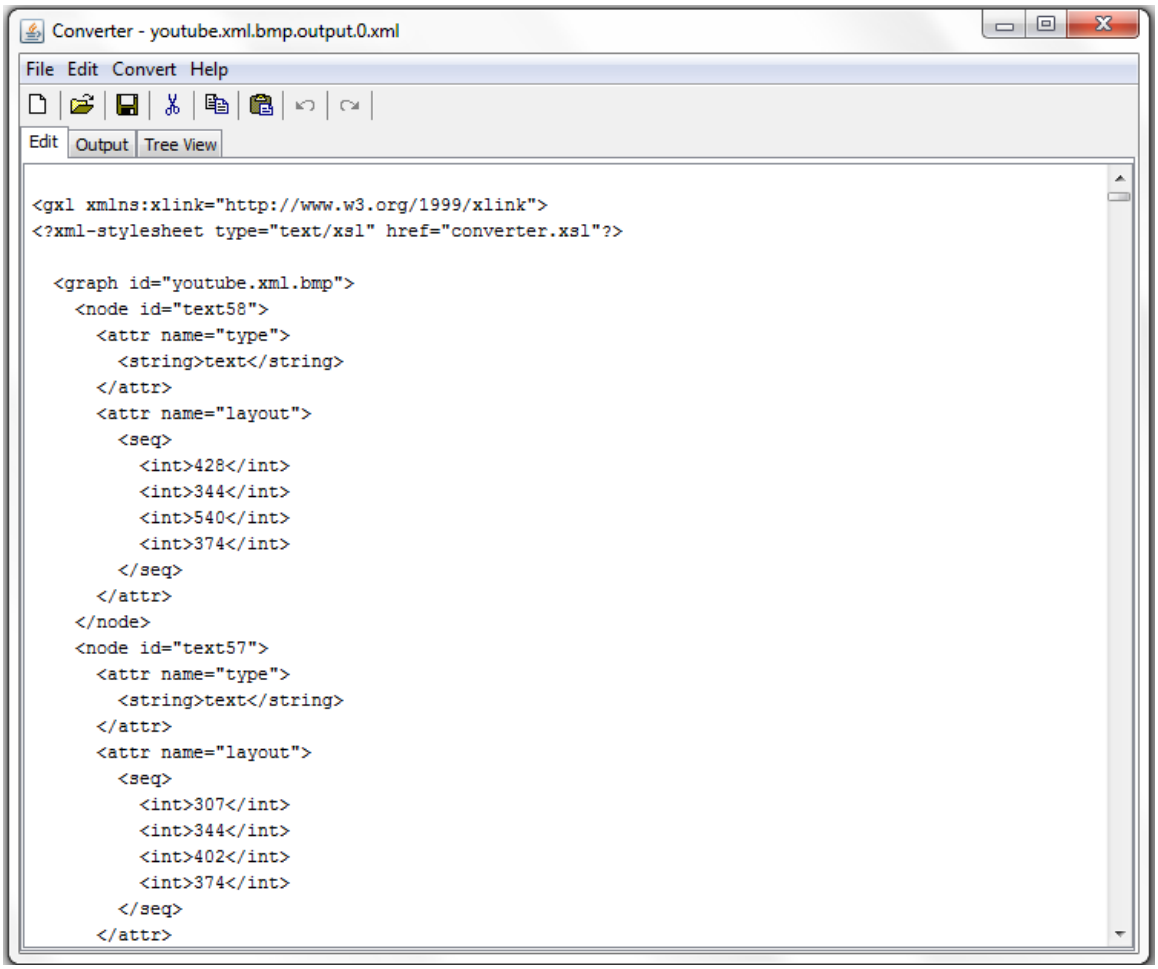


Figure 8. Graph Data Converter: GXL Input.

The following document shows the generated GraphML graph corresponding to the example YouTube GXL input. The root element is "<graphml>," and it contains one graph. The graph contains two nodes, "57" and "58," and two edges. Each node defines the type and position of a visual element, and contains an attribute. The attribute is declared with a data element. The edge specifies the relationship between the nodes.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- This GraphML document was generated from GXL by
a GXL-to-GraphML conversion style sheet. -->

<graphml xmlns="http://graphml.graphdrawing.org/xmlns">

<graph xmlns="http://viscomp.utdallas.edu/VEGGIE" id="youtube.xml.bmp"
edgedefault="undirected">

    <node id="58" type="text" pos="428 344">
            <data key="attrib">
            <attrib id="Terminal" type="2" bool="true"/>
            </data>
   </node>

   <node id="57" type="text" pos="307 344">
            <data key="attrib">
            <attrib id="Terminal" type="2" bool="true"/>
            </data>
  </node>

  ……

  <edge type="{E}" source="57" target="58" directed="false"/>
  <edge type="{E}" source="58" target="57" directed="false"/>

  ……

</graph>

</graphml>
```

Figure 9 shows the tool output panel with generated GraphML data. A user can edit the GraphML or save GraphML to a file on a local machine.

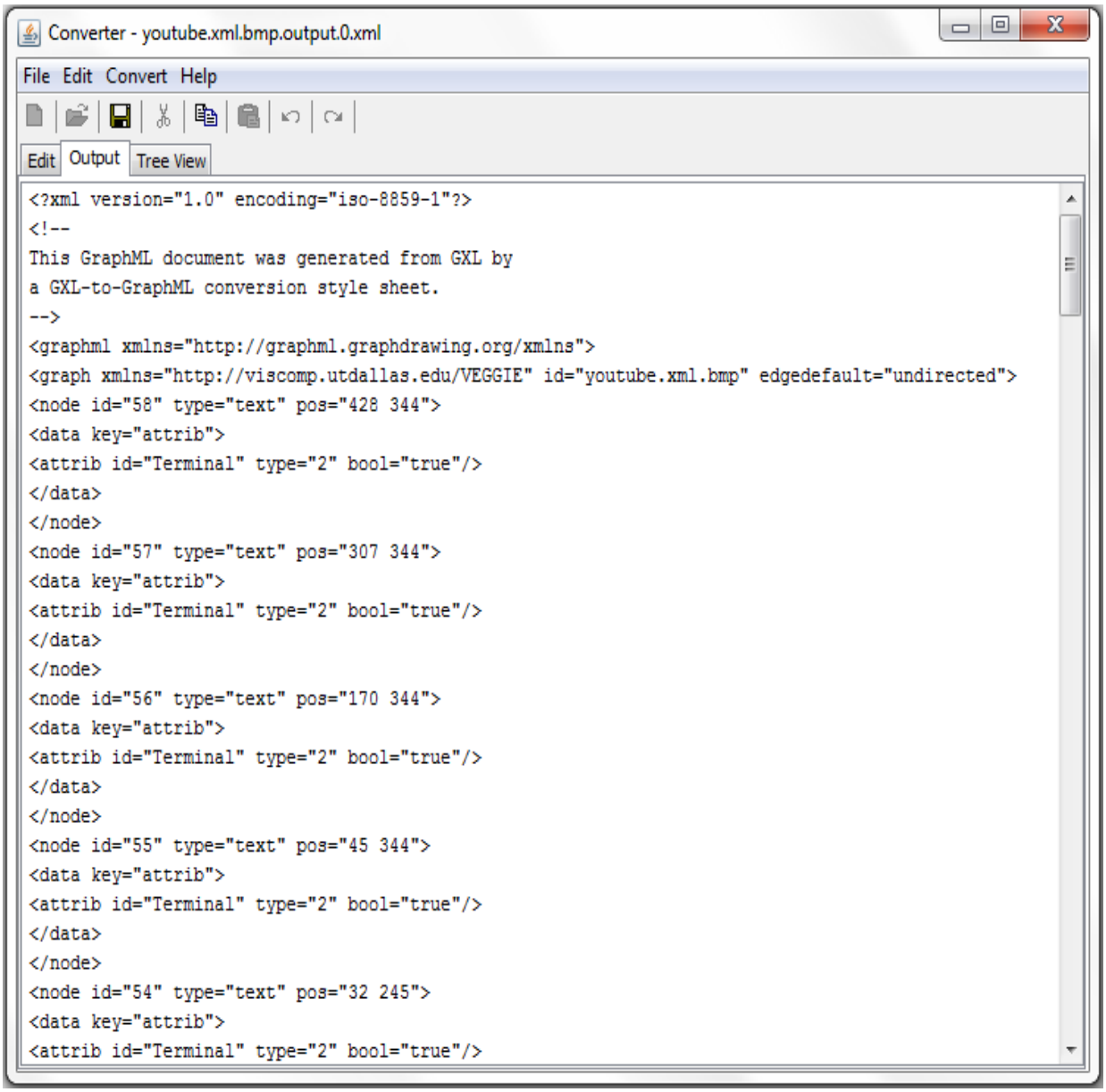


Figure 9. Graph Data Converter: GraphML Output.

CHAPTER 5. TREE VISUALIZATION

Tree visualization generates Tree View, Graph View, HTML View, and Simplified Tree View for GraphML data. VEGGIE only supports one view, i.e., JTree View. Tree Visualization extends the VEGGIE visualization capability by adding four additional views. Options to generate additional views are added to the VEGGIE GUI. Selecting any of these view options displays the generated view in a new window. Figure 10 shows the architecture of the tree visualization.

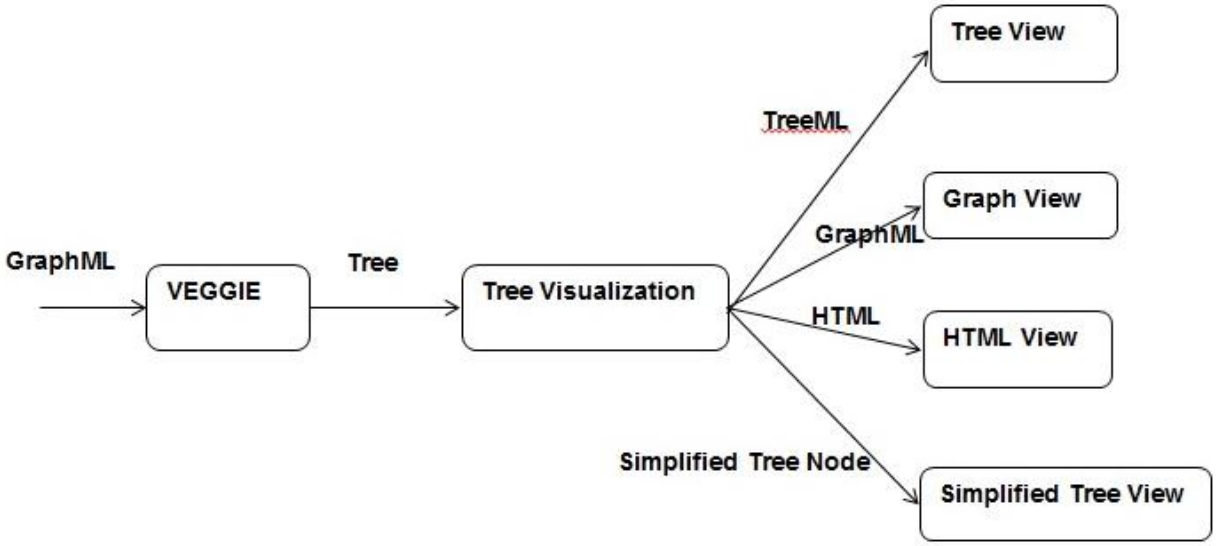


Figure 10. Tree Visualization Architecture.

VEGGIE provides the functions to open and read the type file, grammar file, and graph file. Additionally, it provides functions to parse a graph file and generate a corresponding tree. The VEGGIE parser requires the type file and grammar file to parse a graph file. The type file

defines all nodes and edges. The grammar file contains the production rules. The graph file contains the GraphML data. VEGGIE parses the GraphML and generates a tree by using the production rules defined in the grammar file. The generated tree will be converted to TreeML, GraphML, HTML, or Simplified Tree. The Simplified Tree is the VEGGIE-generated tree without duplicate nodes. Figure 11 shows the use case diagram for the tree visualization.

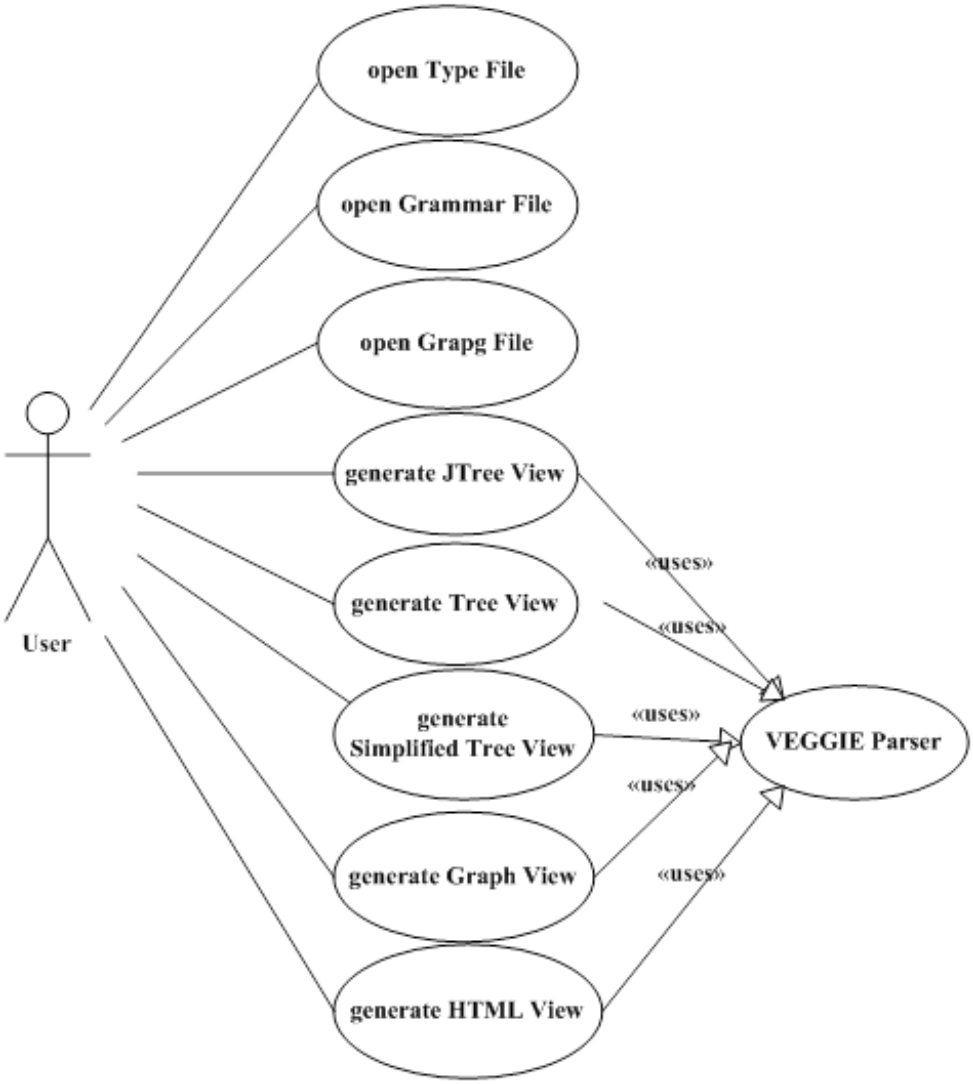


Figure 11. Tree Views: Use Case Diagram.

The above use case diagram lists all the operations. A user can open, edit, and save the grammar file, type file, and GraphML file. The “generate view” action parses and generates a view by using the Prefuse library. User options, such as zoom in, zoom out, resize, etc., are not depicted in the use case diagram.

5.1. Design and Architecture

Figure 12 shows the sequence diagram of the system. The sequence diagram illustrates the objects involved in the system and messages between the objects. There are four objects in the system: UI, Input/Output, Parser, and Prefuse library. Opening a type file, graph file, or grammar file sends a message to Input/Output to read the files. The Input/Output object reads these files and displays them in the UI. Triggering the Parse Menu item in the UI sends a message to the VEGGIE parser to parse the graph file. The Parser parses the graph file by applying production rules and sends a message to the UI to display the JTree View.

Triggering the TreeView menu item in the UI sends a message to the Parser to parse the graph file. The Parser parses the graph file, generates TreeML and HTML, and sends a message to the Input/Output object to save TreeML and HTML to a local machine. The Parser sends another message to the Prefuse library to generate a corresponding tree view. The Input/Output object saves the TreeML and HTML to a local machine. The Prefuse generates the Tree View and sends a message to the UI to display the view.

GraphView generation is similar to the TreeView generation. Triggering the GraphView menu item in the UI triggers message flows between the UI, Parser, Input/Output, and Prefuse to generate and display Graph View.

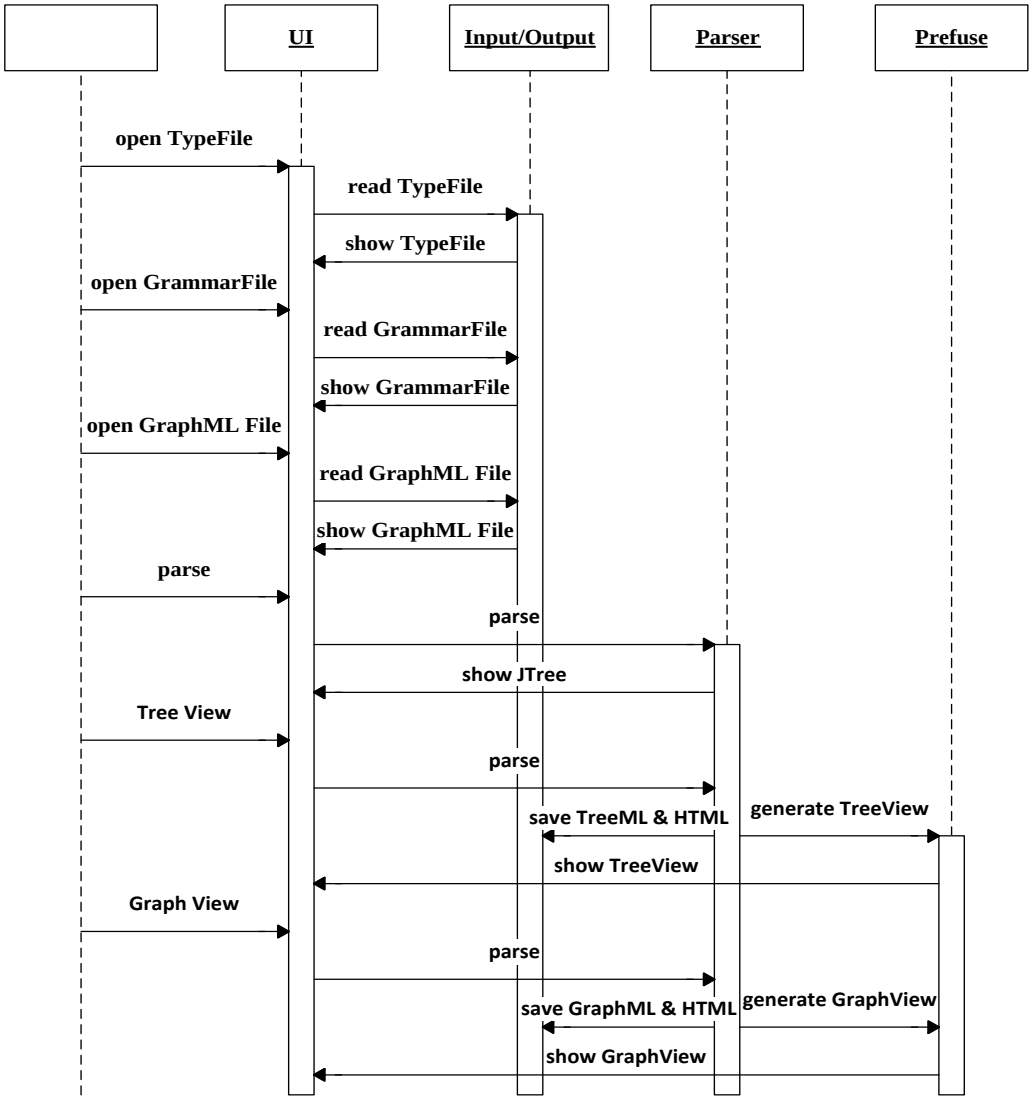


Figure 12. Tree Views: Sequence Diagram.

5.2. System Implementation

Figure 13 shows the class diagram. Each class lists the attributes and operations. There are five main classes in the system: SGGParseTreeFrame, SGGParser, FileInputOutput, TreeView, and GraphView. SGGParseTreeFrame and SGGParser are already present in VEGGIE.

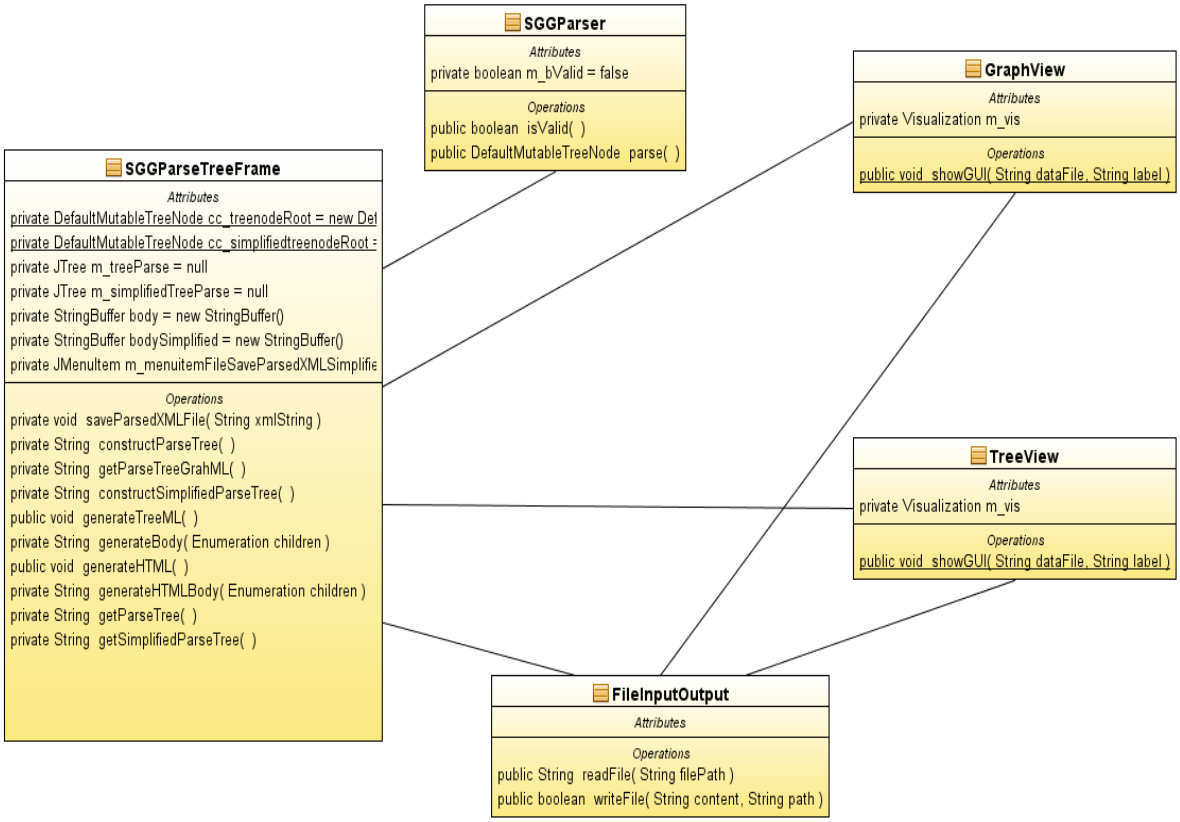


Figure 13. Tree Views: Class Diagram.

SGGParseTreeFrame displays the UI menu items and handles the user's selections. SGGParseTreeFrame is enhanced to support Tree View and Graph View. The SGGParser parses a graph and generates a DefaultMutableTreeNode Java object. The FileInputOutput class reads from a local machine and writes files to the local machine. The Graph/Tree View class extends Prefuse library functions to generate and display a Graph/Tree View. It also handles user actions with the Graph/Tree View, such as zoom in, zoom out, etc.

The Prefuse library simplifies the process of data handling, representation, mapping to an onscreen display, and crafting direct manipulation interactions with the visualization. Below is the pseudo code for creating rich visualizations.

```
1. Initialize Graph G
2. For all Nodes N G do
3.     Add Renderers(label, size, text, shape) to N
4. End for
5. For all Edges E G do
6.     Add Renderers(shape, label) to E
7. End for
8. Add Drag control to G
9. Add Zoom control to G
10. Add Pan Control to G
11. Add Layout to G
```

This pseudo code presents the steps to add user interactivity to the graph. Step 1 initializes the graph/tree from the user input abstract data. Steps 2 to 4 specify the visual properties of the graph/tree, such as color, shape, size, font, etc., for each node in the graph/tree. Steps 5 to 7 visualize each edge in the graph/tree. Step 8 adds the drag control to the nodes and edges. Step 9 adds the zoom control to the display. The zoom control allows zoom-in and zoom-out functions. Step 10 adds the pan control to the display. The pan control allows a user to move the display in a horizontal direction.

5.2.1. Prefuse Tree View

TreeView provides a rich user experience and easy analysis of the generated tree. Prefuse tree view provides interactivity options such as zooming, searching, DOI, etc. The Prefuse library is extended into VEGGIE to generate a tree view. To generate a tree view, the Prefuse library requires the input data to be of the TreeML format. TreeML contains nodes, edges, and links between the nodes.

VEGGIE parses the GraphML file and generates a tree structure Java object, DefaultMutableTreeNode. The DefaultMutableTreeNode Java object contains the tree structure with parent and child relationships. The enhanced VEGGIE parser iterates through this object

and generates TreeML. Each iteration generates a branch with the current node. The end of the recursive process generates a TreeML string data which can be written to a file. The following code snippet generates TreeML from the DefaultMutableTreeNode Java Object.

```
// Generate TreeML string body

private String generateBody(Enumeration children) {

            //create string buffer to append data
        StringBuffer tree = new StringBuffer();

        // iterate through children of each node
       while (children.hasMoreElements()) {

                // get child
          DefaultMutableTreeNode nodeCur = (DefaultMutableTreeNode)
children.nextElement();

            // open branch for this child node
        tree.append("<branch> \n");
        tree.append("\t<attribute name=\"name\" value=\""
                + nodeCur.toString() + "\"/> \n");

        // generate body for all its children
        tree.append(generateBody(nodeCur.children()));

            // close branch
        tree.append("</branch> \n");

    }

  // return body
  return tree.toString();
}
```

The TreeML file is a XML file with nodes and edges. To generate tree view, the Prefuse library requires the TreeML data to be represented as an internal tree data structure. Therefore, the user first loads the TreeML file into Prefuse's data structure, i.e., Tree. Next, the user creates

a Prefuse Visualization object and adds the tree data structure with a unique identifier. Many tree data structures can be added to the visualization.

```
// create new Tree from Input file
Tree t = new TreeMLReader().readGraph(datafile)
// create a new, empty visualization for our data
Visualization m_vis = new Visualization();

// set up visualization with Tree
m_vis.add(tree, t);

// intialize renderers
DefaultRendererFactory rf = new DefaultRendererFactory(m_nodeRenderer);
rf.add(new InGroupPredicate(treeEdges), m_edgeRenderer);

m_vis.setRendererFactory(rf);

// Add actionLists

// add quick repaint
ActionList repaint = new ActionList();
repaint.add(nodeColor);
repaint.add(new RepaintAction());
m_vis.putAction("repaint", repaint);

// add animated transition
ActionList animate = new ActionList(1000);
animate.setPacingFunction(new SlowInSlowOutPacer());
animate.add(autoPan);
animate.add(new QualityControlAnimator());
animate.add(new VisibilityAnimator(tree));
animate.add(new LocationAnimator(treeNodes));
animate.add(new ColorAnimator(treeNodes));
animate.add(new RepaintAction());
m_vis.putAction("animate", animate);
```

The visualization includes features such as spatial layout, color, size, and shape for each of the visual elements. These features are added to the visualization by creating action lists. This section explains adding two action lists. The first action list assigns colors to the nodes and

edges. The second action list adds an animated transition when the focus of the visualization changes. The following code snippet creates a tree data structure, creates visualization, adds the tree data structure to the visualization, and adds action lists.

The generated visualization form above step will be added to a display area. The display area contains an optional number of display controls. Some of the display controllers are Tree Depth Item Sorter, Focus Control, Drag Control, Pan Control, Zoom Control, and Neighborhood Highlight Controller. The Tree Depth Item Sorter separates items based on their depth in a tree. By default, an item deeper in the tree is given a lower score so that its parent node is drawn on top of child nodes. Focus Control focuses the selected node. Drag Control allows the repositioning of nodes, and initiating a recalculation of the layout and an animated transition. Pan Control changes the viewable region of the visualization. Zoom Control alters the display by changing the scale of the viewable region. Neighbor Highlight Control illustrates neighboring nodes for the node currently under the mouse pointer. The following code snippet shows how to add controls to the display area.

```
// initialize the display
Display display = new Display(m_vis);
display.setSize(700,600);

// display controls
display.setItemSorter(new TreeDepthItemSorter());
display.addControlListener(new ZoomToFitControl());
display.addControlListener(new ZoomControl());
display.addControlListener(new WheelZoomControl());
display.addControlListener(new PanControl());
display.addControlListener(new FocusControl());
```

The display area is viewable in the user interface after adding it to a JFrame object. The following code snippet shows the rendering of the display area in user interface.

```
// initialize enclosing window frame
JFrame frame = new JFrame("TreeView");
frame.getContentPane().add(display);
frame.pack(); frame.setVisible(true);

// run actionLists
m_vis.run("repaint");
m_vis.run("animate");
```

5.2.2. Prefuse Graph View

Prefuse Graph View layout is implemented based on the focus plus context technique to view the hierarchical data. Focus plus context presents the primary interest object in full detail along with an overview of the surrounding objects. Prefuse Graph View presents a large number of nodes in a single view by presenting the focused node in the center. Graph view also provides an interactive exploration of the graph's sub-regions.

First, load the generated GraphML file into Prefuse's graph data structure, i.e., Graph. Second, create a Prefuse Visualization object, and add the Graph to the Visualization. Last, action lists are added to improve the visualization. The following code snippet creates a graph data structure, creates visualization, and adds graph data structure to the visualization. Adding action lists to the Graph View visualization is same as adding action lists to Tree View visualization.

```
// Setup global list iterator on aanodeParseHistory
ListIterator<ArrayList<SGGSimpleNode>> iter_aNodes =
m_aanodeParseHistory.listIterator(m_aanodeParseHistory.size());

// while list has elements
while (iter_aNodes.hasPrevious()) {
         // Setup sub list iterator
        ListIterator<SGGSimpleNode> iterNodes = iter_aNodes.previous().listIterator();
 // while sublist has nodes
if (iterNodes.hasNext()) {

         // Get this parent node
         SGGSimpleNode nodeCurParent = iterNodes.next();
        // add parent node to graphml body
        body.append("<node id=\""+nodeCurParent.getID()+"\">\n" +
 "\t <data key=\"d0\">"+nodeCurParent.getName()+ "</data> \n</node> \n");

        // get next parent node
        if (iter_aNodes.hasPrevious()) {
       // get all children
              iterNodes = iter_aNodes.previous().listIterator();

         // while it has children
                  while (iterNodes.hasNext()) {
          // get child
                  SGGSimpleNode nodeCurChild = iterNodes.next();

          //create graph node for Child
                  body.append("<node id=\""+nodeCurChild.getID()+"\"> \n" +

              "\t <data key=\"d0\">"+nodeCurChild.getName()+ "</data> \n</node> \n");
          //create graph edge from parent to Child
                  body.append("<edge id=\"e"+edgeCount+"\"
source=\""+nodeCurParent.getID()+"\ target=\""+nodeCurChild.getID()+"\"/>\n");

                  edgeCount++;
                          }
                  }
              }
}
```

Similar to the creation of Tree View, a display area with height and width is created to show the Graph View. The display area contains an optional number of display controls, such as

Focus Control, Drag Control, Pan Control, Zoom Control, Wheel Zoom Control, Zoom to Fit Control, and Neighborhood Highlight Controller. Wheel Zoom Control changes the scale of the viewable region with the mouse wheel movement. Zoom to Fit displays all nodes in a given group in the display bounds. The following code snippet shows the creation of a display area and the addition of controls to the display area.

```
// set up a display to show the visualization
Display display = new Display(m_vis);
display.setSize(700,700);
display.pan(350, 350);
display.setForeground(Color.GRAY);
display.setBackground(Color.WHITE);

// main display controls
display.addControlListener(new FocusControl(1));
display.addControlListener(new DragControl());
display.addControlListener(new PanControl());
display.addControlListener(new ZoomControl());
display.addControlListener(new WheelZoomControl());
display.addControlListener(new ZoomToFitControl());
display.addControlListener(new NeighborHighlightControl());
```

The display area is viewable in the user interface after adding it to a JFrame object. The following code snippet shows the rendering of the display area in a user interface.

```
// initialize enclosing window frame
JFrame frame = new JFrame("prefuse example");
frame.getContentPane().add(display);
frame.pack(); frame.setVisible(true);

// run actionLists
m_vis.run("draw");
m_vis.run("animate");
```

5.2.3. HTML View

The enhanced VEGGIE parser generates HTML from the DefaultMutableTreeNode. HTML can be viewed in browsers such as Internet Explorer, Chrome, etc. HTML generation is similar to TreeML and GraphML generation. A VEGGIE-generated DefaultMutableTreeNode object is traversed recursively to generate the HTML. The following code snippet shows the generation of HTML content from the DefaultMutableTreeNode.

```
// Generate HTML string body
private String generateHTMLBody(Enumeration children) {

        StringBuffer htmlBody = new StringBuffer();
       // iterate through children
       while (children.hasMoreElements()) {
                //get current child
          DefaultMutableTreeNode nodeCur = (DefaultMutableTreeNode)
children.nextElement();

                // add current child data to HTML
         htmlBody.append("<ul> \n <li> \n ");
         htmlBody.append(nodeCur.toString());
                htmlBody.append("\n  \n");
          // add grand children body
                htmlBody.append(generateHTMLBody(nodeCur.children()));

                htmlBody.append("</li></ul> \n");

     }
  // return HTML Body
  return htmlBody.toString();
}
```

5.2.4. Simplified Tree View

Due to the recursive nature of the graph grammars used in VEGGIE, the generated tree view contains repeated parent and child relationships which are redundant and not useful. The

Simplified Tree View simplifies the tree view by removing the repeated parent and child relationships. The following algorithm shows the generation of Simplified Tree View.

1. Get the original Tree Data.
2. Check if the present node and its parent node are equal.
3. If true, delete the present node, and add all its children to the parent.
4. If not, do nothing.
5. Do 2-4 recursively on the Tree Data.

```
// Add the parse tree descendant nodes.

while ( iter_aNodes.hasPrevious() ) {
            // Setup the sub list iterator...
            ListIterator<SGGSimpleNode> iterNodes =
        iter_aNodes.previous().listIterator();

            if ( iterNodes.hasNext() ){
                // Get this parent node...
                SGGSimpleNode nodeCur = iterNodes.next();

                // Search for this parent as an existing tree node...
                int rootID=searchTreeRoot(nodeCur.getID());
                DefaultMutableTreeNode treenodeParent =
                        simplifiedTreeSearch(rootID);

                boolean bNewTree = false;
                // If it doesn't exist, add as a new tree...
                if (rootID==-1 || treenodeParent==null) {
                        bNewTree = true;
                        treenodeParent = new DefaultMutableTreeNode(nodeCur);
                }

                String parentFullname=treenodeParent.toString();
                int startIndex=parentFullname.indexOf("(");
                int endIndex= parentFullname.indexOf(")");

String parentName= parentFullname.substring(0, startIndex-1).trim();
String parentID = parentFullname.substring(startIndex+1, endIndex).trim();
```

```
bodySimplified.append("<node id=\""+parentID+"\"> \n" +
                    "\t <data key=\"d0\">"+parentName+ "</data> \n</node> \n");

// Add this descendant's children nodes...

if ( iter_aNodes.hasPrevious() ) {
iterNodes = iter_aNodes.previous().listIterator();

while ( iterNodes.hasNext() ) {
nodeCur = iterNodes.next();

if(!nodeCur.getName().equals(parentName)){

treenodeParent.add(new DefaultMutableTreeNode(nodeCur));

bodySimplified.append("<node id=\""+nodeCur.getID()+"\"> \n" +
"\t <data key=\"d0\">"+nodeCur.getName()+ "</data> \n</node> \n");

bodySimplified.append("<edge id=\"e"+edgeCount+"\"
source=\""+parentID+"\" target=\""+nodeCur.getID()+"\"/>\n");
edgeCount++;

}
}
}

// If this descendant node is a new root...
if (bNewTree){
// Add the new root to the tree...
cc_simplifiedtreenodeRoot.add(treenodeParent);
}
}
}
```

The above code snippet shows implementation of the above algorithm. The code iterates through list of nodes. Each-iteration checks whether the current node is equal to the parent node. If they current node equal to parent node, code removes the current node and adds children of the current node to the parent node.

5.3. Results

Figure 14 shows the Prefuse Tree View. When the user clicks one of the nodes, it becomes the focus of interest, and the rest of the tree grows or shrinks appropriately. Additional details about the focused node are available using hypertext links. The search bar at the bottom of the display is utilized to search the nodes in the display area.

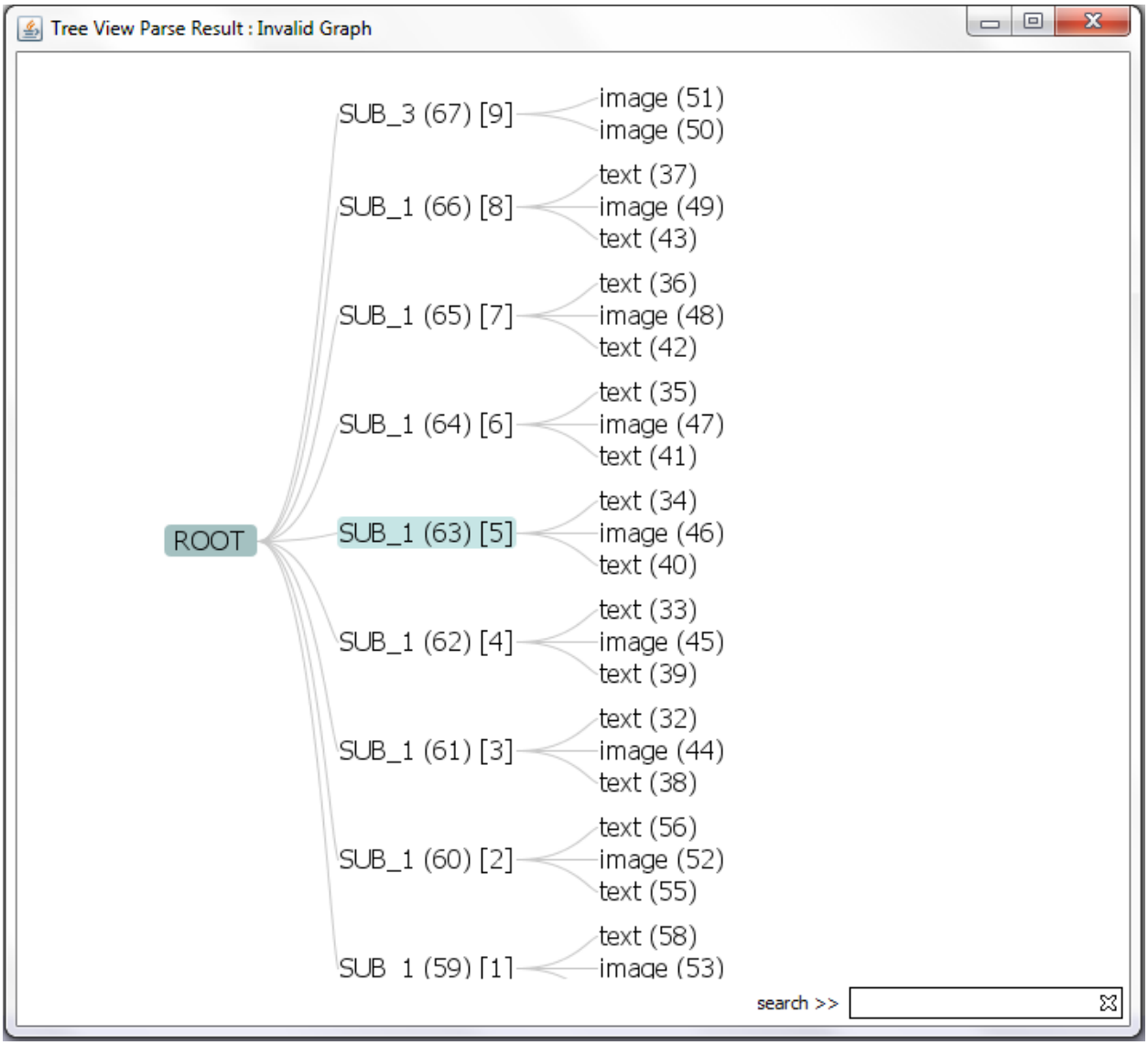


Figure 14. Tree Views: Tree View.

Figure 15 shows the Graph View. Clicking on a node focuses the displays on that node. The graph can be zoomed in and out, and can be moved. It can be zoomed to fit the window size. Nodes can be moved to untangle the graph. Multiple nodes can be selected and highlighted at the same time.

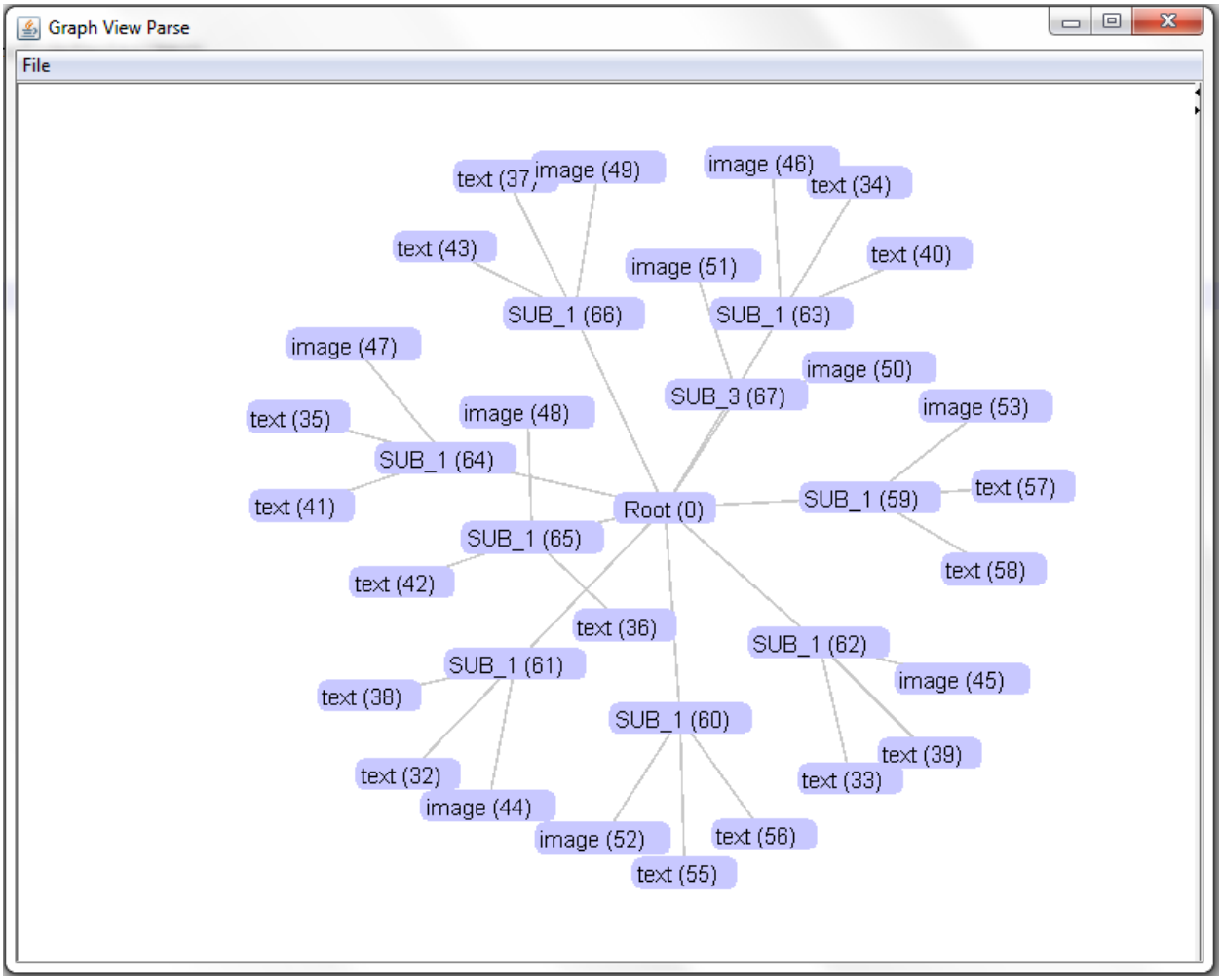


Figure 15. Tree Views: Graph View.

Figure 16 shows the generated HTML view in a web browser. It can be expanded and collapsed by clicking on the "+/-"symbols. Figures 16 (HTML View) and Figure 17 (Tree View)

are the same, except displaying the Tree View requires a Java environment, whereas the HTML view only requires a web browser.

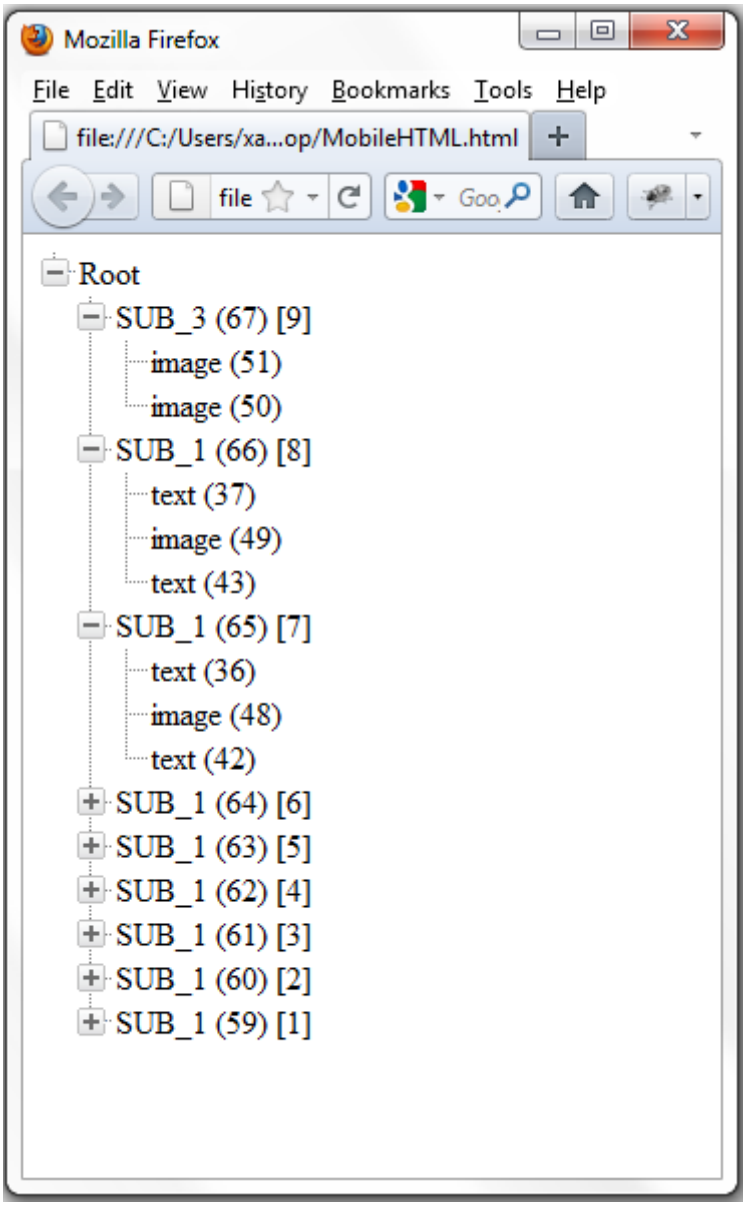


Figure 16. Tree Views: HTML View.

Figure 17 shows both the original tree view and the simplified tree view. The tree can be expanded and collapsed by clicking on the "+/-"symbols. There are no repeated parent and child relationships in the simplified tree view.

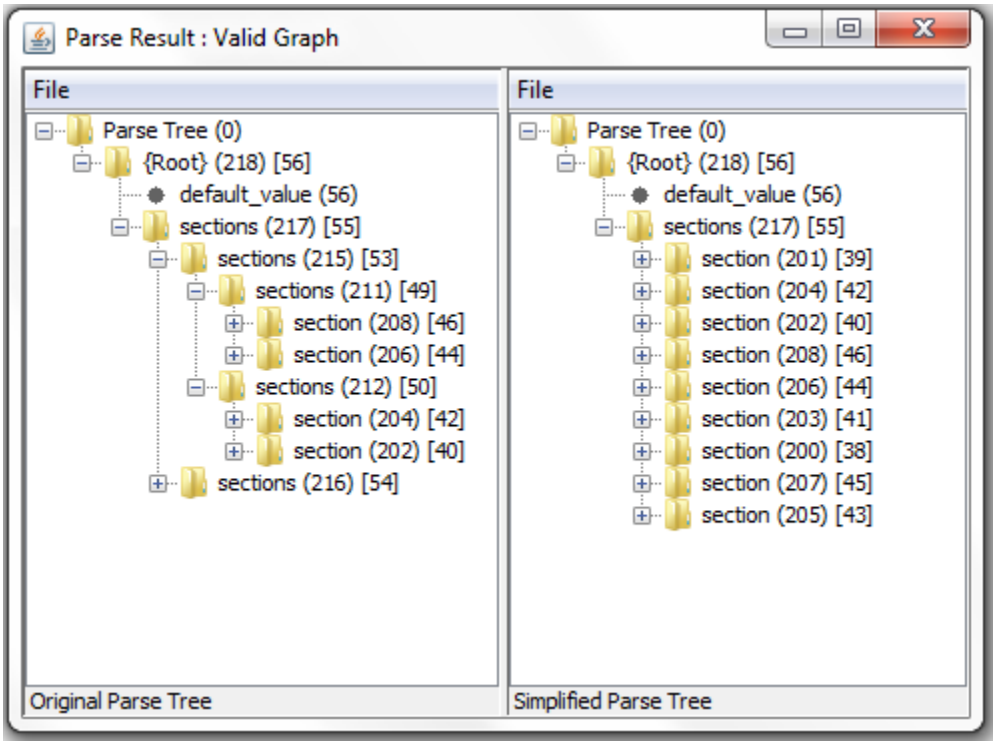


Figure 17. Tree Views: Original Tree and Simplified Tree.

CHAPTER 6. CONCLUSION

In summary, this paper bridged the gap between user-interface segmentation and analysis by providing a tool for interoperability between HPLI's segmentation tool and the VEGGIE analysis tool. This paper also provided additional views to analyze the graphical user interface semantics to derive organization of data underlying a graphical user interface. Based on the content organization, it can be re-organized to fit a wide range of display devices.

The Graph Data Converter tool saves time and effort required to convert GXL to GraphML. The tool is designed to be user friendly with easy navigation options. Additionally, the tool's UI provides edit and save options for the GXL or GraphML. The tool uses the XSLT declarative programming language to map GXL to GraphML nodes. The syntactical changes or additional support in GXL or GraphML requires changes only to the tool's XSLT mapping file. Therefore, it supports future changes in GXL or GraphML with very minimal effort.

The additional views and view features reduce the complexity and effort required to analyze the graph data. The Simplified Tree View eliminates the effort to identify and remove the duplicate nodes from the VEGGIE tree. Tree View and Graph View display all nodes and edges within display boundary to provide an overview of the user-interface data organization. The overall data organization underlying a user interface can be easily analyzed with Graph View or Tree View. Even user interfaces scaling to thousands of nodes can be easily analyzed with features such as panning, zooming, DOI, and search.

6.1. Future Work

The Graph Data Converter tool converts GXL to GraphML, and the VEGGIE tool creates visualization. The Graph Data converter can be integrated into VEGGIE to convert GXL to

GraphML and to analyze graph data in a single step. Future work includes an enhancement to VEGGIE to provide an option for the user to load GXL and create visualization.

The Graph Data Converter tool validates the GXL syntax. If the validation fails, the tool shows an error message: “Invalid GXL format.” It is difficult to correct the syntax with the current error message. Future work includes an enhancement to the tool to display the line number and syntactically invalid text for the user.

Prefuse provides more visualization styles, such as Data Mountain View, Force-Directed layout, TreeMap, etc. The future work includes exploring additional views and integrating the useful views into VEGGIE.

REFERENCES

[1] O. Barkol; R. Bergman; A. Pnueli; S. Schein. Semantic Automation from Screen Capture. HP Laboratories. Retrieved May 23, 2009, from http://www.hpl.hp.com/techreports/2009/HPL-2009-161.pdf

[2] K. Ates; K. Zhang. Constructing Veggie: Machine Learning for Context-Sensitive Graph Grammars. IEEE Computer Society. Retrieved May 23, 2009, from http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4410422

[3] XML BASICS [Online]. Retrieved June 16, 2009, from http://www.softwareag.com/xml/about/starters.htm

[4] A. Winter. Exchanging Graphs with GXL. Proc. 9th Intl. Symp. Graph Drawing. Retrieved August 19, 2009, from http://www.gupro.de/GXL

[5] U. Brandes; M. Eiglsperger; I. Herman; M. Himsolt; M.S. Marshall. GraphML Progress Report: Structural Layer Proposal. Proc. 9th Intl. Symp. Graph Drawing. Retrieved June 21, 2009, from http://graphml.graphdrawing.org/specification.html

[6] W3C. XSL Transformations [Online]. Retrieved September 5, 2009, from http://www.w3.org/TR/xslt

[7] Wikipedia [Online]. Retrieved February 13, 2010, from http://en.wikipedia.org/wiki/A_picture_is_worth_a_thousand_words

[8] Source Forge. Prefuse Documentation Manual [Online]. Retrieved November 10, 2009, from http://prefuse.org/doc/manual

[9] J. O'adadhain; D. Fisher; P. Smyth; Y.-B. Boey. Analysis and Visualization of Network Data Using JUNG. Journal of Statistical Software. Retrieved November 10, 2009, from http://jung.sourceforge.net/doc/JUNG_journal.pdf

[10] Piccolo Toolkit [Online]. Retrieved November 10, 2009, from http://www.cs.umd.edu/hcil/jazz.

[11] GraphViz Documentation Manual [Online]. Retrieved November 10, 2009, from http://www.graphviz.org/Documentation.php

[12] TreeML DTD [Online]. Retrieved December 13, 2009, from http://www.nomencurator.org/InfoVis2003/download/treeml.dtd