An Overview of the GXL Graph Exchange Language*

Andreas Winter, Bernt Kullbach, and Volker Riediger

Universität Koblenz-Landau
Institut für Softwaretechnik
D-56016 Koblenz, Postfach 201602
mailto:(winter|kullbach|riediger)@uni-koblenz.de
http://www.gupro.de/(winter|kullbach|riediger)

Abstract.

GXL (Graph eXchange Language) is designed to be a standard exchange format for graph-based tools. GXL is defined as an XML sublanguage, which offers support for exchanging instance graphs together with their appropriate schema information in a uniform format. Formally, GXL is based on typed, attributed, ordered directed graphs, which are extended by concepts to support representing hypergraphs and hierarchical graphs. Using this general graph model, GXL offers a versatile support for exchanging nearly all kinds of graphs.
This report intends to give a short overview on the main features of GXL.

1 Motivation and Background

A great variety of software tools relies on graphs as internal data representation. A standardized language for exchanging those graphs offers a first step in improving *interoperability* between these tools. For instance, a common graph interchange format allows building a powerful reverse engineering workbench. Such a reverse engineering workbench composes various graph-based tools like extractors (e. g. scanner, parser), abstractors (e. g. query tools, structure recognition tools, slicing tools etc.), and visualizers (e. g. graph and diagram visualizer, code browser). [22] gives an overview on existing combinations of tool components used in various reverse engineering projects.

The development of *GXL (Graph eXchange Language)* aims at supporting data interoperability between reverse engineering tools. GXL was ratified as *standard exchange format* in reverse engineering at the Dagstuhl Seminar "Interoperability of Reverse Engineering Tools" in January 2001 [4]. But since GXL was developed as a general format for describing graph structures, it is applicable in further areas of tool interoperability. Especially, GXL is used to define the graph part in the exchange format GTXL (Graph Transformation eXchange Language) [17], [34].

* This paper is an extended abstract of [37].

S. Diehl (Ed.): Software Visualization, LNCS 2269, pp. 324–336, 2002.

GXL originated in a merger of *GRAph eXchange format (GraX)* [7], *Tuple Attribute Language (TA)* [21], and the graph format of the *PROGRES* graph rewriting system [32]. The graph models used here were supplemented by additional concepts to handle hierarchical graphs and hypergraphs. Furthermore, GXL includes ideas from common exchange formats used in reverse engineering, including *Relation Partition Algebra (RPA)* [27] and *Rigi Standard Format (RSF)* [38]. The development of GXL was also influenced by various formats used in graph drawing, e. g. *daVinci* [10], *GML* [16], *XGMML (eXtensible Graph Markup and Modeling Language)* [39], and *GraphXML* [20]. Thus, GXL covers most of the important graph formats. GXL can be viewed as a generalization of these formats.

Exchanging graphs with GXL deals with both *instance graphs* and their corresponding *graph schemas*. Firstly, GXL offers a versatile support for exchanging all kinds of graphs based on *typed, attributed, directed, ordered graphs* including *hypergraphs* and *hierarchical graphs*. Secondly, GXL offers means for exchanging graph schemas representing the graph structure, i. e. the definition of node and edge types, their attribute schemas and their incidence structure. Both, instance graphs and graph schemas, are exchanged by XML documents (Extended Markup Language) [35].

This paper introduces into the basic concepts of GXL version 1.0 for exchanging instance graphs (cf. section 2) and graph schemas (cf. section 3). The language definition of GXL is given by its XML document type definition (DTD). Section 4 summarizes the current usage of GXL.

A more comprehensive description of GXL is given in [37]. Up-to-date information including tutorials and further GXL documents are collected at `http://www.gupro.de/GXL`.

2 Exchanging Graphs

Due to their mathematical foundation and algorithmic power, graphs are a common data structure in software engineering. Different graph models, e. g. directed graphs, undirected graphs, node attributed graphs, edge attributed graphs, node typed graphs, edge typed graphs, ordered graphs, relational graphs, acyclic graphs, trees, etc. or combinations of these graph models are utilized in many software systems. To support interoperability of graph based tools, the underlying graph model has to be as rich as possible to cover most of these graph models.

Such a common graph model is given by *typed, attributed, directed, ordered graphs (TGraphs)* [6], [7]. TGraphs are *directed* graphs, whose nodes and edges may be *attributed* and *typed*. Each type can be assigned an individual attribute schema specifying the possible attributes of nodes and edges. Furthermore, TGraphs are *ordered*, i. e. the node set, the edge set, and the sets of edges incident to a node have a total ordering. This ordering gives modeling power to describe sequences of objects (e. g. parameter lists) and facilitates the implementation of deterministic graph algorithms. In applying TGraphs to the sketched

graph models, not all properties of TGraphs have to be used to their full extent. These graph models can be viewed as specializations of TGraphs. Exchanging TGraphs with GXL is introduced in section 2.1

To offer support for *hypergraphs* and *hierarchical graphs*, TGraphs were extended by n-ary edges and by nodes and edges containing lower level graphs. GXL language constructs for exchanging those extended graphs are sketched in section 2.2. The complete GXL language definition in terms of an XML document type definition is given in section 2.3.

2.1 Exchanging Typed, Attributed, Directed, Ordered Graphs

The UML object diagram (cf. [31]) in figure 1 shows a node and edge typed, node and edge attributed, directed, ordered graph representing a program fragment on ASG (abstract syntax graph) level. Function *main* calls function $a = max(a, b)$ in line *8* and function $b = min(b, a)$ in line *19*.

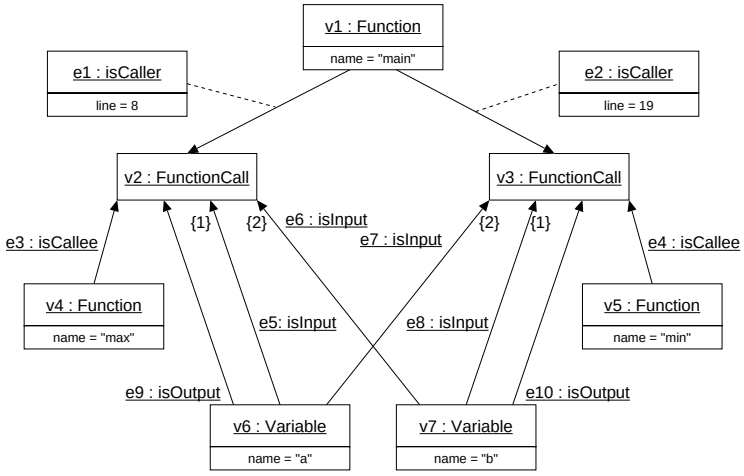

Fig. 1. Typed, attributed, directed, ordered graph

The functions *main*, *max* and *min* are represented by nodes of type *Function*. These nodes are attributed with the functions' name. *FunctionCall* nodes represent the calls of functions *max* and *min*. *FunctionCall* nodes are associated to the caller by *isCaller* edges and to the callee by *isCallee* edges. *isCaller* edges are attributed with a line attribute showing the line number which contains the call. Input parameters (represented by *Variable* nodes that are attributed with the variables' name) are associated by *isInput* edges. The ordering of parameter lists is given by ordering the incidences of *isInput* edges pointing to *FunctionCall* nodes. The first edge of type *isInput* incident to function call *v2* (modeling the call of *max(a,b)*) comes from node *v6* representing variable *a*. The second edge of type *isInput* connects to the second parameter *b* (node *v7*). The incidences

of *isInput* edges associated with node *v3* model the reversed parameter order. Output parameters are associated to their function calls by *isOutput* edges.

Exchanging graphs like the one in figure 1 requires language constructs for representing nodes, edges and their incidence relation. Furthermore, support for describing type information and attribute values is needed.

```
<?xml version = "1.0" ?>
<!DOCTYPE gxl
    SYSTEM "gxl-1.0.dtd">
<gxl>
<graph id = "simpleGraph"
        edgeids = "true">
  <type xlink:href =
  "schema.gxl#Schema"/>
  <node id = "v1" >
    <type xlink:href =
    "schema.gxl#Function"/>
    <attr name = "name" >
      <string>main</string>
    </attr>
  </node>
  <node id = "v2" >
    <type xlink:href =
    "schema.gxl#FunctionCall"/>
  </node>
  <node id = "v3" >
    <type xlink:href =
    "schema.gxl#FunctionCall"/>
  </node>
  <node id = "v4" >
    <type xlink:href =
    "schema.gxl#Function"/>
    <attr name = "name" >
      <string>max</string>
    </attr>
  </node>
  <node id = "v5" >
    <type xlink:href =
    "schema.gxl#Function"/>
    <attr name = "name" >
      <string>min</string>
    </attr>
  </node>
  <node id = "v6" >
    <type xlink:href =
    "schema.gxl#Variable"/>
    <attr name = "name" >
      <string>a</string>
    </attr>
  </node>
  <node id = "v7" >
    <type xlink:href =
    "schema.gxl#Variable"/>
    <attr name = "name" >
      <string>b</string>
    </attr>
  </node>
  <edge id = "e1"
    from = "v2" to = v1">
    <type xlink:href =
    "schema.gxl#isCaller"/>
    <attr name = "line" >
      <int>8</int>
    </attr>
  </edge>
  <edge id = "e2"
    from = "v3" to = v1">
    <type xlink:href =
    "schema.gxl#isCaller"/>
    <attr name = "line" >
      <int>19</int>
    </attr>
  </edge>
  <edge id = "e3"
    from = "v4" to = v2">
    <type xlink:href =
    "schema.gxl#isCallee"/>
  </edge>
  <edge id = "e9"
    from = "v6" to = v2"
    <type xlink:href =
    "schema.gxl#isOutput">
  </edge>
  <edge id = "e5"
    from = "v6" to = v2"
    toorder = "1">
    <type xlink:href =
    "schema.gxl#isInput"/>
  </edge>
  <edge id = "e6"
    from = "v7" to = v2"
    toorder = "2">
    <type xlink:href =
    "schema.gxl#isInput"/>
  </edge>
  <edge id = "e7"
    from = "v6" to = v3"
    toorder = "2">
    <type xlink:href =
    "schema .gxl#isInput"/>
  </edge>
  <edge id = "e8"
    from = "v7" to = v3"
    toorder = "1">
    <type xlink:href =
    "schema.gxl#isInput"/>
  </edge>
  <edge id = "e9"
    from = "v6" to = v2"
    <type xlink:href =
    "schema.gxl#isOutput">
  </edge>
  <edge id = "e10"
    from = "v7" to = v3"
    <type xlink:href =
    "schema.gxl#isOutput">
  </edge>
</graph>
</gxl>
```

Fig. 2. GXL representation of graph from figure 1

Figure 2 depicts the graph from figure 1 as GXL document. XML documents start with specifying the XML version and the underlying document type definition, here "gxl-1.0.dtd" (cf. figure 3). The body of a GXL document is enclosed in <**gxl**> tags. The GXL document in figure 2 contains a graph with a unique identifier "simpleGraph". The graph refers to its associated graph schema object Schema (cf. section 3) stored in file schema.gxl.

Nodes and edges of a given graph are depicted by <**node**> and <**edge**> elements which can be addressed by their id attribute. Incidence information of ed-

ges including edge orientation is stored in from and to attributes within <**edge**> tags. Ordering of incidences is also modeled here. Attributes fromorder and toorder represent the position of an edge in the incidence list of its start and target node.

Node and edge types are represented by links pointing to the appropriate schema information. These links are enclosed in <**type**> elements.

<**node**> and <**edge**> elements may additionally contain further attribute information. <**attr**> elements describe attribute names and values. Like OCL [36], GXL provides <**bool**>, <**int**>, <**float**>, and <**string**> attributes. Furthermore, enumeration values (<**enum**>) and URI-references (<**locator**>) to externally stored objects are supported. Attribute values might be substructured. Here, GXL offers composite attributes like sequences (<**seq**>), sets (<**set**>), multi sets (<**bag**>), and tuples (<**tup**>).

2.2 Exchanging Extended Graphs

In addition to typed, attributed, ordered, directed graphs, GXL provides the exchange of *hypergraphs* and *hierarchical graphs*.

Hypergraphs contain n-ary edges (hyperedges) connecting not only two adjacent nodes. *Hyperedges* are exchanged by <**rel**> elements, containing references to the incident graph objects. These references (tentacles) are stored in <**relend**> elements (relation end).

Edges can be viewed as 2-ary hyperedges. Thus, in GXL, edge information can be represented by *binary hyperedges*. Since graphs with (binary) edges are widespread in software engineering and most applications deal with graphs instead of hypergraphs, GXL offers both, the element <**edge**> for exchanging (binary) edges and and the element <**rel**> for hyperedges.

Like binary edges, tentacles may be directed or undirected as well as ordered. The ordering of tentacles incident to their target object and the ordering of tentacles with respect to their hyperedge object is represented analogously to the ordering of incident edges by using XML attributes.

Hierarchical graphs are graphs where nodes, edges, and hyperedges contain further graphs. GXL supports exchanging hierarchical graphs by nesting those inner graphs as <**graph**> elements in their enclosing node, edge, and hyperedge representation.

2.3 GXL Document Type Definition

The language features of GXL for exchanging typed, attributed, directed, ordered graphs (cf. section 2.1) and extended graphs (cf. section 2.2) are summarized in a conceptual model defining the graph model supported by GXL. The GXL graph model is completely described at `http://www.gupro.de/GXL/` (graph model) with its graph structure part and its attribute part.

Since GXL is an XML sublanguage, the GXL graph model had to be transcribed into an XML document type definition (DTD) or an appropriate XML

schema definition. To keep GXL simple and less verbose, this translation was done by hand. The resulting DTD (cf. figure 3, a commented version is given at `http://www.gupro.de/GXL` (DTD)) requires only 18 XML elements. In contrast, an appropriate DTD generated with IBMs XMI Toolkit [23] according the XML Metadata Interchange (XMI) principles for developing DTDs [26, section 3] requires 66 elements for the GXL core and and additional 63 elements for XMI and Corba related aspects.

```
<!– extensions –>
<!ENTITY % gxl-extension        "" >
<!ENTITY % graph-extension      "" >
<!ENTITY % node-extension       "" >
<!ENTITY % edge-extension       "" >
<!ENTITY % rel-extension        "" >
<!ENTITY % value-extension      "" >
<!ENTITY % relend-extension     "" >
<!ENTITY % gxl-attr-extension   "" >
<!ENTITY % graph-attr-extension "" >
<!ENTITY % node-attr-extension  "" >
<!ENTITY % edge-attr-extension  "" >
<!ENTITY % rel-attr-extension   "" >
<!ENTITY % relend-attr-extension "" >

<!– attribute values –>
<!ENTITY % val " locator | bool | int |
                 float | string | enum |
                 seq | set | bag | tup
                 % value-extension;" >

<!– gxl –>
<!ELEMENT gxl (graph* %gxl-extension;) >
<!ATTLIST gxl
    xmlns:xlink CDATA          #FIXED
                "www.w3.org/1999/xlink"
    %gxl-attr-extension; >

<!– type –>
<!ELEMENT type EMPTY>
<!ATTLIST type
    xlink:type  (simple)       #FIXED "simple"
    xlink:href  CDATA          #REQUIRED >

<!– graph –>
<!ELEMENT graph (type? , attr* ,
                 ( node | edge | rel )*
                 %graph-extension;) >
<!ATTLIST graph
    id          ID             #REQUIRED
    role        NMTOKEN        #IMPLIED
    edgeids     ( true | false ) "false"
    hypergraph  ( true | false ) "false"
    edgemode    ( directed | undirected |
                  defaultdirected |
                  defaultundirected)
                "directed"
    %graph-attr-extension; >

<!– node –>
<!ELEMENT node (type? , attr*, graph*
                %node-extension;) >
<!ATTLIST node
    id          ID             #REQUIRED
    %node-attr-extension; >

<!– edge –>
<!ELEMENT edge (type?, attr*, graph*
                %edge-extension;) >
<!ATTLIST edge
    id          ID             #IMPLIED
    from        IDREF          #REQUIRED
    to          IDREF          #REQUIRED
    fromorder   CDATA          #IMPLIED
    toorder     CDATA          #IMPLIED
    isdirected  ( true | false ) #IMPLIED
    %edge-attr-extension; >

<!– rel –>
<!ELEMENT rel (type? , attr*, graph*, relend*
               %rel-extension;) >
<!ATTLIST rel
    id          ID             #IMPLIED
    isdirected  ( true | false ) #IMPLIED
    %rel-attr-extension; >

<!– relend –>
<!ELEMENT relend (attr* %relend-extension;) >
<!ATTLIST relend
    target      IDREF          #REQUIRED
    role        NMTOKEN        #IMPLIED
    direction   ( in | out | none) #IMPLIED
    startorder  CDATA          #IMPLIED
    endorder    CDATA          #IMPLIED
    %relend-attr-extension; >

<!– attr –>
<!ELEMENT attr (type?, attr*, (%val;)) >
<!ATTLIST attr
    id          IDREF          #IMPLIED
    name        NMTOKEN        #REQUIRED
    kind        NMTOKEN        #IMPLIED >

<!– locator –>
<!ELEMENT locator EMPTY >
<!ATTLIST locator
    xlink:type  (simple)       #FIXED "simple"
    xlink:href  CDATA          #IMPLIED >

<!– attribute values –>
<!ELEMENT bool (#PCDATA) >
<!ELEMENT int (#PCDATA) >
<!ELEMENT float (#PCDATA) >
<!ELEMENT string (#PCDATA) >
<!ELEMENT enum (#PCDATA) >
<!ELEMENT seq (%val;)* >
<!ELEMENT set (%val;)* >
<!ELEMENT bag (%val;)* >
<!ELEMENT tup (%val;)* >
```

Fig. 3. GXL Document Type Definition

3 Exchanging Graph Schemas

Graphs only offer a plain structured means for describing objects (nodes) and their interrelationship (edges, hyperedges). Graphs have no meaning of their own. The meaning of graphs corresponds to the context in which they are used and exchanged. The application and interchange context determines

- which node, edge, and hyperedge types are used,
- how nodes, edges, and hyperedges of given types are related,
- which attribute structures are associated to nodes, edges, and hyperedges, and
- which additional constraints (like ordering of incidences, degree-restrictions etc.) have to be complied.

This schematic data can be described by *conceptual modeling techniques*. Class diagrams offer a suited declarative language to define graph classes with respect to a given application or interchange context [7].

3.1 Describing Graph Classes by UML Class Diagrams

In GXL, graph classes are defined by UML class diagrams [31]. Figure 4 shows a graph schema defining classes of graphs like the one given in figure 1. Node classes (*FunctionCall*, *Function*, and *Variable*) are defined by classes. Edge classes (*isCallee*, *isInput*, and *isOutput*) are defined by associations. Attributed edge classes (*isCaller*) are described by association classes. Like classes, they contain the associated attribute structures. The orientation of edges is depicted by a filled triangle (cf. [31, p. 155]. Multiplicities denote degree restrictions. Ordering of incidences is indicated by the keyword {ordered}.

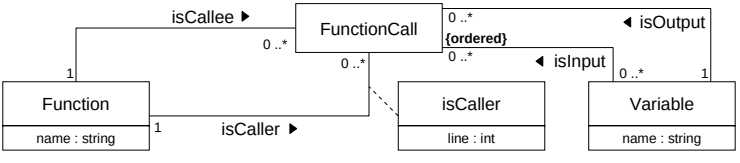

Fig. 4. Graph schema (UML class diagram)

In a similar way, UML class diagrams offer language constructs to model classes of hyperedges (diamonds) and classes of attributed hyperedges (diamonds with an associated class). The definition of hierarchical graphs requires an additional language construct representing graph classes themselves. This is done by <<GraphClass>> stereotypes.

To offer up-to-date conceptual modeling power, the GXL schema notation provides generalization of node-, edge-, and hyperedge classes as well as aggregation and composition by using the appropriate UML notation.

3.2 Describing Graph Classes by Graphs

Since UML class diagrams are structured information themselves, they may be represented as graphs as well. For exchanging graph schemas in GXL, UML class diagrams are transfered into equivalent graph representations. Thus, instance graphs and schemas are exchanged with the *same type of document*, i. e. XML documents matching the GXL DTD (cf. section 2.3).

In contrast to the strategy proposed by XML Meta Data Interchange (XMI) [26], GXL schemas are *not* exchanged by XML documents according to the Meta Object Facility (MOF) [25]. XMI/MOF offers a general, but very verbose format for exchanging UML class diagrams as XML streams. By generating individual document type definitions to a given UML class diagram, it also supports exchanging instance graphs as XML documents. Next to its exaggerated verbosity, which contradicts the requirement for exchange formats of as compact as possible documents, the XMI/MOF approach requires *different types of documents* for representing schema and instance graphs. Especially in applications dealing with schema information on instance level (e. g. in tools for editing and analyzing schemas), this leads to the disadvantage of different documents representing the same information, one on instance level (as XML document) and one on schema level (as XML DTD). The GXL approach treats schema and instance information in exactly the same way. Schema and instance graphs are exchanged according to the DTD given in Figure 3.

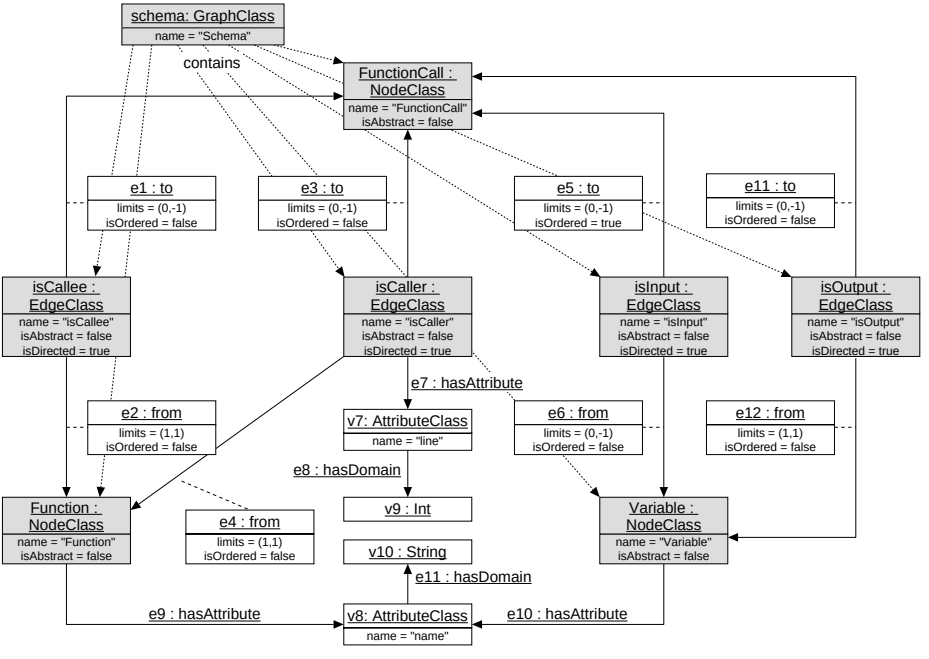

Fig. 5. Graph schema (schema graph)

Figure 5 depicts the transformation of the class diagram in figure 4 into a node and edge typed, node and edge attributed, directed graph. Node classes, edge classes, attributes and their domains are modeled by nodes of suitable node types. Their attributes describe further properties. Interrelationships between surrogates of these classes are represented by edges of proper types. Attribute information is associated with surrogates of node classes, edge classes and associations by *hasAttribute* and *hasDomain* edges. *from* and *to* edges model incidences of associations including their orientation. Multiplicities of associations are stored in limits-attributes. The boolean attribute isOrdered indicates ordered incidences.

Further attribute types and extended concepts like graph hierarchy, classes of hyperedges, aggregation and composition, generalization and default attribute values are modeled analogously.

GXL documents, representing instance graphs of a given graph schema refer to those nodes of the equivalent schema graph representing node classes (*NodeClass*) and edge classes (*EdgeClass*). The graph class itself is represented by a *GraphClass* node. This node is connected by *contains* edges to all surrogates of node and edge classes defined in this graph class. Schema references in GXL-documents refer to these *GraphClass* nodes in GXL schema graphs (cf. the type element of graph simpleGraph in figure 2). In figure 5 nodes representing these class definitions are shaded. These items are refered to by the instance graph in figure 1.

GXL views edges as *first class objects* which have their own identity, might be typed and attributed, and might be included in a generalization hierarchy. Thus, surrogates of associations and associated classes have to be connected to further information. For generality and simplicity reasons GXL schema graphs are restricted to ordinary typed, attributed, directed graphs. Hence, this edge-like information is represented by nodes as well. Although the GXL DTD provides edges connecting edges, GXL schema graphs do not use this feature.

The graph class of correct GXL schema graphs is represented as a GXL schema. A UML diagram representing this *GXL metaschema* is presented with its graph part, its attribute part, and its value part at `http://www.gupro.de/GXL/` (meta schema).

Each UML class diagram defining a GXL graph schema can be represented by a graph (schema graph) matching the GXL metaschema. Thus, schema graphs are instances of the GXL metaschema. They are exchanged like all instance graphs (cf. section 2) referring to a GXL document, here representing the GXL metaschema. Since the schema graph representing the GXL metaschema is an instance of itself, it is exchanged by a self referring GXL document.

4 Using GXL

At the Dagstuhl seminar on "Interoperability of Reverse Engineering Tools" GXL version 1.0 was ratified as the *standard exchange format* in reverse engineering [4]. Currently, various groups in software (re)engineering are implementing

GXL import and export facilities to their tools (e. g. Bauhaus [1], Columbus [8], CPPX [3], Fujaba [11], GUPRO [18], PBS [28], RPA (Philips Research), PROGRES [29], Rigi [30], Shrimp [33]). Others are going to implement tools to support working with GXL. For instance, a framework for GXL Converters [12] and an XMI2GXL translator [40] were developed at Univ. BW München. Further activities deal with providing graph query machines (GReQL, Univ. Koblenz) to GXL graphs or GXL-based graph databases (Univ. Aachen).

An important feature of GXL is its support for exchanging schema information. Based on this capability, reference schemas for certain standard applications in reverse engineering are currently under development. These activities address reference schemas for data reverse engineering (DRE, Univ. Namur, Paderborn, Victoria), the Dagstuhl Middle Model [24] or abstract syntax graph models for C++ [3], [9].

Furthermore, groups developing graph transformation tools (e. g. GenSet [15], PROGRES [29]) or graph visualization tools (e. g. GVF [19], Shrimp [33], yFiles [41]) already use GXL or pronounced to use GXL. At University of Toronto, GXL is applied within an undergraduate software engineering course to create a graph editor/layouter [5].

GXL also serves as foundation to define further graph oriented exchange formats. Thus, GXL defines the graph part in the exchange format GTXL (Graph Transformation eXchange Language) [17], [34]. Activities in the graph drawing community also deal with the development of an exchange format for graph layout [13]. In a panel on graph exchange formats at Graph Drawing 2001 in Vienna [14] GXL and GraphML [2] were discussed and compared. There is evidence of combining the structure part of GXL with the graph layout part and the modularization part of GraphML to form a general and comprehensive graph exchange format.

5 Conclusion

The previous sections gave a short introduction in the GXL Graph eXchange Language version 1.0 and its current applications.

Summarizing, GXL offers an already widely used XML sublanguage for interchanging typed, attributed, directed ordered graphs including hypergraphs and hierarchical graphs including their appropriate schemas. By focusing on graph structure, GXL provides the core for defining a family of special suited graph exchange formats.

Acknowledgment. We would like to thank the GXL co-authors Richard C. Holt, Andy Schürr, and Susan Sim for various fruitful discussions on the development of GXL, and Oliver Heinen and Kevin Hirschmann for realizing the GXL web-site and for implementing the GUPRO related GXL tools. Thanks to Rainer Koschke for many interesting discussions on interchange formats in reverse engineering, and some helpful remarks to improve this paper. Thanks to

all the users of GXL, who currently applying and testing GXL 1.0 in their tools. Their experience will be a significant aid to improve GXL.

References

1. Bauhaus: Software Architecture, Software Reengineering, and Program Understanding. `http://www.informatik.uni-stuttgart.de/ifi/ps/bauhaus/` (01.09.2001).
2. U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marschall. GraphML Progress Report, Structural Layer Proposal. In *to appear: Graph Drawing 2001 (Proceedings)*. 2001.
3. CPPX: Open Source C++ Fact Extractor. `http://swag.uwaterloo.ca/~cppx/` (01.09.2001).
4. J. Ebert, K. Kontogiannis, J. Mylopoulos: Interoperability of Reverse Engineering Tools. `http://www.dagstuhl.de/DATA/Reports/01041/` (18.04.2001), 2001.
5. S. Easterbrook. CSC444F: Software Engineering I (Fall Term 2001), University of Toronto. `http://www.cs.toronto.edu/~sme/CSC444F/` (15.09.2001), 2001.
6. J. Ebert and A. Franzke. A Declarative Approach to Graph Based Modeling. In *E. Mayr, G. Schmidt, and G. Tinhofer, editors.* Graphtheoretic Concepts in Computer Science*, LNCS 903. Springer, Berlin*, pages 38–50. 1995.
7. J. Ebert, B. Kullbach, and A. Winter. GraX – An Interchange Format for Reengineering Tools. In *Sixth Working Conference on Reverse Engineering*, IEEE Computer Society, Los Alamitos, pages 89–98, 1999.
8. R. Ferenc, F. Magyar, Á. Beszédes, Á. Kiss, and M. Tarkiainen. Columbus - Tool for Reverse Engineering Large Object Oriented Software Systems. In *Proceedings SPLST 2001*, Szeged, Hungary (`http://www.inf.u-szeged.hu/~ferenc/research/ferencr_columbus.pdf`, (01.09.2001)), pages 16–27. June 2001.
9. R. Ferenc, S. Elliott Sim, R. C. Holt, R. Koschke, and T. Gyimòthy. Towards a Standard Schema for C/C++. In *Eighth Working Conference on Reverse Engineering*. IEEE Computer Society, Los Alamitos, pages 49–58. 2001.
10. M. Fröhlich and M. Werner. daVinci V2.0.x Online Documentation. `http://www.tzi.de/~davinci/docs/` (18.04.2001), June 1996.
11. Fujaba: From UML to Java and back again. `http://www.uni-paderborn.de/cs/fujaba/` (01.09.2001).
12. GCF - a GXL Converter Framework. `http://www2.informatik.unibw-muenchen.de/GXL/triebsees/index.htm` (01.09.2001).
13. Satellite Workshop on Data Exchange Formats 8th Int. Symposium on Graph Drawing (GD 2000). `http://www.cs.virginia.edu/~gd2000/gd-satellite.html` (14.09.2001), 2001.
14. Graph Drawing (GD 2001), Vienna. `http://www.ads.tuwien.ac.at/gd2001/` (06.12.2001), September 23.-26., 2001.
15. GenSet: Design Information Fusion. `http://www.cs.uoregon.edu/research/perpetual/dasada/Software/GenSet/index.html` (01.09.2001).
16. The GML File Format. `http://www.infosun.fmi.uni-passau.de/Graphlet/GML/index.html` (18.04.2001).

17. Graph Transformation System Exchange Language. http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html (18.08.2001).
18. GUPRO: Generic Understanding of Programs. http://www.gupro.de/ (01.09.2001).
19. GVF - The Graph Visualization Framework . http://www.cwi.nl/InfoVisu/ (01.09.2001).
20. I. Herman and M. S. Marshall. Graph XML – An XML based graph interchange format. Report INS-0009, Centrum voor Wiskunde en Informatica, Amsterdam, April 2000.
21. R. C. Holt. An Introduction to TA: The Tuple-Attribute Language. http://plg.uwaterloo.ca/~holt/papers/ta.html (18.4.2001), 1997.
22. R. C. Holt, A. Winter, and A. Schürr. GXL: Toward a Standard Exchange Format. In *Seventh Working Conference on Reverse Engineering*. IEEE Computer Society, Los Alamitos, pages 162–171. 2000.
23. XMI Toolkit 1.15 (Updated on: 25.04.2000). http://alphaworks.ibm.com/tech/xmitoolkit (01.09.2001), 2000.
24. T. Lethbridge, E. Plödereder, S. Tichelar, C. Riva, and P. Linos. The Dagstuhl Middle Level Model (DMM). internal note, 2001.
25. Meta Object Facility (MOF) Specification. http://www.omg.org/technology/documents/formal/mof.htm (02.09.2001), March 2000.
26. XML Meta Data Interchange (XMI) Specification. http://www.omg.org/technology/documents/formal/xmi.htm (01.09.2001), November 2000.
27. R. Ommering, L. van Feijs, and R. Krikhaar. A relational approach to support software architecture analysis. *Software Practice and Experience*, 28(4):371–400, April 1998.
28. PBS: The Portable Bookshelf. http://swag.uwaterloo.ca/pbs/ (01.09.2001).
29. A Graph Grammar Programming Environment - PROGRES. http://www-i3.informatik.rwth-aachen.de/research/projects/progres/main.html (01.09.2001).
30. RIGI: a visual tool for understanding legacy systems. http://www.rigi.csc.uvic.ca/ (01.09.2001).
31. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, Reading, 1999.
32. A. Schürr, A. J. Winter, and A. Zündorf. PROGRES: Language and Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook on Graph Grammars: Applications, Languages, and Tools*, volume 2, World Scientific, Singapore, pages 487–550. 1999.
33. ShriMP Views: simple Hierarchical Multi-Perspective. http://www.shrimpviews.com/ (01.09.2001).
34. G. Taenzer. Towards Common Exchange Formats for Graphs and Graph Transformation Systems. In *Proceedings UNIGRA satellite workshop of ETAPS'01*. 2001.
35. Extensible Markup Language (XML) 1.0. W3C recommendation, W3C XML Working Group, http://www.w3.org/XML/ (17.04.2001), February 1998.
36. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language : Precise Modeling With UML*. Addison-Wesley, 1998.
37. A. Winter. Exchanging Graphs with GXL. In *to appear: Graph Drawing 2001 (Proceedings)*. 2001.
38. K. Wong. RIGI User's Manual, Version 5.4.4. http://www.rigi.csc.uvic.ca/rigi/ (18.04.2001), 30. June 1998.

39. Extensible Graph Markup and Modeling Language). http://www.cs.rpi.edu/~puninj/XGMML/ (19.08.2001), 2001.
40. XIG - An XSLT-based XMI2GXL-Translator. http://ist.unibw-muenchen.de/GXL/volk/index.htm (01.09.2001).
41. yFiles - Interactive Visualization of Graph Strucutres. http://www-pr.informatik.uni-tuebingen.de/yfiles/ (01.09.2001).