GXL Instance API

30th April 2003

Darya Zavgorodnya
Volker Riediger
Andreas Winter
Contents

1 Introduction 4

2 GXL Instance Graphs 5

3 GXL Instance API 9
   3.1 General description ........................................... 9
   3.2 An Example of GXL Instance API usage .................... 10

4 Appendix. Source Code of a Simple Example 19
1 Introduction

GXL (Graph eXchange Language) is a standardized language for exchanging graph-based information [2]. GXL is a sublanguage of XML (eXtensible Markup Language), which aims at providing interoperability between graph-based tools. The syntax of GXL is defined by a DTD (Document Type Definition).

GXL supports storage of typed, attributed, directed, ordered graphs, hypergraphs and hierarchical graphs [1]. Consequently, GXL can be used by the majority of tools.

GXLAPI is a C++ implementation for handling GXL instance graphs as well as GXL graph schemas. GXLAPI is based on a standard XML parser. The API methods manipulate a DOM (Document Object Model) data structure representing the underlying XML document.

GXL defines valid graphs - so called ’instance graphs’ - by means of a schema. Consequently, GXLAPI consists of two parts - GXL Instance API and GXL Schema API. This paper gives a brief description of GXL Instance API and explains its usage on an example.

The paper is structured in the following way: The second chapter considers GXL graphs and how they can be exchanged with the help of GXL in more detail. The third chapter explains usage of GXL Instance API with an example.


## 2 GXL Instance Graphs

This chapter explains how graphs can be exchanged with the help of GXL. The following description of GXL graphs applies not only to GXL instance graphs, but also to graph schemas, which are instance graphs of a metaschema.

GXL Graph Model defines GXL graphs that can be exchanged with GXL (see figure 1) [2]. GXL documents can contain an arbitrary number of graphs. As you can see from the figure, GXL-graphs are composed of graph-elements. Those are nodes, edges and relations.

![GXL Graph Model](figure1.png)

Figure 1: GXL Graph Model (graph part) [2]

GXL graphs can possess a type and an arbitrary number of attributes. The same applies to graph elements. Attributes contain exactly one value. A value (see figure 2 [2]) can be of one of five atomic domains (bool, float, int, string and enum), an URI (Unified Resource Identifier) or of one of four composite domains (bag, set, seq, tup). Relations can exist among an arbitrary
number of graph elements. Exactly one relation end (relend) corresponds to each graph element, participating in a relation. Edges are binary relations, i.e. they exist between two graph elements. RelatesTo-incidence of reloends is a generalization of from- and to-incidences of edges.

![Figure 2: GXL Graph Model (value part)](image)

Let us consider one example of a GXL graph [2]. An UML object diagram of this sample graph is shown on the figure 3. This graph contains four typed attributed nodes and three typed attributed edges. Types of the nodes and edges correspond to a graph-schema. This simple graph is represented in GXL in quite an intuitive and easy way (see figure 4).

![Figure 3: a simple graph](image)

We can describe a sample structure of a GXL document on this example. GXL documents, like all XML documents, start with a specification of an XML version and a type of document,
<xml version="1.0"/>
<!DOCTYPE gxl SYSTEM "gxl-1.0.dtd">
<gxl>
<graph id="simpleExample" edgeids="true">
  <node id="p">
    <type xlink:href="schema.gxl#Proc"/>
    <attr name="file">main.c</attr>
  </node>
  <node id="q">
    <type xlink:href="schema.gxl#Proc"/>
    <attr name="file">test.c</attr>
  </node>
  <node id="v">
    <type xlink:href="schema.gxl#Var"/>
    <attr name="line">225</attr>
  </node>
  <node id="w">
    <type xlink:href="schema.gxl#Var"/>
    <attr name="line">316</attr>
  </node>
  <edge id="r1" from="p" to="v">
    <type xlink:href="schema.gxl#ref"/>
    <attr name="line">127</attr>
  </edge>
  <edge id="r2" from="q" to="w">
    <type xlink:href="schema.gxl#ref"/>
    <attr name="line">27</attr>
  </edge>
  <edge id="c" from="p" to="q">
    <type xlink:href="schema.gxl#call"/>
    <attr name="line">42</attr>
  </edge>
</graph>
</gxl>

Figure 4: a simple graph in GXL

referring to a corresponding DTD. All tags of GXL documents are placed between <gxl> start and end tags. The document can describe an arbitrary number of graphs, each of that has a unique identifier. Each graph element belongs to a type, specified in the type element by the reference to a schema graph. Inside the corresponding <graph> tags there are descriptions of graph elements that compose the graph. Those are <node> and <edge> tags which describe corresponding graph elements. Graph elements have unique identifiers and they can be given an element type of a certain schema. Graph elements can in turn contain an arbitrary number of attributes, each containing a single value of a certain domain.

Let us consider one more example. Figure 5 [2] depicts a hypergraph, i.e. a graph with n-ary relations. This graph contains four typed, attributed nodes connected by a typed and attributed hyperedge. Those are depicted as diamonds. Note, that relation ends, also called tentacles, can be ordered at their both ends, startpoint (which is a hyperedge) and endpoints. In this example, two of the tentacles are ordered at their start-point. Tentacles can be assigned roles, which allows an arbitrary number of tentacles to be connected to a node [5]. Like in our example, the node v6 has two corresponding tentacles with different roles.
Figure 5: a graph with a hyperedge

Figure 6 depicts the corresponding GXL document for the hypergraph. The hyperedge is represented by a `<rel>` element. Tentacles of a relend are represented by `<relend>` elements. Relend elements contain references to identifiers of target elements. They can also contain attributes for specification of roles, start- and endorders.

```xml
<?xml version="1.0" ?>
<!DOCTYPE gxl SYSTEM "gxl-1.0.dtd" >
<gxl>
  <graph id="hypergraph">
    <edgeids>true</edgeids>
    <node id="v1">
      <type xlink:href="hyschema.gxl#Function"/>
      <attr name="name">main</attr>
    </node>
    <node id="v4">
      <type xlink:href="hyschema.gxl#Function"/>
      <attr name="name">max</attr>
    </node>
    <node id="v6">
      <type xlink:href="hyschema.gxl#Variable"/>
      <attr name="name">a</attr>
    </node>
    <node id="v7">
      <type xlink:href="hyschema.gxl#Variable"/>
      <attr name="name">b</attr>
    </node>
    <rel id="r1">
      <type xlink:href="hyschema.gxl#FunctionCall2"/>
      <attr name="line">8</attr>
      <relend target="v1" role="callee"/>
      <relend target="v4" role="callee"/>
      <relend target="v6" role="output" startorder="1"/>
      <relend target="v6" role="input" startorder="2"/>
      <relend target="v7" role="input" startorder="2"/>
    </rel>
  </graph>
</gxl>
```

Figure 6: A hypergraph in GXL
3  GXL Instance API

3.1 General description

The GXL Instance API allows easy access to and modification of existing GXL documents, as well as creation of new GXL documents.

GXLInstance API has been implemented in C++. It is runnable under Windows and LINUX. Internally it uses a DOM Tree for representing a document. All classes of the GXL API have a prefix GXL. The complete documentation of all methods can be generated with the help of doxygen documentation system[4].

Necessary software to compile and link the GXL Instance API are XERCES C++ parser v. 1.7.0 [3], and a collection of utilities named rgutil. The XERCES C++ parser is used for generating and validating XML documents. The rgutil utilities are used for option processing, logging and error reporting.

When creating instance graphs, several constraints have to be met. If you violate those constraints, certain exceptions will be thrown by GXL Instance API (e.g. if you want to create two nodes within the same graph with the same Ids). API design doesn’t allow to remove elements of a GXL document. If a constraint is violated (e.g. due to wrong parameters) by a constructor call, then an invalid element will be created and it will result in a creation of an invalid GXL document. This is not the case for other methods.

The following example demonstrates a subset of the available features.
3.2 An Example of GXL Instance API usage

To show the usage of GXL Instance API we will consider the following example. The object diagram on figure 7 depicts a node and edge typed, node and edge attributed, directed, ordered graph [1]. This graph represents a fragment of a program on an abstract syntax graph (ASG) level.

![Object diagram](image)

**Figure 7: Example of a simple graph**

![Graph representation](image)

**Preliminary steps**

GXL API uses Apache XML parser „XERCES“. Usage of this library requires initialization of the XML Platform Utilities (XMLPlatformUtils) [3].

```cpp
XMLPlatformUtils::Initialize();
```

Let us start with creation of an empty GXL document. We first create an object of the class GXLDocument.

```cpp
GXLDocument doc;
```

GXL API uses a DOM Tree data structure for XML document representation. The root of the DOM Tree is a gxl element. To add new elements to this tree we have to create corresponding objects and append them to other elements, which become parent elements for them. When creating objects, we have to specify a reference to a parent object as a parameter in a constructor...
call. The constructor will take care of the correct attachment of an element to its parent. To create a new graph in our empty document, we have to create an object of a class \textit{GXLGraph}. We need to get a reference to the root gxl element to enable attachment of a graph to it. A method \textit{getGXLGxl()} of the GXMLDocument class returns this reference.

GXLGxl gxl = doc.getGXLGxl();

Now we can create a new, empty GXLGraph.

GXLGraph gr(gxlEl, "simpleGraph",
    "simpleSchema.gxl#simpleGraphType", "simpleGraphRole",
    GXLConst::edgeIds, GXLConst::noHyperGraph,
    GXLConst::directedEdges );

An identifier of the graph has been assigned as \"simpleGraph\", type of it \"simpleGraphType\", specified in a schema \textit{simpleSchema.gxl}, and its role \"simpleGraphRole\". Other parameters of the graph constructor enable edge identifiers and directed edges. Our graph is not a hypergraph, which is specified in one of the parameters.

Nodes

Now we can create nodes and edges in our graph. The first parameter of node constructors is a reference to the graph \textit{gr}, the second is identifier of this node and the third is type. The following constructor call creates in our graph a node of type \textit{Function} from the schema \textit{simpleSchema.gxl} with identifier \textit{v1}.

GXLNode v1Node(gr, "v1", "simpleSchema.gxl#Function");

In the same way the other nodes can be created:

GXLNode v2Node(gr, "v2", "simpleSchema.gxl#FunctionCall");
GXLNode v3Node(gr, "v3", "simpleSchema.gxl#FunctionCall");
GXLNode v4Node(gr, "v4", "simpleSchema.gxl#Function");
GXLNode v5Node(gr, "v5", "simpleSchema.gxl#Function");
GXLNode v6Node(gr, "v6", "simpleSchema.gxl#Variable");
GXLNode v7Node(gr, "v7", "simpleSchema.gxl#Variable");

We have to attach attributes to the nodes with identifiers \textit{v1}, \textit{v4}, \textit{v5}, \textit{v6}, \textit{v7}. To do that, we have to create an object of a class \textit{GXLAttribute}. This attribute has to be attached to a graph element, in this case to a node. Constructor for creation of GXLAttributes takes a reference to the corresponding graph element and a name of an attribute to be created. All attributes to be created have a name \textit{name}:

GXLAttribute v1StrAttr( v1Node, "name");
GXLAttribute v4StrAttr( v4Node, "name");
GXLAttribute v5StrAttr( v5Node, "name");
GXLAttribute v6StrAttr( v6Node, "name");
GXLAttribute v7StrAttr( v7Node, "name");
To attach values to these attributes, we have to create objects of class \textit{GXLStringVal}, since names have to be strings. They have to be attached to the corresponding attributes. Parameters of creation constructors are references to GXLAttribute objects and values of type \textit{GXLString}.

\begin{verbatim}
GXLStringVal v1StrVal(v1StrAttr, "main");
GXLStringVal v4StrVal(v4StrAttr, "max");
GXLStringVal v5StrVal(v5StrAttr, "min");
GXLStringVal v6StrVal(v6StrAttr, "a");
GXLStringVal v7StrVal(v7StrAttr, "b");
\end{verbatim}

\textbf{Edges}

The first parameter of edge constructors is a reference to the graph \textit{gr}, the second and third refer to the incident nodes of this edge, the fourth is an edge identifier. Next parameters are edge type and a constant value, specifying if an edge is directed. The following constructor call creates an edge of type \textit{isCaller} with identifier \textit{e1}, going from node with id \textit{v2} into node with id \textit{v1}.

\begin{verbatim}
GXLEdge e1Edge(gr, v2Node, v1Node, "e1", "simpleSchema.gxl#isCaller",
GXLConst::directed);
\end{verbatim}

In the same way we create edges with ids \textit{e2}, \textit{e3}, \textit{e4}, \textit{e9}, \textit{e10}.

\begin{verbatim}
GXLEdge e2Edge(gr, v3Node, v1Node, "e2", "simpleSchema.gxl#isCaller",
GXLConst::directed);
GXLEdge e3Edge(gr, v4Node, v2Node, "e3", "simpleSchema.gxl#isCallee",
GXLConst::directed);
GXLEdge e4Edge(gr, v5Node, v3Node, "e4", "simpleSchema.gxl#isCallee",
GXLConst::directed);
GXLEdge e9Edge(gr, v6Node, v2Node, "e9", "simpleSchema.gxl#isOutput",
GXLConst::directed);
GXLEdge e10Edge(gr, v7Node, v3Node, "e10", "simpleSchema.gxl#isOutput",
GXLConst::directed);
\end{verbatim}

For edges \textit{e5}, \textit{e6}, \textit{e7}, \textit{e8} we also have to specify incidence ordering. \textit{ToOrder} and \textit{fromOrder} values can be specified as last parameters of a creation constructor:

\begin{verbatim}
GXLEdge e5Edge(gr, v6Node, v2Node, "e5", "simpleSchema.gxl#isInput",
GXLConst::directed, 1);
GXLEdge e6Edge(gr, v7Node, v2Node, "e6", "simpleSchema.gxl#isInput",
GXLConst::directed, 2);
GXLEdge e7Edge(gr, v6Node, v3Node, "e7", "simpleSchema.gxl#isInput",
GXLConst::directed, 2);
GXLEdge e8Edge(gr, v7Node, v3Node, "e8", "simpleSchema.gxl#isInput",
GXLConst::directed, 1);
\end{verbatim}

In the way similar to that of for nodes, we append attributes to the edges with identifiers \textit{e1} and \textit{e2}. Values of those attributes are integers, specifying a line in which a function calls another function. Consequently, attributes to be created have a name \textit{line}:
GXLAttribute e1IntAttr(e1Edge, "line");
GXLIntVal e1IntVal(e1IntAttr, 8);

GXLAttribute e2IntAttr(e2Edge, "line");
GXLIntVal e2IntVal(e2IntAttr, 19);

### Saving and Loading a GXL Document

To save created document to a file, we use method `store` of a class `GXLDocument`, which takes a file output stream as a parameter.

```cpp
ofstream outFileStr("simpleGraphExample.gxl");
doc.store(outFileStr);
```

A GXL document constructor has to be called to load a document.

```cpp
GXLDocument doc("simpleGraphExample.gxl");
```

The created GXL document is depicted below.

```xml
<?xml version="1.0"?>
<!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.1.dtd">
<gxl xmlns:xlink="http://www.w3.org/1999/xlink">
  <graph id="simpleGraph" role="simpleGraphRole" edgeids="true"
edgemode="directed" hypergraph="false">
    <type xlink:href="simpleSchema.gxl#simpleGraphType"/>
    <node id="v1">
      <type xlink:href="simpleSchema.gxl#Function"/>
      <attr name="name">
        <string>main</string>
      </attr>
    </node>
    <node id="v2">
      <type xlink:href="simpleSchema.gxl#FunctionCall"/>
    </node>
    <node id="v3">
      <type xlink:href="simpleSchema.gxl#FunctionCall"/>
    </node>
    <node id="v4">
      <type xlink:href="simpleSchema.gxl#Function"/>
      <attr name="name">
        <string>max</string>
      </attr>
    </node>
    <node id="v5">
      <type xlink:href="simpleSchema.gxl#Function"/>
      <attr name="name">
        <string>min</string>
      </attr>
    </node>
    <node id="v6">
```
<type xlink:href="simpleSchema.gxl#Variable"/>
<attr name="name">
  <string>a</string>
</attr>
</node>

<node id="v7">
  <type xlink:href="simpleSchema.gxl#Variable"/>
  <attr name="name">
    <string>b</string>
  </attr>
</node>

<edge id="e1" to="v1" from="v2" isdirected="true">
  <type xlink:href="simpleSchema.gxl#isCaller"/>
  <attr name="line">
    <int>8</int>
  </attr>
</edge>

<edge id="e2" to="v1" from="v3" isdirected="true">
  <type xlink:href="simpleSchema.gxl#isCaller"/>
  <attr name="line">
    <int>19</int>
  </attr>
</edge>

<edge id="e3" to="v2" from="v4" isdirected="true">
  <type xlink:href="simpleSchema.gxl#isCallee"/>
</edge>

<edge id="e4" to="v3" from="v5" isdirected="true">
  <type xlink:href="simpleSchema.gxl#isCallee"/>
</edge>

<edge id="e5" to="v2" from="v6" toorder="1" isdirected="true">
  <type xlink:href="simpleSchema.gxl#isInput"/>
</edge>

<edge id="e6" to="v2" from="v7" toorder="2" isdirected="true">
  <type xlink:href="simpleSchema.gxl#isInput"/>
</edge>

<edge id="e7" to="v3" from="v6" toorder="2" isdirected="true">
  <type xlink:href="simpleSchema.gxl#isInput"/>
</edge>

<edge id="e8" to="v3" from="v7" toorder="1" isdirected="true">
  <type xlink:href="simpleSchema.gxl#isInput"/>
</edge>

<edge id="e9" to="v2" from="v6" isdirected="true">
  <type xlink:href="simpleSchema.gxl#isOutput"/>
</edge>

<edge id="e10" to="v3" from="v7" isdirected="true">
  <type xlink:href="simpleSchema.gxl#isOutput"/>
</edge>
3.2 An Example of GXL Instance API usage

Iterators

Let us assume we need to know the identifiers and some attributes of all nodes and edges of a graph. We have to iterate through the whole document tree and print out those. We can use `GXLNodeIterator` and `GXLEdgeIterator` to solve this task. The reference to the graph to iterate through is required for creation of both iterators. We will consider only the output of node identifiers and their `name` attributes, the solution for edges is analogical (see Appendix Source Code of a Simple Example). We will create an object of the `GXLNodeIterator` class.

```
GXLNodeIterator nodeIt(gr);
```

The method `next()` of the class `GXLNodeIterator` has to be used to get all nodes of the graph as long as a `hasNext()` method of this class returns true. We can get ids, `name` attributes and types of all edges with the help of corresponding get-methods and print them out.

```
while(nodeIt.hasNext()) {
    GXLNode node = nodeIt.next();
    GXLString tmpId = node.getId();
    GXLString tmpType = node.getTypeName();
    GXLString tmpAttrName = node.getAttribute("name");
    cout << "Node Id: " << GXLUtil::transcode(tmpId)
         << "	Node Type: " << GXLUtil::transcode(tmpType)
         << "	Name Attribute: " << GXLUtil::transcode(tmpAttrName) << endl;
}
```

Another Simple Task

One more simple task to solve could be, for example, print out the first parameter for all called functions with corresponding variable and function names. To do so, we have to iterate through all nodes, and only for nodes of type `FunctionCall`, iterate through all incoming edges. For edges of type `isCallee` retrieve the incident node, it should be of type `Function`, and get its `name` attribute. It will be a name of a called function. For edges of type `isInput` we also have to retrieve incident node, but only if a to-order for the edge equals 1. Retrieved node should be of type `Variable`, and we can get `name` attribute for this node. It will be a name of a first parameter of the function call. The following code is an implementation of a possible solution:

```
//Return a first parameter for all functions with corresponding
//variable and function names

//Create a node iterator
GXLNodeIterator nodeIter(gr);

//Iterate through all nodes of the graph
while(nodeIter.hasNext()){
    //Retrieve next node
    GXLNode node = nodeIter.next();
    //Get type of this node
    GXLString tmpType = node.getTypeName();
```
//Check for node type
if(tmpType.equals("simpleSchema.gxl#FunctionCall")){

    //For FunctionCall nodes create incoming edge iterator
    //to search for a called function name
    GXLIncomingEdgeIterator InEdgeIt(node);

    //Iterate through all incoming edges
    while(InEdgeIt.hasNext()){  

        //Retrieve next incoming edge
        GXLEdge edge = InEdgeIt.next();
        //Get type of this edge
        GXLString typeName = edge.getTypeName();

        if(typeName.equals("simpleSchema.gxl#isCallee")){

             //If this type is isCallee
             //Retrieve a value of name attribute of the incident node
             GXLAttribute funcAttr = edge.getFrom().getAttribute("name");
             GXLString funcName = funcAttr.getValue().getString();

             //Print a name of a called function
             cout<<endl<<"Function "<<GXLUtil::transcode(funcName)  
                " is called with the first parameter ";
        }
    }
}

//For FunctionCall nodes create incoming edge iterator
//to search for a first parameter of a called function
GXLIncomingEdgeIterator InEdgeItVarSearch(node);

//Iterate through all incoming edges
while(InEdgeItVarSearch.hasNext()){ 

    //Retrieve next incoming edge
    GXLEdge edge = InEdgeItVarSearch.next();
    //Get type of this edge
    GXLString typeName = edge.getTypeName();

    //If this type is isInput and to-order equals 1
    if((typeName.equals("simpleSchema.gxl#isInput")  
        &&(edge.toOrder() == 1))){

        //Retrieve a value of name attribute of the incident node
        GXLAttribute varAttr = edge.getFrom().getAttribute("name");
        GXLString varName = varAttr.getValue().getString();

        //Print a name of a first parameter of a called function
        cout<<GXLUtil::transcode(varName);
    }
}
}
We will get the necessary results:

Function max is called with the first parameter a.
Function min is called with the first parameter b.

**Finish Application**

If we want to finish our application, we have to terminate the Xerces XMLPlatformUtils [3].

XMLPlatformUtils::Terminate();
References


void simpleGraphExample()
{
    // Create an empty GXL document
    GXLDocument doc;
    if (!doc.isValid()) return false;

    // Create an empty graph
    cout << "- test if graph is created " << endl;
    GXLGxl gxlEl = doc.getXML();
    GXLGraph gr(gxlEl, "simpleGraph", "simpleSchema.gxl#simpleGraphType",
               "simpleGraphRole", GXLConst::edgeIds, GXLConst::noHyperGraph,
               GXLConst::directedEdges);

    // Create nodes with corresponding attributes
    GXLNode v1Node(gr, "v1", "simpleSchema.gxl#Function");
    GXLAttribute v1StrAttr( v1Node, "name");
    GXLStringVal v1StrVal(v1StrAttr, "main");

    GXLNode v2Node(gr, "v2", "simpleSchema.gxl#FunctionCall");
    GXLNode v3Node(gr, "v3", "simpleSchema.gxl#FunctionCall");

    GXLNode v4Node(gr, "v4", "simpleSchema.gxl#Function");
    GXLAttribute v4StrAttr( v4Node, "name");
    GXLStringVal v4StrVal(v4StrAttr, "max");

    GXLNode v5Node(gr, "v5", "simpleSchema.gxl#Function");
    GXLAttribute v5StrAttr( v5Node, "name");
    GXLStringVal v5StrVal(v5StrAttr, "min");

    GXLNode v6Node(gr, "v6", "simpleSchema.gxl#Variable");
    GXLAttribute v6StrAttr( v6Node, "name");
    GXLStringVal v6StrVal(v6StrAttr, "a");

    GXLNode v7Node(gr, "v7", "simpleSchema.gxl#Variable");
    GXLAttribute v7StrAttr( v7Node, "name");
    GXLStringVal v7StrVal(v7StrAttr, "b");

    // Create edges with corresponding attributes
    GXLEdge e1Edge(gr, v2Node, v1Node, "e1", "simpleSchema.gxl#isCaller",
                   GXLConst::directed);
    GXLAttribute e1IntAttr( e1Edge, "line");
    GXLIntVal e1IntVal(e1IntAttr, 8);

    GXLEdge e2Edge(gr, v3Node, v1Node, "e2", "simpleSchema.gxl#isCaller",
                   GXLConst::directed);
    GXLAttribute e2IntAttr( e2Edge, "line");
    GXLIntVal e2IntVal(e2IntAttr, 19);

    GXLEdge e3Edge(gr, v4Node, v2Node, "e3", "simpleSchema.gxl#isCallee",
                   GXLConst::directed);
    GXLEdge e4Edge(gr, v5Node, v3Node, "e4", "simpleSchema.gxl#isCallee",
                   GXLConst::directed);
GXLEdge e5Edge(gr, v6Node, v2Node, "e5", "simpleSchema.gxl#isInput",
GXLConst::directed, 1);
GXLEdge e6Edge(gr, v7Node, v2Node, "e6", "simpleSchema.gxl#isInput",
GXLConst::directed, 2);
GXLEdge e7Edge(gr, v6Node, v3Node, "e7", "simpleSchema.gxl#isInput",
GXLConst::directed, 2);
GXLEdge e8Edge(gr, v7Node, v3Node, "e8", "simpleSchema.gxl#isInput",
GXLConst::directed, 1);
GXLEdge e9Edge(gr, v6Node, v2Node, "e9", "simpleSchema.gxl#isOutput",
GXLConst::directed);
GXLEdge e10Edge(gr, v7Node, v3Node, "e10", "simpleSchema.gxl#isOutput",
GXLConst::directed);

//Print out a document on a screen
doc.store(cout);

//Save a document
ofstream outFileStr("simpleGraphExample.gxl");
doc.store(outFileStr);

//Traverse
//Print Identifiers of all Nodes of the Graph
GXLN odeIterator nodeIt(gr);
cout<<"List of all nodes"<<endl;
while(nodeIt.hasNext()){
    GXLN ode node = nodeIt.next();
    GXLS tring tmpId = node.getId();
    GXLS tring tmpType = node.getTypeName();
    GXLS tring tmpAttrName = node.getAttribute("name");
    cout << "Node Id: " << GXLUtil::transcode(tmpId)
         << "\tNode Type: " << GXLUtil::transcode(tmpType)
         << "\tName Attribute: " << GXLUtil::transcode(tmpAttrName) << endl;
}

GXLEdgeIterator edgeIt(gr);
cout<<"List of all edges"<<endl;
while(edgeIt.hasNext()){
    GXLEdge edge = edgeIt.next();
    GXLS tring tmpId = edge.getId();
    GXLS tring tmpType = edge.getTypeName();
    cout<<"Edge Id: "<<GXLUtil::transcode(tmpId)"\t Edge Type: "
        <<GXLUtil::transcode(tmpType)<<endl;
}

//Return a first parameter for all functions
//with corresponding variable and function names

//Create a node iterator
GXLNodeIterator nodeIter(gr);

// Iterate through all nodes of the graph
while(nodeIter.hasNext()){
    // Retrieve next node
    GXLNode node = nodeIter.next();
    // Get type of this node
    GXLString tmpType = node.getTypeName();

    // Check for node type
    if(tmpType.equals("simpleSchema.gxl#FunctionCall")){
        // For FunctionCall nodes create incoming edge iterator
        // to search for a called function name
        GXLIncomingEdgeIterator InEdgeIt(node);

        // Iterate through all incoming edges
        while(InEdgeIt.hasNext()){
            // Retrieve next incoming edge
            GXLEdge edge = InEdgeIt.next();
            // Get type of this edge
            GXLString typeName = edge.getTypeName();
            // If this type is isCallee
            if(typeName.equals("simpleSchema.gxl#isCallee")){
                // Retrieve a value of name attribute of the incident node
                GXLAttribute funcAttr = edge.getFrom().getAttribute("name");
                GXLString funcName = funcAttr.getValue().getString();

                // Print a name of a called function
                cout<<endl<<"Function "<<GXLUtil::transcode(funcName) <<" is called with the first parameter ";
            }
        }
    }
}

// For FunctionCall nodes create incoming edge iterator
// to search for a first parameter of a called function
GXLIncomingEdgeIterator InEdgeItVarSearch(node);

// Iterate through all incoming edges
while(InEdgeItVarSearch.hasNext()){
    // Retrieve next incoming edge
    GXLEdge edge = InEdgeItVarSearch.next();
    // Get type of this edge
    GXLString typeName = edge.getTypeName();
    // If this type is isInput and to-order equals 1
    if((typeName.equals("simpleSchema.gxl#isInput"))
        &&(edge.toOrder() == 1)){
        // Retrieve a value of name attribute of the incident node
        GXLAttribute varAttr = edge.getFrom().getAttribute("name");
        GXLString varName = varAttr.getValue().getString();
    }
int main(int argc, char *argv[]) {

    // Initialize the XML4C system
    try {
        RGLOG(1, "Initialize XML4C");
        XMLPlatformUtils::Initialize();
    }

    catch (const XMLException& e) {
        cerr << e;
        throw;
    }

    simpleGraphExample();

    RGLOG(1, "Terminate XML4C");
    XMLPlatformUtils::Terminate();
    return 0;
}