GXL: A Graph-Based Standard Exchange Format for Reengineering

Richard C. Holt [a] Andy Schürr [b] Susan Elliott Sim [c] Andreas Winter [d]

[a] *University of Waterloo, School of Computer Science, Waterloo N2L 3G1, Canada,* holt@plg.uwaterloo.ca

[b] *Darmstadt University of Technology, Real-Time Systems Lab, Merckstr. 25, D-64283 Darmstadt, Germany,* andy.schuerr@es.tu-darmstadt.de

[c] *University of California, Irvine, Department of Informatics, 444 Computer Science, Irvine, CA 92697-3425, USA,* ses@ics.uci.edu

[d] *University of Koblenz-Landau, Institute for Software Technology, Universitätsstraße 1, D-56070 Koblenz, Germany,* winter@uni-koblenz.de

Abstract

GXL (Graph eXchange Language) is an XML-based standard exchange format for sharing data between tools. Formally, *GXL* represents typed, attributed, directed, ordered graphs which are extended to represent hypergraphs and hierarchical graphs. This flexible data model can be used for object-relational data and a wide variety of graphs. An advantage of *GXL* is that it can be used to exchange instance graphs together with their corresponding schema information in a uniform format, i.e. using a common document type specification. This paper describes *GXL* and shows how *GXL* is used to provide interoperability of graph-based tools. *GXL* has been ratified by reengineering and graph transformation research communities and is being considered for adoption by other communities.

Key words: graph exchange language, graph-based tools, data interoperability, reengineering, XML

1 Introduction

GXL (Graph eXchange Language) is a standard format for exchanging graph-based data. It is the culmination of a cooperative effort among an international group of researchers from disparate areas, including software reengineering and graph transformation. Researchers and tool builders have had a growing interest in comparing and combining approaches to their respective problems and leveraging each other's results. These collaborations provide lessons learned

Preprint submitted to Elsevier Science *26 July 2005*

that are critical to advancing the maturity of the discipline. A standard exchange format for data facilitates tool interoperability and allows users to select the most suitable approach or tool when building a workbench.

Interoperability is the challenge of enabling tools from different suppliers to work together. Wasserman [1] describes a taxonomy with five types of interoperability: platform, presentation, data, control, and process. Another model from Earl has three levels: control, user interface, and data [2]. Data interoperability appears in both of them.

Data interoperability requires the data to be compatible both syntactically and semantically. In other words, tools need to agree on both the format and the meaning of this data. The graph-based data model of *GXL* can be used to represent both instance data and schemas. Thus, *GXL* provides a standardized notation for exchanging instance data (graphs) including their structure definition (graph schemas). Both instance and schema graphs are encoded using the same kind of XML (eXtensible Markup Language) documents [3]. While these schema graphs do not provide semantics, they serve as a basis for users to agree upon semantics. This feature is important because it helps tools and researchers communicate about the assumptions inherent in their approaches. This increased mutual understanding is a critical step in building on each other's work to increase the impact of research results.

In addition to being a generic format for representing graph structures, *GXL* is also suitable for object-relational data. Consequently, *GXL* can be used to represent data from a wider range of applications, including data repositories and factbases from reengineering tools.

Organisation of this paper

This paper is organised as follows. The next section provides background on interoperability of reengineering tools and their requirements for a standard exchange format. This background provides a motivation for the design decisions for *GXL*, including the selection of features to be included in the graph model. Section 3 describes how these features are used to represent graph-based instance data from software reengineering. The syntax of *GXL* is given by the XML DTD in Subsection 3.4. Section 4 explains how the same graph features are used to represent graph schemas. Adoption of *GXL* is outlined in Section 5 along with some examples of how *GXL* has been used successfully to facilitate data interoperability between reengineering tools. The paper concludes with a summary and a discussion of how *GXL* meets the requirements for data interoperability between reengineering tools.

2 Data Interoperability of Reengineering Tools

GXL was created to fulfill the need to exchange data between reengineering tools. Previously, interoperability between tools relied on converters between local formats. This approach requires case-by-case negotiation of exchange

syntax, schema, and even semantics. As the research area matured, it became apparent that a standard exchange format was needed and that this format should provide a mechanism to help articulate these schemas and semantics.

These experiences with interoperability and local file formats form the context for the development of *GXL*. Moreover, they circumscribe the requirements and criteria for success for a standard exchange format. In this section, we will describe this background and how it informed the emergence of *GXL*.

2.1 Interoperability of graph-based tools

A variety of reengineering tools employ graphs as an internal data representation. With improved data interoperability, reengineering workbenches can be composed by choosing the best component for a particular task. A typical reengineering workbench consists of three types of tools: Extractors, Abstractors, and Visualizers [4].

Extractors

These tools extract information from software artefacts, such as source code. Examples of such extractors for the C/C++ source language are ACACIA [5], CPPX [6], and Columbus/CAN [7]. ASIS (ADA Semantic Interface Specification) [8] offers similar functionality. Data extracted by these tools are usually exported as abstract syntax trees or graphs.

Abstractors

These components of reengineering workbenches analyze the extracted data, generating further information, and sometimes changing the form of the data. Tools of particular interest here treat the data as graphs. These tools include the PROGRES graph transformation system [9] and GUPRO [10]. General graph-based query mechanisms such as Grok and GReQL are used for analyzing graph-structured data [11]. RPA uses a relational approach to analyze software systems [12]. A generic approach for generating analyzers operating on abstract syntax trees is given e. g. in GENOA [13]. ASTLOG uses a Prolog-based environment for analyzing programs [14]. Further specialized abstractors have been developed for architectural analysis and recovery [15] [16], for control flow, data flow, and dependency analysis [17] [18], and for software metrics [19].

Visualizers

These tools display the information derived in the previous steps. This information can be visualized textually or graphically. Source code browsers are typical textual visualizers [20] [21]. Graphical visualizers have been used to display class diagrams [22], sequence diagrams [23], statecharts [24] and software architectures [25]. General graph drawing tools – for instance daVinci [26], Graphlet [27], and GraphViz [28] – have been used to visualize small-

and medium-sized graphs. Large complex graphs are better handled by visualization tools designed for reengineering, such as Rigi [25] and SHriMP [29].

2.2 Collaborating tool sets

The approaches and tools shown above, provide good support for various aspects of reengineering. Individual tools from different workbenches have been combined to tackle a range of reengineering challenges. Here are some illustrative examples of the data exchange. More of them can be found at [30].

- Acacia and PBS. Acacia is a tool kit by AT&T Labs [31] for analyzing and visualizing programs written in C++. There is a command line interface that allows extraction of facts about an parsed C++ program into Acacia's database. The analysis is at the external declaration level. Acacia was used to extract facts from the Mozilla source code [32] that were subsequently converted into a corresponding TA stream (Tuple Attribute Language) [33] and analyzed using PBS (Portable Bookshelf) [34] tools.
- Dali and SNiFF+. The Dali reverse engineering tool kit was created by the Software Engineering Institute [35], [36]. This tool kit combines features from a number of tools. To analyze the Linux kernel, SNiFF+'s API [37] was used as a fact extractor. These facts were stored in TA [33], analyzed using a relational database, and viewed using Rigi [38].
- CPPAnal and GUPRO. The CPPAnal tools by Harry Sneed [39] extract source code information on an architectural level from large software systems and store them in SQL tables. By using GraX [40] these tables are transferred into TGraphs [41] for further analysis with GUPRO tools from University of Koblenz [42]. These TGraphs are used in GUPRO to browse large graphs [43].

All of these collaborations were made possible through converters that take files from one local format and transform the data into another local format. While this approach has been used successfully, it does not scale well. In other words, a converter would need to be written for each *pair* of local data formats and this effort quickly becomes unmanageable. A standard exchange format serves as an intermediary for these file formats; tool developers would only need to convert to and from the local format and the exchange format.

There are a number of data formats that are used internally in a reengineering workbench, such as RSF [38] and GraX [40]. These formats, while efficient, are not suitable as an *exchange format* because they have different underlying graph models, are optimized for particular analyses, and frequently contain artefacts that reflect the tool internals. For instance, RSF can represent hierarchical graphs that are not supported by GraX. By the same token, GraX provides extensive support to represent and exchange the structure of graphs by schema graphs. A standard exchange format was needed that is flexible and

general enough to represent the most common representations of data from software systems. Such a format was required for interoperability of different reengineering tools to support exchange of data without loss of detail.

GXL provides such a *common and generally applicable format* for interchanging data on software systems between Extractors, Abstractors, and Visualizers, as well as other tools used to support software evolution.

2.3 Requirements for a standard exchange format

Examination of the collaborations in the previous subsection and further analysis of data interoperability [30] [44] [45] [46] provide insights into the problem of standard exchange formats. These in turn lead us to the following requirements for such an exchange format in reengineering: universality, typing, flexibility, ease of use, scalability, modularity, and extensibility.

Universality: A standard exchange language shall support data exchange for multiple purposes. In a reengineering exchange format, this includes exchanging data about different programming languages and at different levels of abstraction, ranging from fine-grained representations such as abstract syntax trees and more coarse-grained representations such as architectural descriptions. A standard exchange format needs to be flexible enough to be an intermediary in these and other situations.

Typing: A standard exchange language shall be typed. Knowing the types of objects being exchanged makes it easier to interpret the exchanged data. Typed exchange languages also permit validation of exchanged data and allow adaption to problem-specific data exchange. Defining types for data and their interdependencies helps in standardizing domain-specific exchange models.

Flexibility: A standard exchange language shall be flexible. It should be easily adaptable to exchanging domain specific data (cf. typed language) to provide far reaching use. Furthermore, it must allow annotations on all kinds of data objects, e. g. layout information, source code references, and metrics.

Ease of Use: A standard exchange language shall be designed to provide easy tool implementation. These tools include import and export filters, translators from and to other formats, and helpers to validate and ensure the integrity of exchanged data.

Scalability: A standard exchange language shall cope with data software systems independently from their level of granularity. It has to scale for data of arbitrary magnitude. Software systems in reengineering can be quite large, sometimes consisting of millions of lines of source code, leading to abstract syntax trees. Thus, the standard exchange language and the supporting tool sets have to deal with a large amount of data, efficiently.

Modularity: A standard exchange language shall support modular and incremental data exchange, so that data can be separated, hidden, or shared

as needed. In other words, it should be possible to exchange data sets in parts, as subsystems, or in multiple documents.

Extensibility: A standard exchange format shall provide support for extending the modeling concepts used by specialized versions of the exchange language. Extensibility allows the exchange format to be used in additional domains, through the addition of new elements or through the use of the format as a sublanguage.

These requirements for a standard exchange format for reengineering provided the starting point for our design decisions in creating GXL. In the next section, the requirements are mapped to specific features in the format.

2.4 Graph exchange formats in reengineering

The examples of collaborating tool sets in Section 2.2 demonstrate the need for a general and applicable exchange format for reengineering data. These tools typically use object-relational or graph-based file formats. The underlying data model in the standard exchange format needs to be robust and flexible enough to act as a bridge between myriad existing formats. Thus, a widely applicable *lingua franca* in reengineering needs to be an *adaptable, graph-based format.* The high-level requirements on exchange formats presented in Section 2.3 motivate decisions on more technical requirements for the suggested reengineering exchange format. In this section, we relate those requirements to specific design decisions regarding features in *GXL*.

We decided to create a new format rather than use an existing one because

- we needed a format that is simultaneously compatible with as many of these as possible,
- has only and all the necessary graph features,
- is flexible enough to work with disparate data and different levels of abstraction, and
- is simple.

To ensure on ease on use, specifically ease of implementation, we decided to use XML. This standard for semi-structured data allows us to define our own format, while at the same time taking advantage of XML infrastructure for constructing tools. One repercussion of this decision is the size of the files being exchanged (cf. scalability). These files will be larger due to XML syntax and the length of tag and attribute names. However, this is a problem faced by all XML users and standard compression techniques are effective remedies due to the amount of repetition in the files.

In addition to gathering requirements for a standard exchange format, we analyzed a number of existing formats. This investigation identified both the kinds of features we should support and different approaches to satisfying our requirements. The formats that we studied included the internal representations of tools in software engineering and reengineering (e. g. ATerms [47], DiaGen

[48], GraX [40], RPA [12] RSF [38], TA [33]), in graph databases (e. g. PROGRES [9]), and in graph drawing (e. g. daVinci [26], dot [28], GML/Graphlet [49], GRL [50], XGMML [51], GraphXML [52]). From this review, we identified nine features that we included in *GXL*. These features are described below:

Graph elements: Basic graph elements like nodes, directed and undirected edges and attributes must be supported. For maximal flexibility, we permit both directed and undirected edges in the same graph.

Hyperedges: *N-ary relationships* (hyperedges) must be supported natively. Tools or formats that use hyperedges need to be able to use the exchange format as well. Mapping n-ary relationships onto special nodes and binary edges is an unsatisfactory work-around that does not provide equivalent structural characteristics.

First class elements: Nodes, edges, and hyperedges must be identifiable *first-class elements*, or objects, such that they can have unique identifiers. Viewing edges as first class elements treats them equal to nodes and enables multiple edges between nodes.

Attributes: All graph elements may have attributes added to them. This also includes the attributes themselves, e. g. to express layout features of attributes.

Ordering: Ordering of incidences, i. e. the order of edges incident on a node, must be available such that ordered lists of parameters or declarations can be conveniently expressed.

Hierarchy: Hierarchical graphs must be supported to provide simple substructuring of graphs. Subgraphs may be exchanged as separate documents.

Graph schemas: The format must be able to define graph classes, or schemas. These are needed to constrain the form of graphs used in different domains of application. These graph schemas permit the specification and use of types.

Extension Points: The exchange language syntax has to be extensible, so that the format can be easily adapted to other areas. Furthermore, extension points must be available to permit enhancement of the language.

Simplicity: The exchange format has to be simple, so it can be read and understood by humans. This feature is achieved through a document type definition with a modest number of elements and corresponding exchange documents that are also small.

Figure 1 lists graph-based representations that we studied and their support for nine features. It shows that a graph exchange format which supports all required features in one common language does not exist. *GXL* integrates these features in a general graph model (cf. Section 3.4). Additionally, *GXL* is adaptable because it supports metamodel-based definition of graph classes (cf. Section 4) and extensions to the language (cf. Section 3.4).

Figure 2 illustrates how the features selected for *GXL* fulfills the requirements identified section 2.3. Every feature satisfies at least one requirement and every requirement is met by a feature. The universality requirement was achieved by

Format	Feature								
	Graph elements	Hyper-edges	First class elements	Attributes	Ordering	Hierarchy	Graph schemas	Extension points	Simplicity
Software Engineering and Reengineering									
Aterms [47]	•[1]			•		•			
DiaGen [48]	•	•	•	•					
GraX [40]	•		•	•	•		•		•
RPA [12]	•[2]		•[3]			•			•
RSF [38]	•		•	•		•	•		•
TA [33]	•		•[3]	•			•		•
Graph Transformation									
PROGRES [9]	•		•[3]	•[4]			•		
Graph Drawing									
daVinci [26]	•[1]			•		•	•[5]		
dot [28]	•		•	•		•			•
GML [49]	•		•	•				•	•
GraphML [53]	•	•	•	•		•		•	
GraphXML [52]	•		•	•		•		•	
GRL [50]	•		•	•			•[5]		
XGMML [51]	•	•	•	•		•			

[1] Aterms and daVinci are based on terms.

[2] RPA is based on sets and relations.

[3] Only nodes are viewed as first class elements.

[4] Only nodes can be attributed.

[5] Nodes and edges may have types, but an explicitly defined schema is not supported.

Fig. 1. Supported features in graph formats

including graph elements, hyperedges, attributes, and ordering. This collection of features in the *GXL* graph model ensured compatibility with a large number of graph formats. These features were also considered primitive because they could not be achieved through the combination of other features. The typing requirement was implemented through graph schemas. While graph schemas appeared in only a few formats, their expressive power and flexibility made them an attractive mechanism for supporting typing. Flexibility was further achieved in *GXL* with user-defined attributes and extension points. The Ease of Use requirement was satisfied through simplicity and the decision to use XML. Scalability was enabled using the Hierarchy and Simplicity features. Modularity was implemented through the Hierarchy feature and some XML features. Finally, Extensibility was realized using Extension Points.

In addition to *GXL*, there are other XML-based formats for exchanging graphs or software artefacts in software engineering and reengineering. GraphML [53] is a graph exchange format oriented towards graph layout which succeeds GraphXML. GraphML offers a core graph model similar to *GXL*. Whereas adaptability of *GXL* is based on metamodeling technology for defining convenient graph schemas (cf. Section 4), adaptability of GraphML is given by extending the GraphML document definition. Thus, GraphML documents use different, domain-specific document definitions with a common core. In contrast, *GXL* uses one common, application-independent document definition.

GXL Features	Requirements for Exchange Formats						
	Universality	Typing	Flexibility	Ease of Use	Scalability	Modularity	Extensibility
Graph elements	•						
Hyperedges	•						
First class Elements	•						
Attributes	•		•				
Ordering	•						
Hierarchy	•				•	•	
Graph schemas		•	•				
Extension points			•				•
Simplicity				•	•		

Fig. 2. Requirements for graph-based exchange languages

Another relatively minor difference is that GraphML has the concept of ports. Ports are properties of nodes and are convenient for controlling the incidence of edges. Since ports can be mapped onto existing concepts in *GXL*, we elected to not add another feature to support them. For example, one possible way to represent ports in *GXL* is by using *edge attributes* to indicate the port's name or *ordering* to indicate numbered ports.

The graph drawing community has compared the two approaches and found that they are compatible [54]. As part of this exercise, a set of filters for converting between GraphML and *GXL* were developed.

Exchange of graph-based data can (also) be accomplished using MOF (Meta-Object Facility) [55] as modeling language and XMI (XML Metadata Interchange) [56] as exchange language. While MOF does not have native concepts for modeling graphs, e. g. n-ary relationships, graph properties, and link attributes, it may be used to define a graph modeling and exchange language. Once this graph model has been defined, XMI can be used to generate an XML document specification. MOF/XMI and *GXL* are similar in that both are generic exchange languages based on metamodeling technology. However, there are three important differences.

One, the document type specifications created by MOF/XMI are complex and contain a large number of XML elements that are not directly relevant to encoding the data, e. g. elements for CORBA compatibility. The *GXL* document specification was implemented by creating a UML class diagram that defines the underlying graph model and then manually deriving the elements and attributes. Consequently, the design of *GXL* is much cleaner and requires only a small number of elements (cf. see Section 3.4 for more details).

Two, each distinct MOF metamodel generates a new document type specification. The advantage is the XML format that is created is specially tailored for specialized exchange scenarios. The disadvantage is that tools then must also

be tailored for each XML format generated. Here, *GXL* provides one common XML notation for exchanging variants of graph-based data.

Finally, in MOF/XMI two different syntaxes are used for representing instance data and for exchanging the metamodels of that data. The metamodels are stored as document type specifications, that is, DTDs or XML Schemas. Instance data are represented as XML documents in a variety of notations. In contrast, *GXL* uses a single common document type specification (cf. Section 4) for graph instances and graph schemas, which simplifies the task of developing tools to work with the format.

Other XML-based approaches (cf. [57], [58]) to storing, analyzing, and exchanging program data make use of the tree structure inherent in XML documents, that is, DOM trees. XML tags are added to source code so that the structure of the XML document mirrors the abstract syntax tree. Consequently, when the XML document is parsed, the parse tree for the program is re-created. These approaches differ from *GXL* because they require different document type specifications for different languages and they are restricted to tree like structures. Despite these differences, both the XMI and the DOM approaches are both based on XML, so data can be interchanged with *GXL* by using appropriate XSLT scripts.

In summary, *GXL* seeks to be a *general, compact and simple graph-based exchange format.*

2.5 Genealogy of GXL

The genealogy of *GXL* presented in this section shows how *GXL* matured and how other graph formats in reengineering and graph technology influenced the development of *GXL*. The genealogy of *GXL* is depicted in Figure 3.

Development of *GXL* began with a merger of *GRAph eXchange format (GraX)* [40], *Tuple Attribute Language (TA)* [33], and the file format from the *PROGRES* graph rewriting system [9] introducing the general graph features. This collection was presented in *GXL* 0.4.2 for comment by the general community. Criticisms and suggestions directed us to consider including features from a broader collection of formats.

The development of *GXL* was advanced during various conferences and workshops since 1998. Initial discussions on defining a general exchange format for reengineering tools were held at WCRE 1998 [59] and at CASCON 1998 [60]. Approaches for graph-based exchange formats were discussed during meetings at WCRE 1999 [61], and GROOM 2000 [62]. These interactions and investigations resulted in an initial prototype of *GXL* that was presented at the ICSE 2000 Workshop on Standard Exchange Formats (WoSEF) [63]. This proposal was subsequently discussed, compared, and critiqued at meetings on exchange formats at APPLIGRAPH [64] and Graph Drawing [65]. Refinements of the prototype were presented at conferences and workshops throughout 2000, in-

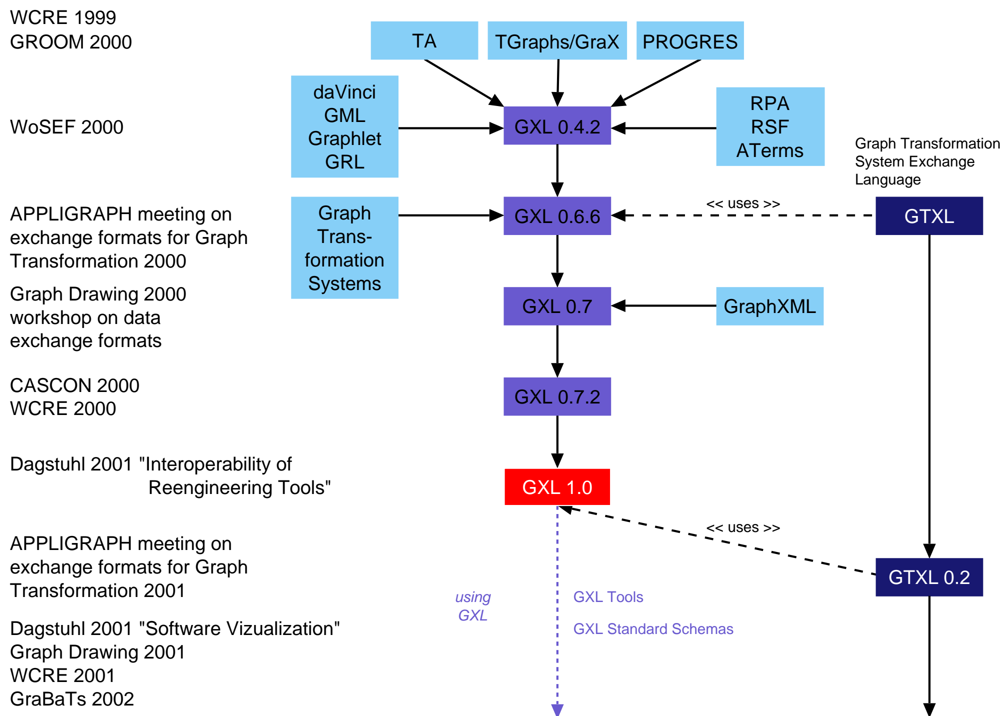

Fig. 3. Genealogy of *GXL*

cluding CASCON 2000 [66], [67] and WCRE 2000 [68]. *GXL* was ratified as a standard exchange format in software reengineering at the Dagstuhl Seminar "Interoperability of Reengineering Tools" in January 2001 [69].

Soon afterwards, *GXL* was presented at meetings in other research areas. The graph transformation community is using *GXL* as a starting point for the *Graph Transformation Exchange Language* (GTXL) [70] [71]. In this context *GXL* is being used to represent graphs and work is under way to add features for representing transformation rules. This decision was made after the APPLIGRAPH meetings for exchange formats [64] and the GraBaTs Workshop on Graph-Based Tools [72]. Discussions have been held with the graph drawing community to make *GXL* a standard exchange format for graph layouts as well. Presentations were made at GD2000 [65] and a panel held at GD2001 [73].

Since *GXL* specifies only graphs, it remains to standardize schemas to further describe what these graphs represent. In other words, standard schemas, or reference schemas, are needed for being fully interoperable to data interchange. While this approach can be said to merely shift the debate from syntax to semantics, it is a desirable change because it raises negotiations about interoperability to a more conceptual level. This level of abstraction is one that is properly in the realm of discourse for research as it is more likely to lead to breakthroughs in understanding.

The current version of *GXL*, news about ongoing development efforts, and up-to-date information including tutorials and documentation are available at `http://www.gupro.de/GXL`.

3 Exchanging Graphs with *GXL*

In the previous section, we argued that a graph-based standard exchange format is appropriate for reengineering. In this section, we discuss the specific graph features included in *GXL* and how these can be used to represent software.

GXL supports graphs which can have directed or undirected edges, typed nodes and edges, attributes attached to nodes and edges, and ordered edges [41]. Section 3.1 illustrates the use of these *GXL* features. To this set of features, *GXL* adds n-ary edges (hyperedges) as well as hierarchical graphs (subgraphs within graphs). Sections 3.2 and 3.3 illustrate the use of these features. Finally, the *GXL* language definition is given in Section 3.4 using an XML document type definition (DTD).

3.1 Exchanging typed, attributed, directed, ordered graphs

Figure 4 shows a fragment of source code along with its abstract syntax graph, which we depict using UML object diagram notation [74]. The diagram is at the level of an abstract syntax graph. In the program, function *main* calls function *max* in line *8* and function *min* in line *12*.

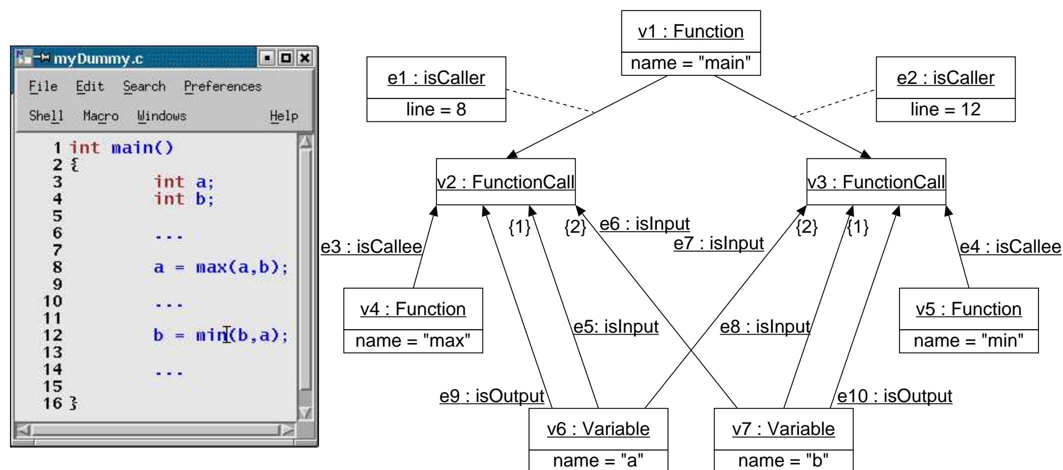

Fig. 4. typed, attributed, directed, ordered graph

In the diagram, functions *main*, *max*, and *min* are represented by nodes of type *Function*, while variables *a* and *b* are represented by nodes of type *Variable*. These nodes are attributed with the names of the functions and variables.

The calls to functions *max* and *min* are represented by *FunctionCall* nodes. These nodes are associated with the caller by *isCaller* edges and with the callee by *isCallee* edges. The *isCaller* edges are attributed with a *line* attribute giving the line number that contains the call. Parameters (represented by *Variable* nodes) are associated with function calls by *isInput* edges. The ordering of parameter lists is given by the ordering incidences of *isInput* edges

```
<?xml version = "1.0" ?>
<!DOCTYPE gxl
    SYSTEM "gxl-1.0.dtd">
<gxl xmlns:xlink="www.w3.org/1999/xlink">
<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 = "v1" to = "v2">
    <type xlink:href =
    "schema.gxl#isCaller"/>
    <attr name = "line" >
      <int>8</int>
    </attr>
  </edge>
  <edge id = "e2"
    from = "v1" to = "v3">
    <type xlink:href =
    "schema.gxl#isCaller"/>
    <attr name = "line" >
      <int>12</int>
    </attr>
  </edge>
  <edge id = "e3"
    from = "v4" to = "v2">
    <type xlink:href =
    "schema.gxl#isCallee"/>
  </edge>
  <edge id = "e4"
    from = "v5" to = "v3"
    <type xlink:href =
    "schema.gxl#isCallee">
  </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. 5. *GXL* representation of graph from Figure 4

pointing to *FunctionCall* nodes.[1] The first edge of type *isInput* incident to function call *v2*, for the call *max(a,b)*, comes from node *v6* representing variable *a*. The second edge of type *isInput* comes from the second parameter *b* (node *v7*). The ordering of the parameters of the other call (*v3*) are represented analogously.

GXL provides constructs for exchanging graphs such as the one in Figure 4. These constructs represent nodes, edges, and edge ordering, as well as type information and attribute values.

Figure 5 depicts the graph from Figure 4 as an XML document following the *GXL* structure. The second and third lines of Figure 5 give the DTD version for *GXL* as gxl-1.0.dtd. The body of the *GXL* document is enclosed in <**gxl**> tags. The fifth line gives the name of the graph as simpleGraph and specifies that edges are to have identifiers, such as e5. Next, the graph refers to its associated graph schema named Schema (cf. Section 4) stored in file schema.gxl.

[1] In contrast to UML, which orders adjacencies, *GXL* uses ordering of incidences.

Nodes and edges are represented by <**node**> and <**edge**> elements. These can be located by their id attribute. Incidence information of edges including edge orientation is stored in from and to attributes within <**edge**> tags. Ordering of incidences is also represented here. Attributes fromorder and toorder represent the order 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.

The <**node**> and <**edge**> elements may contain further attribute information. The <**attr**> elements describe attribute names and values. For compatibility with tools using typed attributes, *GXL* also offers typing of attributes. Usually, this information is defined within the schema of a given graph class (cf. section 4). But, since *GXL* is not constrained to use graph schemas, attribute types are specified within the instance documents by appropriate tags. Using schemas, additional constraints ensure that these attribute tags match the schema specification. Like OCL [75], *GXL* provides <**bool**>, <**int**>, <**float**>, and <**string**> attributes. Furthermore, enumeration values (<**enum**>) and URI references (<**locator**>) to externally stored objects are supported. *GXL* offers composite attributes including sequences (<**seq**>), sets (<**set**>), multi sets (<**bag**>), and tuples (<**tup**>). <**Attr**> elements only contain one data element, e. g. <**int**> or <**set**>. But, they may contain other <**attr**> elements to exchange attributes of attributes.

3.2 Exchanging Hypergraphs

GXL supports *hypergraphs* [76] (graphs with n-ary edges) as well as graphs with binary edges. These n-ary edges can be typed, attributed, directed or undirected and ordered.

Figure 6 shows a hypergraph in UML notation, modeling the function call $a = max(a, b)$ by a 5-ary hyperedge of type *FunctionCall2*. The diamond, representing the hyperedge, is connected by lines (tentacles) to its related *Function* and *Variable* nodes. These tentacles are marked with roles, identifying *caller*, *callee*, *input*, and *output*. Numbers on the tentacles give the ordering of parameters. The hyperedge has a *line* attribute giving its line number as *8*.

The *GXL* representation of this hyperedge is given in Figure 7. Hyperedges are represented by <**rel**> (relation) elements. Like <**node**> and <**edge**> elements, <**rel**> elements can contain type (<**type**>) and attribute (<**attr**>) information. Tentacles, which point to the related graph objects (target), are represented by <**relend**> (relation end) subelements. Roles of tentacles are stored in role attributes. The ordering of tentacles at the hyperedge is given by startorder attributes. The ordering of tentacles at target objects is given by endorder attributes. Directed or undirected hyperedges and tentacles are distinguished by attributes isdirected and direction.

Edges, which are inherently binary, can be represented as 2-ary hyperedges.

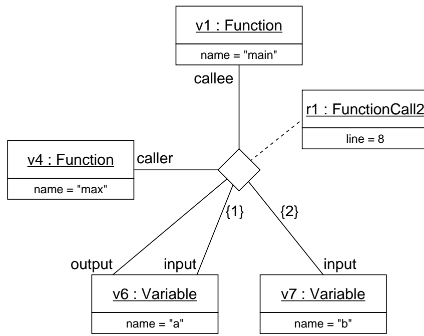

Fig. 6. Hypergraph

```
...
<rel id = "r1" >
   <type xlink:href = "schema2.gxl#FunctionCall2"/>
   <attr name = "line" >
      <int>8</int>
   </attr>
   <relend target = "v1" role = "callee" />
   <relend target = "v4" role = "caller" />
   <relend target = "v6" role = "output" />
   <relend target = "v6" role = "input"
            startorder = "1"/>
   <relend target = "v7" role = "input"
            startorder = "2"/>
</rel>
...
```

Fig. 7. *GXL* representation

This means that *GXL* does not need to support edges explicitly. However, since binary edges are so common, *GXL* provides a special notation <**edge**> for them.

3.3 Exchanging Hierarchical Graphs

Although graphs are intuitive and convenient, when large, they become complex to manage and to visualize. This complexity can be reduced by introducing subgraphs, in which parts of graphs representing related objects are grouped into subgraphs. The resulting *hierarchical graphs* [77] support structuring of graphs by grouping and encapsulation.

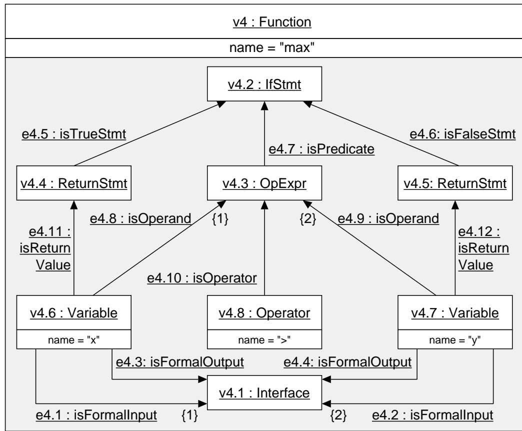

Fig. 8. Hierarchical Graph

```
...
<node id = "v4" >
   <type xlink:href = "schema.gxl#Function" />
   <attr name = "name" >
      <string>max</string>
   </attr>
   <graph id = "g4" >
      <type xlink:href = "asg.gxl" >
      <node id = "v4.1" >
         <type xlink:href = "asg.gxl#Interface" >
      </node>
      ...
      <edge id = "e4.12"
         from = "v4.7" to = "v4.5"/>
         <type xlink:href =
            "asg.gxl#isReturnValue"/>
      </edge>
   </graph>
</node>
...
```

Fig. 9. *GXL* representation

Figure 8 gives an example of a hierarchical graph. Node *v4*, which represents the *max* function from Figure 4, contains a subgraph representing *max*'s function body. The *GXL* representation in Figure 9 shows this subgraph as a <**graph**> element inside node *v4*. Subgraphs inside edges or hyperedges are written analogously (cf. the *GXL* DTD in Section 3.4).

The *GXL* form of hierarchical graphs is convenient when there is a strong sense of ownership that can be modeled by nesting of graphs. But, *GXL* also

permits edges and hyperedges crossing the boundaries of graph hierarchies up, down, diagonally, and sideways. Consequently, edges can be used to connect subgraphs and graph elements from any level in the hierarchy. No restrictions have been placed in these hierarchical edges and hyperedges to permit the greatest flexibility when using hierarchical graphs.

GXL provides one explicit form for graph hierarchies. There are alternate approaches to modeling them. For example, references to subgraphs and their elements may be represented using <**locator**> attributes pointing to their appropriate *GXL* representations. This approach does not support connectivity between sub- and supergraphs. Since locator attributes usually refer to external documents, the subgraph is only visible from the supergraph, and not vice versa.

3.4 GXL DTD

This section introduces the structure of *GXL* as XML notation. It begins this by giving a UML class diagram that defines the kind of graphs provided by *GXL*. This serves as a starting point for specifying *GXL*'s DTD [3] and XML Schema definition [78].

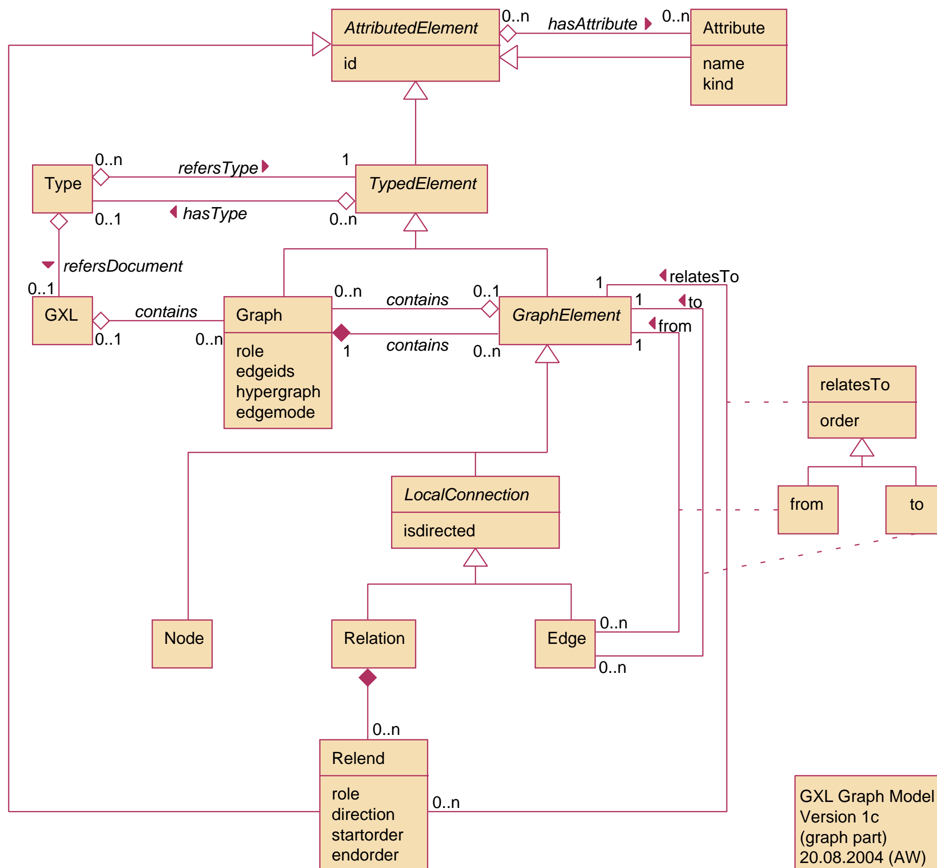

Fig. 10. *GXL* Graph Model

The class diagram in Figure 10 specifies all graph features supported by *GXL* (cf. Figure 2). The diagram omits the classes for the portion of *GXL* for representing attributes and associated data types. As the figure shows, a *Graph* contains *GraphElements*, which are *Nodes*, *Relations*, and *Edges*. To support hierarchical graphs, each *GraphElement* may contain other *Graphs*. *Edges* record binary connections and *Relations* record n-ary connections between *GraphElements*. Note, that *GXL* allows edges and hyperedges to make connections between other edges and hyperedges as well as between nodes. Ordering of incidences is stored in *order* attributes of *relatesTo* associations. *Graphs* and *graph elements* can be typed and attributed. Graph types are defined by graph schemas represented as *GXL* documents (cf. Section 4). This set of entities with their interrelationships means that *GXL* defines *typed, attributed, directed, ordered, hierarchical graphs and hypergraphs.*

The user writes *GXL* graphs as XML documents. Therefore, it is convenient to specify the syntax of *GXL* as an XML document type definition or as an XML schema definition. To keep this definition simple and understandable, it was created manually, basically by translating Figure 10 into DTD and XML schema notation. Figure 10 shows the resulting document type definition in its entirety. A commented version of this DTD and a corresponding XML Schema are available at `http://www.gupro.de/GXL`. The handcrafted *GXL* DTD has only 18 XML elements. In contrast, a DTD for *GXL* generated using IBM's XMI (XML Metadata Interchange) Toolkit [79] requires 66 elements for the *GXL* core and an additional 63 elements for XMI and CORBA compatibility.

The *GXL* DTD (see Figure 11) begins by specifying predefined points (cf. [80]) for extending *GXL*. These lines can be used to add sub-elements or attributes to their corresponding graph elements. The rest of the DTD gives the syntax for graph components (<**graph**>, <**node**>, <**edge**>, <**rel**>, <**relend**>), attributes (<**attr**>), and references (<**type**>) to schema information.

To keep the language design of *GXL* simple, *GXL* did not use the XML schema mechanism for data types [81] provided for attributes. Instead *GXL* used special tags for simple types (<**bool**>, <**int**>, <**float**>, <**string**>, <**enum**>) and nesting of tags for composite types (<**seq**>, <**set**>, <**bag**>, <**tup**>). The composite types of sequences, sets and multisets (bags) are expected to be homogenous. However, tuples can hold data of different types.

XML DTDs impose syntactic constraints on documents, but the semantic constraints that it can impose are limited. Some semantic constraints in *GXL*, such as "<**edges**> and <**rel**> elements only connect elements of the given graph", can be enforced within XML, using the referencing mechanism for identifiers (i. e. ID, IDREF). The more restrictive *GXL* constraint that these references are only allowed to refer to graph elements (and not attributes), can not be expressed or enforced using only XML. Additional constraints such as the ones listed below must be defined outside the DTD:

- Edges and hyperedges only connect graph elements. Each IDREF pointing to incident graph elements refer only to <**node**>, <**edge**>, and <**rel**> elements.

```
<!– 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 (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. 11. GXL Document Type Definition (DTD)

- Edges and hypergraphs only connect graph elements within the same graph. Each IDREF pointing to incident graph elements has to refer to a graph element, which is defined within the same <**graph**> element (including subgraphs) or within a <**graph**> element representing the convenient supergraph.
- Attribute identifiers have to be unique for each graph element. Each <**node**>, <**edge**>, and <**rel**> element does not contain multiple <**attr**> elements with the same name.

- Ordered incidences have to be linear. All fromorder/toorder attributes of <**edge**> elements and all startorder/endorder attributes of <**relend**> elements, respectively have to define a proper ordering according to their incident graph elements. No fixed lower bound or initial index is prescribed.

A detailed list of constraints has been published separately and a *GXL* validator suite has been made available for checking that documents conform to these constraints [82].

4 Exchanging Graph Schemas

Graphs are used for describing objects (nodes) and their interrelationships (edges, hyperedges). In a particular application domain, it is commonly appropriate to constrain the form of the graph, for example by limiting the types of the nodes. A schema provides a means for describing and constraining the graph. In particular, a schema determines:

- which node, edge, and hyperedge classes (types) can be used;
- which relations can exist between nodes, edges, and hyperedges of given classes;
- which attributes can be associated with nodes, edges, and hyperedges;
- which graph hierarchies are supported; and
- which additional constraints (such as ordering of incidences, degree restrictions) have to be imposed.

These constraints specialize the graph structure to represent the domain of interest.

4.1 GXL schemas as UML class diagrams

This section explains how *GXL* schemas are written and used. We start by giving three example schemas, i) the schema in Figure 12 for use with the simple graph in Figure 4, ii) the schema in Figure 13 for use with the hypergraph in Figure 6, and iii) the schema in Figure 14 for use with the hierarchical graph in Figure 8. We also show how *GXL* schemas are exchanged using a particular form of a graph. The next section after this one shows how this format is itself described by another schema (by a metaschema).

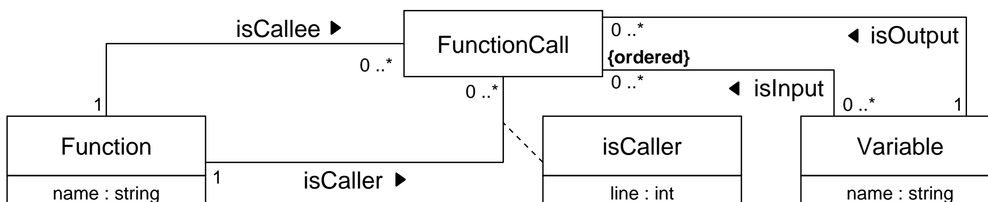

Fig. 12. Simple Schema Graph

As illustrated in Figure 12, Figure 13 and Figure 14, we can represent *GXL* schemas as UML class diagrams [74]. Each node, edge, or hyperedge of a particular type in the instance graph has a corresponding class in the schema diagram. The schema in Figure 12 has classes representing node classes (*FunctionCall*, *Function*, and *Variable*) used in Figure 4, and it has associations (*isCaller*, *isCallee*, *isInput*, and *isOutput*) representing edge classes. The edge class *isCaller* has an integer attribute named *line*, which reflects the fact that in Figure 4, *isCaller* edges are attributed with line numbers. The orientation of edges is depicted by a filled triangle [74, p. 155]. Multiplicities denote degree restrictions. Ordering of *incidences* is indicated by the keyword {ordered}.

The schema for the hypergraph in Figure 6 is given by Figure 13. The hyperedge's class is shown in Figure 13 as a diamond with attached tentacles. These tentacles can be annotated by multiplicity information to specify cardinalities, and by names indicating the roles of participating classes. The keyword {ordered} can be used to require ordering of incident tentacles in instance graphs. Attributes of hyperedge classes are defined within an associated class attached to the diamond representing the hyperedge class.

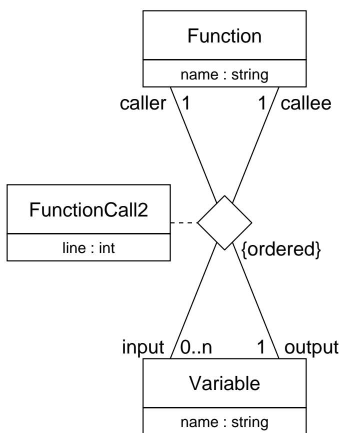

Fig. 13. Hypergraph Schema

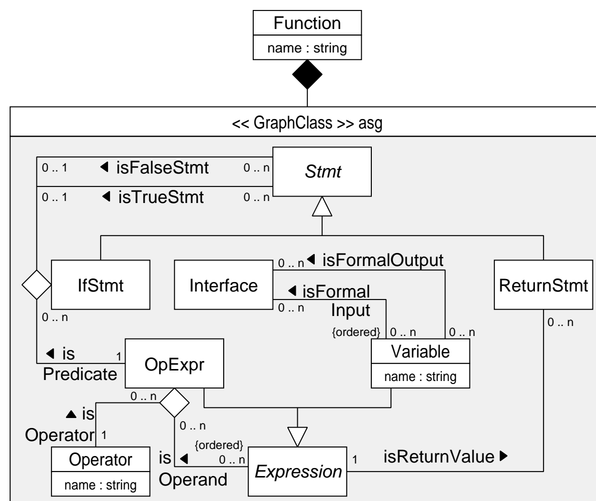

Fig. 14. Hierarchical Graph Schema

The schema for the hierarchical graph in Figure 8 is given by Figure 14. This schema uses a UML stereotype <<GraphClass>> to distinguish classes containing subgraphs from (ordinary) node classes. Composition (depicted by nesting or filled diamonds) is used, to define ownership of graph classes and containment of graph objects within a graph class. By convention, nesting is used to describe graph class definition and filled diamonds express ownership. The specification of graph class *asg* is nested within the <<GraphClass>> node. Nodes of class Function own graphs of graph class *asg* (abstract syntax graph). The definition of graph class *asg* also shows the use of higher modeling constructs like generalization and aggregation.

4.2 GXL schemas represented as graphs

GXL provides a great deal of flexibility in the handling of various kinds of data, by allowing the user to transmit a graph's schema along with the graph itself. This is done by translating the schema so it becomes an ordinary graph and encoding this graph in *GXL* the same manner as any other graph.

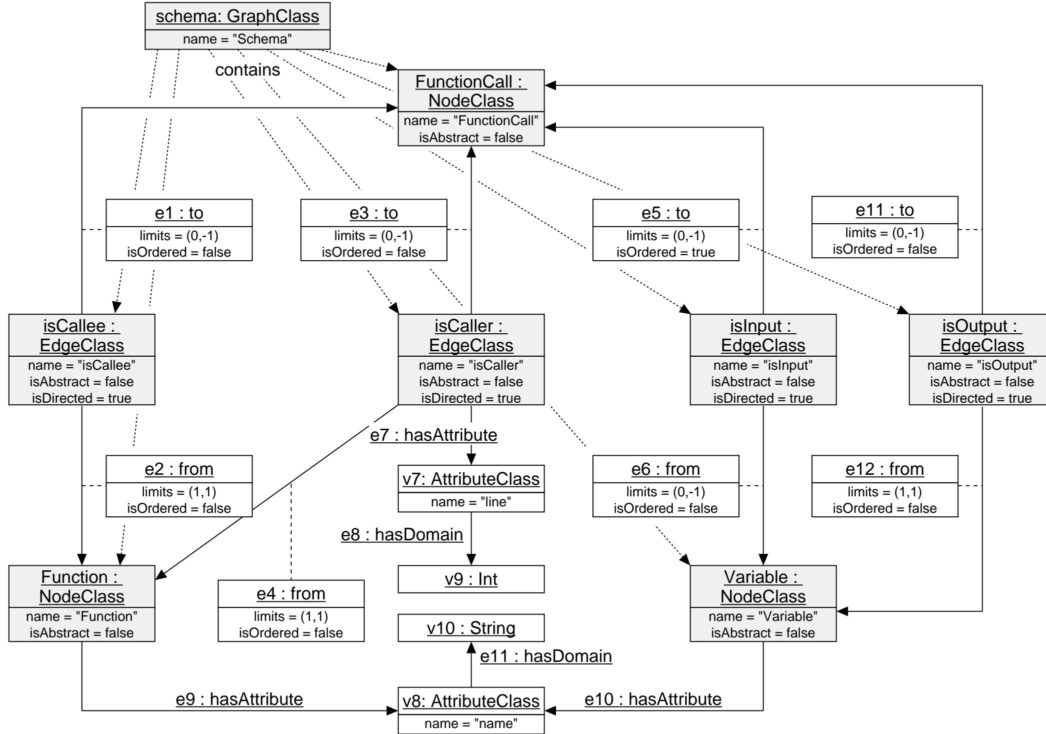

Fig. 15. Graph for schema in Figure 12

Figure 15 shows the result of translating the schema in Figure 12 into a graph. Each node class is translated to a corresponding *NodeClass*-node, for example, the *Function* node is translated to a *NodeClass* named *Function*. Each edge class is translated to a corresponding *EdgeClass*-node, for example, the *isCaller* edge is translated to the node *isCaller* of type *EdgeClass*. The connections of *isCaller* node and edge classes are translated into edges of type *from* and *to*.

Similarly, attributes or attribute types are translated to *AttributeClass*-nodes and appropriate attribute type nodes like *Int* or *Set*. Attribute information are connected to node and edge class representations by *hasAttribute* and *hasDomain* edges. Multiplicities of associations are stored in limits attributes (infinity is represented by -1). The boolean attribute isOrdered indicates ordered incidences. Attribute types and extended concepts such as graph hierarchy, classes of hyperedges, aggregation and composition, generalization and default attribute values are modeled analogously.

Each schema has a node of type *GraphClass* which is attached by *contains* edges to all nodes which represent elements of the schema (see Figure 15).

This node is referred to by data graphs which use this schema. Elements in a data graph refer to corresponding nodes in their schema graph. The *GXL* Validator [82] checks that data graphs conform to their schemas.

4.3 GXL metaschema

Every *GXL* schema is translated into a graph with the same form. In other words, there is a single *GXL* metaschema that gives the format of all *GXL* schemas. The class diagram in Figure 16 shows this *GXL* metaschema (except for the part defining attributes).

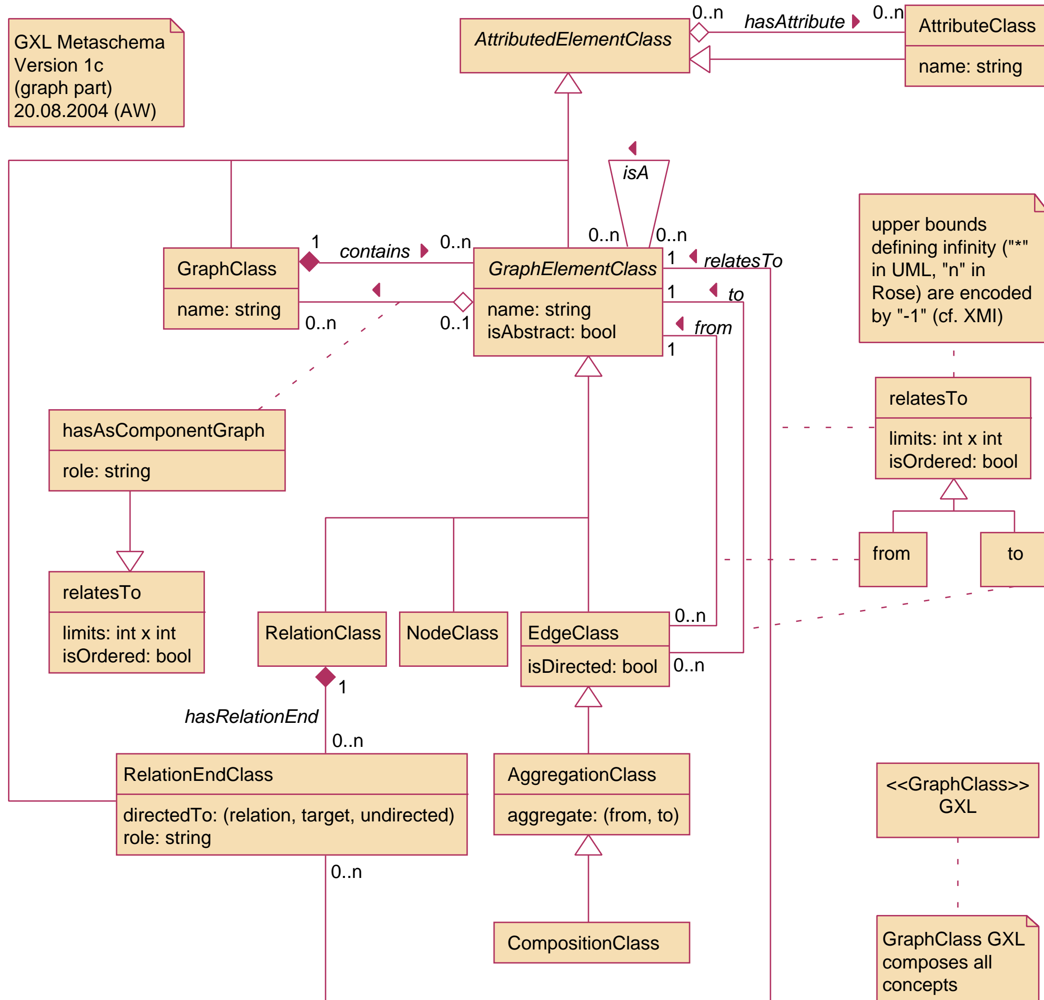

Fig. 16. *GXL* Metaschema

Attributes are added to GraphElementClasses by deriving them from AttributedElementClass. The definition of attribute structures supports the structured attributes used in *GXL* including the definition of default values. Generalization is provided for all GraphElementClasses by isA edges. GraphElementClasses containing subgraphs are associated with the representation of the lower level

GraphClass by contains edges. The GraphClass contains those node, edge, and hyperedge classes representing its structure. Aggregation (AggregationClass) and composition (CompositionClass) are modeled by specializations of EdgeClasses. Incidences of EdgeClasses and RelationClasses are modeled by from, to, relatesTo-edges. These incidences refer to all GraphElementClasses.

As with instance graphs, *GXL* schema graphs have to comply to some constraints that cannot be expressed with class diagrams [82]. In addition to the constraints discussed in Section 3.4, the following conditions are imposed:

- Schema graphs define graph classes. A schema graph contains at least one GraphClass node.
- Generalization hierarchies are acyclic. A schema graph does not contain a cycle of isA edges.
- Generalization is only permitted between classes of the same kind. In each schema graph isA edges only connect NodeClass nodes with other NodeClass nodes, EdgeClass nodes with others of its kind, and so on.

The *GXL* metaschema is itself a schema. Like all *GXL* schemas it is an instance of the *GXL* metaschema. It follows that the *GXL* metaschema is its own schema.

5 Using GXL

In the years since ratification of *GXL*, groups in reengineering, graph transformation, graph visualization, and other areas of software engineering have added support for *GXL* in their tools. Various tools have been created to support working with *GXL*. A framework for *GXL* converters [83] and a XMI2GXL translator [84] was developed at University BW München. In addition, a validator for checking *GXL* documents on instance, schema, and metaschema level has been developed [82]. A list of tools known to use *GXL* can be found on our web site [85].

There are many filters for converting *GXL* documents into local file formats and vice versa. These formats include Bauhaus Resource Graphs [86], DOT (GraphViz) [87], GraLab graphs [88], PROGRES graphs [89], RSF [38], and TA [33]. *GXL* is also supported by various fact extractors, such as Columbus/CAN [7], CPPX [6], TkSee/SN [90], and XOgastan [91]. Some reengineering workbenches that use *GXL* are Bauhaus [86], GUPRO [92], Rigi [93], SoftAnal [39], and SwagKit [94]. There are both general purpose graph drawing tools that support *GXL*, as well as visualizers for reengineering. These are GraphVis [28], Graph Visualization Framework [95], Shrimp [96], JGraph [97], touchgraph [98], and yFiles [99]. Furthermore *GXL* is supported by the GRAS [100] graph database, graph transformation systems (DiaGen [101], Fujaba [102], GenSet [103], and PROGRES/UPGRADE [89],[104]), and metacase tools (DiaGen [101], MetaEdit [105]).

These tools and converters have been used as the basis of data interchange on a number of occasions.

SoftAnal and GUPRO: SoftAnal [39] stores information about a stock trading system within relational databases. Using a *GXL* filter, this data was transferred to GUPRO [42] for further analysis, which in turn enabled comparison of the capabilities of both systems [43].

GReQL and grok: GReQL [106] and grok [107] offer powerful, query based analysis for graph based data. A survey, comparing the analysis capability of GReQL and grok, was done using a common *GXL* factbase [11].

Bauhaus and GUPRO: During the Dagstuhl seminar on "Software Architecture: Recovery and Modeling," there was an exercise in collaborative architecture reconstruction and modeling [108]. Groups had to work together to analyze the Apache web Server. During this exercise, *GXL* was used to transfer facts about Apache from Bauhaus [86] to GUPRO for further analysis.

Columbus/CAN and GUPRO: Columbus [109] is an extractor for C++ that emits ASTs in *GXL*. For refactoring purposes, this extractor was used within GUPRO for analysis of C++ sources.

In addition, there have been interesting applications of *GXL* in software engineering pedagogy, business process modeling, and biochemistry. At University of Toronto, *GXL* was used in an undergraduate software engineering course. Students were required to create graph editor/layouter components that communicated using *GXL* [110]. *GXL* was also applied to exchange business process models. *GXL* schemas for exchanging business processes depicted as Workflow Nets or Event-Driven Process Chains are given in [111]. The same authors also used *GXL*'s extension points to integrate with MathML [112] to exchange elaborated Workflow Nets containing expressions on the relational calculus [113]. Outside computer science, *GXL* has been used to represent regulatory networks of biological processes and biochemical behaviour [114].

6 Conclusion

In this paper, we gave an introduction to *GXL* 1.0 and its applications. We conclude with a summary of the key features of *GXL* and an assessment of its merits as a standard exchange format.

GXL is an XML language for representing graphs. The main features of the model in *GXL* are as follows.

- *Nodes, edges, and hyperedges are first class entities in GXL.* Consequently, each of these have unique identifiers, can be typed and attributed, and can be included in a generalization hierarchy.
- *Graphs, nodes, edges, hyperedges, and attributes have attributes.* This feature is used to add further information. For example, user annotations and coordinates for graph layout, are attached to the graph and passed as *GXL* attributes.
- *Graphs, nodes, edges, hyperedges and attributes are typed.* These element types are associated with a corresponding class in the schema. These rela-

tionships provide further information and constraints on the data.

- *Hierarchical graphs are supported.* This feature is implemented by permitting nodes, edges, and hyperedges to contain graphs. Edges and hyperedges are allowed to join nodes from different levels of the hierarchy.
- *Edges can be directed or undirected.* This flexibility supports in a general format for graphs. Both directed and undirected edges are permitted in the same graph.
- *Edges and hyperedges are ordered.* Incidence to and from the nodes at the endpoints of edges and hyperedges can be stored.

In GXL, both the data representing the graph and the data representing the schema are passed using the same graph model as an XML stream. The format and metaschema are sufficiently simple that it is possible to build schemas by hand. However, most users will likely create a schema by first modeling it as a UML class diagram and then using a tool to convert it to GXL (cf. [84]). This uniform application of syntax across the different levels of abstraction ensures that tools that implement GXL are capable of working with a variety of data.

The graph model of GXL ensures *universality*, because it includes the structural features needed to achieve compatibility with a wide variety of graph models. GXL is *typed* to facilitate interpretation and validation of exchanged data by making use of graph schemas. *Flexibility*, that is, the ability to adapt to domain specific data, is achieved through GXL-schemas, user-definable attributes, and extension points. They are used to specify domain-specific graph structures. GXL is *easy to use*. Furthermore, GXL is readable by humans, which facilitates learning, understanding, and debugging. Instance graphs and schemas are exchanged using the same document type, thus, only one language has to be learned. *Scalability* has been achieved, as GXL can be used with graphs of varying sizes and representations of software at different levels of abstraction. However, it does face the same issues as other XML formats regarding the increased size of data due to the addition of tags. Fortunately, standard compression techniques and other XML technologies can help solve the problem. *Modularity* is provided by supporting hierarchical graphs and by providing links to external documents. Incremental data exchange can be realized by GXL-based applications, as graphs can be exchanged in parts. Finally, GXL supports *extensibility* by offering predefined extension-points for enhancement.

Developing and deploying GXL has been an exciting and challenging experience. Through many intense discussions, we were able to build bridges between research groups and even between research areas and cultures. Arriving at a standard required us to understand the differences in data formats, research approaches, and problem domains. The result has been fruitful collaborations between researchers and improved data interoperability between tools.

GXL is currently being applied and evaluated by the research community. There is work still to be done in developing standard schemas and broadening the acceptance of GXL. Current projects include the implementation of tools to filter and validate GXL, and for drawing graph schemas. In addition, GXL

reference schemas have been proposed. Some proposed reference schemas for reengineering include abstract syntax trees for specific source languages [115], an external declaration or "middle model" [116], a high-level architectural schema, and one for data reverse engineering. These schemas span different levels of abstraction for reengineering tools and they involve a wide range of participants from the community. We look forward to maturing *GXL* along with the research discipline and tools for reengineering.

Acknowledgments. We thank our collaborators for many fruitful discussions on the development of *GXL*. In particular, we thank Jürgen Ebert, Bernt Kullbach, and Volker Riediger for insights into TGraphs and *GXL*. We also like to thank our students, who did much important work on presenting *GXL* in the Web and implementing *GXL* tools and filters. We owe a great debt to Kostas Kontogiannis and Rainer Koschke who were nurturing an interest in a standard exchange format for reverse engineering, long before we started our work. Thanks also to Tim Lethbridge, Hausi Müller, and other members of CSER. Thanks to Ulrik Brandes, Scott Marshall, Mark Minas, and Gabriele Taentzer who helped us build bridges to other research communities. These relationships helped to improve GXL and to increase its use. Thanks to all users of *GXL*, who currently applying and testing *GXL* 1.0 in their tools. Their experience and change requests are important contributions for improving *GXL*.

References

[1] A. I. Wasserman, Tool Integration in Software Engineering Environments, in: International Workshop on Software Engineering Environments (SEE), Chinon, France, 1989, pp. 137–149.

[2] A. Earl, A Reference Model for Computer Assisted Software Engineering Environments Frameworks, Softwaretechnik-Trends 11 (2), 1991, 15–48.

[3] Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommendation, W3C XML Working Group, `http://www.w3.org/TR/2004/REC-xml-20040204`, February 2004.

[4] S. R. Tilley, Domain-Retargetable Reverse Engineering, Phd thesis, Department of Computer Science, University of Victoria, January 1995.

[5] Y.-F. Chen, M. Y. Nishimoto, C. V. Ramamoorthy, The C Information Abstraction System, IEEE Transactions on Software Engineering 16 (3), 1990, 325–334.

[6] T. Dean, A. Malton, R. Holt, Union Schemas as a Basis for a C++ Extractor, in: 8th Working Conference on Reverse Engineering, IEEE Computer Society, Los Alamitos, 2001, pp. 59–67.

[7] R. Ferenc, A. Beszédes, Data Exchange with the Columbus Schema for C++, in: 6th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Los Alamitos, 2002, pp. 59–66.

[8] Information technology – Programming languages - Ada Semantic Interface Specification (ASIS), http://www.acm.org/sigs/sigada/wg/asiswg/, 1999.

[9] A. Schürr, A. J. Winter, A. Zündorf, PROGRES: Language and Environment, in: H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Handbook on Graph Grammars: Applications, Languages, and Tools, Vol. 2, World Scientific, Singapore, 1999, pp. 487–550.

[10] J. Ebert, B. Kullbach, A. Panse, The Extract-Transform-Rewrite Cycle - A Step towards MetaCARE, in: P. Nesi, F. Lehner (Eds.), 2nd Euromicro Conference on Software Maintenance & Reengineering, IEEE Computer Society, Los Alamitos, 1998, pp. 165–170.

[11] J. Wu, R. C. Holt, A. Winter, Towards a Common Query Language for Reverse Engineering, Fachberichte Informatik, http://www.uni-koblenz.de/fb4/publikationen/gelbereihe/RR-8-2002.pdf 8/2002, Universität Koblenz-Landau, Institut für Informatik, Koblenz, 2002.

[12] R. Ommering, L. van Feijs, R. Krikhaar, A relational approach to support software architecture analysis, Software Practice and Experience 28 (4), 1998, 371–400.

[13] P. T. Devanbu, GENOA – A Customizable, Language and Front-End independent Code Analyzer, 14th International Conference on Software Engineering, Melbourne, 1992, 307–317.

[14] R. F. Crew, ASTLOG: A Language for Examining Abstract Syntax Trees, in: Conference on Domain-specific Languages, October 15-17, 1997, Santa Barbara, USENIX Association, Berkley, 1997.

[15] J.-F. Girard, R. Koschke, Finding Components in a Hierarchy of Modules - a Step towards Architectural Understanding, in: International Conference on Software Maintenance, IEEE Computer Society Press, 1997.

[16] H. M. Fahmy, R. C. Holt, Software architecture transformations, in: International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, 2000, pp. 88–96.

[17] M. Weiser, Program Slicing, IEEE Transactions on Software Engineering SE-10 (4), 1984, 352–357.

[18] K. Chen, V. Rajlich, RIPPLES: Tool for Change in Legacy Software, in: 5th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Los Alamitos, 2001, pp. 230–239.

[19] T. Mens, M. Lanza, A Graph-based Metamodel for Object-oriented Software Metrics, ENTCS 72 (2), 2002, (http://www.elsevier.nl/locate/entcs/volume72.html).

[20] T. Lethbridge, N. Anquetil, Architecture of a Source Code Exploration Tool: A Software Engineering Case Study, Computer Science Technical report http://www.site.uottawa.ca/ tcl/papers/Cascon/TR-97-07.pdf, University of Ottawa, 1997.

[21] Y.-F. Chen, F. G. S, E. Koutsofios, R. S. Wallach, Ciao: A Graphical Navigator for Software and Document Repositories, in: International Conference on Software Maintenance, IEEE Computer Society Press, 1995, pp. 66–75.

[22] H. Eichelberger, J. von Gudenberg, On the Visualization of Java Programs, in: S. Diehl (Ed.), Software Visualization, International Seminar, Dagstuhl Castle, Germany, May 20-25, 2001, LNCS 2269, Springer, Berlin, 2002, pp. 295–306.

[23] K. Oechsle, T. Schmitt, JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams using the Java Debug Interface, in: S. Diehl (Ed.), Software Visualization, International Seminar, Dagstuhl Castle, Germany, May 20-25, 2001, LNCS 2269, Springer, Berlin, 2002, pp. 176–175.

[24] R. Castello, R. Milli, I. G. Tollis, A Framework for the Static and Interactive Visualization for Statecharts, Journal of Graph Algorithms and Applications 6 (3), 2002, 313–351.

[25] K. Wong, RIGI User's Manual, Version 5.4.3, `http://www.rigi.csc.uvic.ca/rigi/rigiframe1.shtml?Download`, 1996.

[26] M. Fröhlich, M. Werner, daVinci V2.0.x Online Documentation, `http://www.tzi.de/~davinci/docs/`, June 1996.

[27] M. Himsolt, GML: Graph Modeling Language, `http://www.infosun.fmi.uni-passau.de/Graphlet/`, December 1996.

[28] Graphviz - open source graph drawing software, `http://www.research.att.com/sw/tools/graphviz/`, 2002.

[29] M.-A. Storey, C. Best, J. Michand, SHriMP views: an interactive environment for exploring Java programs, in: 9th International Workshop on Program Comprehension, IEEE, Los Alamitos, 2001, pp. 111–112.

[30] R. C. Holt, A. Winter, A. Schürr, GXL: Toward a Standard Exchange Format, in: 7th Working Conference on Reverse Engineering, IEEE Computer Society, Los Alamitos, 2000, pp. 162–171.

[31] Y.-F. Chen, E. R. Gansner, E. Koutsofios, A C++ Data Model Supporting Reachability Analysis and Dead Code Detection, IEEE Transactions on Software Engineering 24 (9), 1998, 682–694.

[32] M. W. Godfrey, E. H. S. Lee, Secrets from the Monster: Extracting Mozilla's Software Architecture, in: 2nd International Symposium on Constructing Software Engineering Tools, Limerick, Ireland, 2000.

[33] R. C. Holt, An Introduction to TA: The Tuple-Attribute Language, `http://plg.uwaterloo.ca/~holt/papers/ta.html`, 1997.

[34] R. C. Holt, PBS: Portable Bookshelf Tools, `http://www.turing.toronto.edu/pbs` 1997.

[35] R. Kazman, J. Carrière, View Extraction and View Fusion in Architectural Understanding, in: International Conference on Software Reuse, IEEE Computer Society Press, Los Alamitos, 1998, pp. 290–299.

[36] R. Kazman, J. Carrière, Playing Detective: Reconstructing Software Architecture from Available Evidence, Automated Software Engineering 6 (2), 1999, 107–138.

[37] W. R. Bischofberger, Sniff: A Pragmatic Approach to a C++ Programming Environment, in: USENIX C++ conference, Portland, Oregon, August 1992, pp. 67–82.

[38] K. Wong, RIGI User's Manual, Version 5.4.4, `http://www.rigi.csc.uvic.ca/rigi/rigiframe1.shtml?Download`, 1998.

[39] H. M. Sneed, T. Dombovari, Comprehending a Complex, Distributed, Object-oriented Software System, A Report from the Field, in: 7th international Workshop on Program Comprehension, IEEE, Los Alamitos, 1999, pp. 218–225.

[40] J. Ebert, B. Kullbach, A. Winter, GraX – An Interchange Format for Reengineering Tools, in: [61], 1999, pp. 89–98.

[41] J. Ebert, A. Winter, P. Dahm, A. Franzke, R. Süttenbach, Graph Based Modeling and Implementation with EER/GRAL , in: B. Thalheim (Ed.), Conceptual Modeling — ER'96, LNCS 1157, Springer, Berlin, 1996, pp. 163–178.

[42] J. Ebert, B. Kullbach, V. Riediger, A. Winter, GUPRO – Generic Understanding of Programs, An Overview, ENTCS 72 (2), (`http://www.elsevier.nl/locate/entcs/volume72.html`).

[43] C. Lange, H. Sneed, A. Winter, Comparing Graph-based Program Comprehension Tools to Relational Database-based Tools, in: 9th International Workshop on Program Comprehension, IEEE, Los Alamitos, 2001, pp. 209–218.

[44] H. Müller, Criteria for Success, in Exchange Formats for Information Extracted from Computer Programs, `http://plg2.math.uwaterloo.ca/~holt/sw.eng/exch.format/`, 1998.

[45] R. Koschke, J.-F. Girard, M. Würthner, An Intermediate Representation for Integrating Reverse Engineering Analyses, in: [59], 1998, pp. 241–250.

[46] I. Bowman, M. Godfrey, R. Holt, Conecting Architecture Reconstruction Frameworks, in: First International Symposium on Constructing Software Engineering Tools (CoSET1999), 1999.

[47] M. van den Brand, H. A. de Jong, P. Klint, P. A. Olivier, Efficient annotated Terms, Software: Practice and Experience 30 (3), 2000, 259–291.

[48] M. Minas, Visual specification of visual editors with DiaGen, in: International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'03), Charlottesville, 2003.

[49] The GML File Format, `http://www.infosun.fmi.uni-passau.de/Graphlet/GML/index.html`.

[50] F. Newbery Paulish, The Design of an Extendible Graph Editor, LNCS 704, Springer, Berlin, 1991.

[51] Extensible Graph Markup and Modeling Language), http://www.cs.rpi.edu/~puninj/XGMML/, 2001.

[52] I. Herman, S. Marshall, GraphXML – An XML-Based Graph Description Format, in: J. Marks (Ed.), Graph Drawing, 8th International Symposium, GD 2000 Colonial Williamsburg, LNCS 1984, Springer, Berlin, 2000, pp. 52–61.

[53] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, M. S. Marschall, GraphML Progress Report, Structural Layer Proposal, in: P. Mutzel, M. Jünger, S. Leipert (Eds.), Graph Drawing, 9th International Symposium, GD 2001 Vienna, LNCS 2265, Springer, Berlin, 2001, pp. 501–512.

[54] U. Brandes, C. Pich, J. Lerner, GXL to GraphML and Vice Versa with XSLT, in: 2nd Intl. Workshop Graph-Based Tools (GraBaTs '04), ENTCS, to appear.

[55] Meta Object Facility (MOF) Specification, http://www.omg.org/technology/documents/formal/mof.htm, March 2000.

[56] XML Meta Data Interchange (XMI) Specification, http://www.omg.org/technology/documents/formal/xmi.htm, November 2000.

[57] E. Mamas, K. Kontogiannis, Towards Portable source Code Representation Using XML, in: 7th Working Conference on Reverse Engineering, IEEE Computer Society, Los Alamitos, 2000, pp. 172–182.

[58] Y. Zou, K. Kontogiannis, Towards A Portable XML-based Source Code Representation, Workshop of XML Technologies and Software Engineering (XSE), ICSE 2001, Toronto.

[59] 5th Working Conference on Reverse Engineering, IEEE Computer Society, Los Alamitos, 1998.

[60] Data Exchange Group, Conclusions from Meeting at CASCON 1998, Monday,30 Nov 1998, http://plg.uwaterloo.ca/~holt/sw.eng/exch.format/minutes98_11_30.html.

[61] 6th Working Conference on Reverse Engineering, IEEE Computer Society, Los Alamitos, 1999.

[62] 7-ter Workshop des GI-Arbeitskreises GROOM, UML - Erweiterungen (Profile) und Konzepte der Metamodellierung, 4.-5. April 2000, Universität Koblenz-Landau, http://www2.informatik.unibw-muenchen.de/GROOM/META/index.htm.

[63] S. E. Sim, R. C. Holt, R. Koschke, ICSE 2000 Workshop on Standard Exchange Format (WoSEF), Limerick, 2000.

[64] First EU Working Group on "Application of Graph Transformation" meeting on GXL (graph exchange language) and GTXL (graph transformation exchange language) in Paderborn (September 5-6, 2000), http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl/paderborn.html.

[65] Satellite Workshop on Data Exchange Formats 8th Int. Symposium on Graph Drawing (GD 2000), http://www.cs.virginia.edu/~gd2000/gd-satellite.html, 2001.

[66] R. C. Holt, A. Winter, Software Data Interchange with GXL: Introduction and Tutorial, CASCON 2000, Mississauga, Ontario, http://www.cas.ibm.com/archives/2000/workshops/descriptions.shtml#16, 2000.

[67] S. E. Sim, Software Data Interchange with GXL: Implementation Issues, CASCON 2000, Mississauga, Ontario, http://www.cas.ibm.com/archives/2000/workshops/descriptions.shtml#17 November, 2000.

[68] K. Kontogiannis, Exchange Formats Workshop, in: 7th Working Conference on Reverse Engineering, IEEE Computer Society, Los Alamitos, 2000, pp. 277–301.

[69] J. Ebert, K. Kontogiannis, J. Mylopoulos: Interoperability of Reverse Engineering Tools, http://www.dagstuhl.de/DATA/Reports/01041/, 2001.

[70] Graph Transformation System Exchange Language, http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html, 2001.

[71] G. Taentzer, Towards Common Exchange Formats for Graphs and Graph Transformation Systems, in: UNIGRA satellite workshop of ETAPS'01, 2001.

[72] T. Mens, A. Schürr, G. Taentzer (Eds.), Graph-Based Tools, ENTCS 72/2, http://www.elsevier.com/locate/entcs/volume72.html, 2002.

[73] Graph Drawing (GD 2001), Vienna, http://www.ads.tuwien.ac.at/gd2001/, 2001.

[74] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, Addison Wesley, Reading, 1999.

[75] J. B. Warmer, A. G. Kleppe, The Object Constraint Language : Precise Modeling With UML, Addison-Wesley, 1998.

[76] C. Berge, Graphs and Hypergraphs, 2nd Edition, North-Holland, Amsterdam, 1976.

[77] G. Busatto, An Abstract Model of Hierarchical Graphs and Hierarchical Graph Transformation, http://www.informatik.uni-bremen.de/~giorgio/papers/phd-thesis.ps.gz, 2001.

[78] XML Schema Part 0: Primer, W3C Recommendation, 2 May 2001 , http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/, 2001.

[79] XMI Toolkit 1.15 (Updated on: 25.04.2000), http://alphaworks.ibm.com/tech/xmitoolkit, 2000.

[80] I. Herman, M. S. Marshall, Graph XML – An XML based graph interchange format, Report INS-0009, Centrum voor Wiskunde en Informatica, Amsterdam, April 2000.

[81] XML Schema Part 2: Datatypes, http://www.w3.org/TR/xmlschema-2/, 02 May 2001.

[82] A. Kaczmarek, GXL Validator, Validierung von GXL-Dokumenten auf Instanz-, Schema, und Metaschema-Ebene, Studienarbeit, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, 2003.

[83] GCF - a GXL Converter Framework, http://www2.informatik.unibw-muenchen.de/GXL/triebsees/index.htm.

[84] XIG - An XSLT-based XMI2GXL-Translator, http://ist.unibw-muenchen.de/GXL/volk/index.htm.

[85] GXL: Graph Exchange Language, http://www.gupro.de/GXL/tools/tools.html.

[86] Bauhaus:
Software Architecture, Software Reengineering, and Program Understanding, http://www.informatik.uni-stuttgart.de/ifi/ps/bauhaus/.

[87] Graphviz - open source graph drawing software, GXL2DOT, DOT2GXL, http://custom.lab.unb.br/pub/graph/graphviz/tools/src/, 2002.

[88] P. Dahm, F. Widmann, Das Graphenlabor, Version 4.2, Fachbericht Informatik 11/98, Universität Koblenz-Landau, Institut für Informatik, Koblenz, 1998.

[89] A Graph Grammar Programming Environment - PROGRES, http://www-i3.informatik.rwth-aachen.de/research/projects/progres/main.html.

[90] TkSee, http://www.site.uottawa.ca/~tcl/kbre/options/.

[91] XOgastan: Xml-Oriented Gnu AST Analyzer, University of Sannio, Benevento, http://web.ing.unisannio.it/villano/students/masone/, 2002.

[92] GUPRO: Generic Understanding of Programs, http://www.gupro.de/.

[93] RIGI: a visual tool for understanding legacy systems, http://www.rigi.csc.uvic.ca/.

[94] SWAG Software Toolkit, http://www.swag.uwaterloo.ca/~swagkit/.

[95] GVF - The Graph Visualization Framework , http://sourceforge.net/projects/gvf/, 2003.

[96] ShriMP Views: simple Hierarchical Multi-Perspective, http://www.shrimpviews.com/.

[97] JGraph: The Home Page of JGraph, http://jgraph.sourceforge.net/index.html, 2003.

[98] TouchGraph, http://www.touchgraph.com/index.html.

[99] yFiles - Interactive Visualization of Graph Strucutres, http://www-pr.informatik.uni-tuebingen.de/yfiles/.

[100] GRAS - A graph oriented database system for (software) engineering environments, http://www-i3.informatik.rwth-aachen.de/research/projects/gras/index.html, 1999.

[101] DiaGen:The Diagram Editor Generator, Universität Erlangen-Nürnberg, http://www2.informatik.uni-erlangen.de/DiaGen/, 2002.

[102] Fujaba: From UML to Java and back again, http://www.uni-paderborn.de/cs/fujaba/.

[103] GenSet: Design Information Fusion, http://www.cs.uoregon.edu/research/perpetual/dasada/Software/GenSet.

[104] UPGADE: A framework for graph-based applications, RWTH Aachen, http://www-i3.informatik.rwth-aachen.de/research/projects/upgrade/.

[105] MetaEdit+ metaCASE tool, http://www.metacase.com/.

[106] M. Kamp, B. Kullbach, GReQL - Eine Anfragesprache für das GUPRO–Repository, Sprachbeschreibung, Projektbericht 8/2001, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 2001.

[107] R. C. Holt, Introduction to the Grok Programming Language, http://plg.uwaterloo.ca/~holt/papers/grok-intro.doc, 2002.

[108] Collaborative architecture reconstruction and modeling task, Workshop at the Dagstuhl-Seminar 03061 "Software Architecture: Recovery and Modelling´´, http://www.bauhaus-stuttgart.de/dagstuhl/#tools.

[109] R. Ferenc, F. Magyar, A. Beszédes, A. Kiss, M. Tarkiainen, Columbus - Tool for Reverse Engineering Large Object Oriented Software Systems, in: SPLST 2001, Szeged, Hungary (http://www.inf.u-szeged.hu/~ferenc/research/ferencr_columbus.pdf, 2001, pp. 16–27.

[110] S. Easterbrook, CSC444F: Software Engineering I (Fall term 2001), University of Toronto, http://www.cs.toronto.edu/~sme/CSC444F/, 2001.

[111] A. Winter, C. Simon, Exchanging Business Process Models with GXL, in: M. Nüttgens and J. Mendling: XML4BPM 2004, 1st GI Workshop XML4BPM – XML Interchange Formats for Business Process Management at 7th GI Conference Modellierung 2004, Marburg Germany, March 2004, http://wi.wu-wien.ac.at/~mendling/XML4BPM/xml4bpm-2004-proceedings-gxl.pdf, 2004, pp. 103–122.

[112] Mathematical Markup Language (MathML) Version 2.0 (2nd Edition), http://www.w3.org/TR/2003/REC-MathML2-20031021/.

[113] C. Simon, A. Winter, Exchanging Process Specifications for Identifying Cooperative Information Systems, in: 11th Workshop on Algorithms and Tools for Petri Nets September 30 - October 1, 2004, Paderborn Germany, 2004, 31–36.

[114] C. Chaouiya, A. G. Gonzalez, D. Thieffry, GINML: Towards a GXL based format for logical regulatory networks and dynamic graphs, http://www.esil.univ-mrs.fr/~chaouiya/Recherche/GINML, 2003.

[115] R. Ferenc, S. E. Sim, R. C. Holt, R. Koschke, T. Gyimòthy, Towards a Standard Schema for C/C++, in: 8th Working Conference on Reverse Engineering, IEEE Computer Society, Los Alamitos, 2001, pp. 49–58.

[116] T. C. Lethbridge, S. Tichelaar, E. Ploedereder, The Dagstuhl Middle Metamodel, A Schema for Reverse Engineering, in: J.-M. Favre, M. Godfrey, A. Winter, International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM 2003), Electronic Notes in Theoretical Computer Science, Vol. 94, http://www.sciencedirect.com/science/journal/15710661, 2004, pp. 7–18.