GXL: Toward a Standard Exchange Format

Richard C. Holt
University of Waterloo
Department of Computer Science
Waterloo N2L 3G1, Canada
holt@plg.uwaterloo.ca

Andreas Winter
University of Koblenz-Landau
Institute for Software Technology
D-56016 Koblenz, Germany
winter@uni-koblenz.de

Andy Schürr
University Bw, München
Institute for Software Technology
D-85577 Neubiberg, Germany
Andy.Schuerr@unibw-muenchen.de

Abstract

This paper describes ongoing work toward the development of a standard software exchange format (SEF), for exchanging information among tools that analyze computer programs. A particular exchange format called GXL (Graph Exchange Language) is proposed. GXL can be viewed as a merger of well known formats (e. g. GraX, PROGRES, RPA, RSF, and TA) for exchanging typed, attributed, directed graphs. By using XML as notation, GXL offers a scaleable and adaptable means to facilitate interoperability of reengineering tools.

Keywords: software exchange format, tool interoperability, XML, typed attributed directed graphs,

1 Introduction

The fields of reverse engineering and program analysis have matured to the extent that there are many tools to extract information about programs, to manipulate this information and to analyze it. What is missing is a generally accepted means for exchanging information among these tools, to allow these tools to interoperate. This paper introduces GXL, a graph-based format in XML, as a means to support this interoperation. GXL has evolved from many discussions among groups developing software tools, with the actual GXL design coming out of a cooperative effort from the members of the GUPRO (University of Koblenz) [8], PBS (University of Waterloo) [15] and PROGRES (University Bw, Munich) [39] teams. GXL also was influenced by the Rigi Standard Format (RSF, University of Victoria) [48] and Relation Partition Algebra (RPA, Philips Research, Eindhoven) [26]. So GXL can be viewed as a common graph exchange format encompassing TA, TGraphs, RPA, RSF, and PROGRES. It appears that other interchange formats used in graph based tools (e. g. graph browsers, graph layouters, graph transformation systems) like daVinci [30], GML, Graphlet [17], GRL [36], and in software reengineering tools like ATerms [44] can easily be mapped on GXL.

Being a structural description, graphs have no meaning of their own. The meaning of graphs corresponds to the context in which they are exchanged. This context is, in part, given by a schema (essentially an E/R diagram or class diagram) [10]. In GXL, both the actual data representing the graph and the schema are passed as XML stream. Other information associated with the graph, such as user annotations or *(x, y)* locations for graph layout, is attached to the graph and passed in GXL as attributes. Although this paper concentrates on use of an SEF for software analysis, GXL is a general notation for representing typed graphs, and could be used for many other purposes.

The rest of the paper is organized as follows. We present examples of information to be exchanged, examples of interconnection of software tools, and the levels of information to be exchanged (from abstract syntax tree to architectural facts). We list a range of software tools that have been interconnected using the SEFs that were combined into GXL, to show how GXL can scale up to handle large industrial graphs. We explain that the syntax of the XML stream for GXL is specified by a document type definition (DTD), and that the form of graphs to be described is specified by an UML class diagram, encoded as GXL document. Finally, we discuss criteria for the success of an SEF.

2 Using Typed Graphs to Exchange Information

The information to be exchanged about programs is often presented as a diagram, as illustrated in figure 1. This diagram represents an attributed typed directed graph with two types of edges (*Call* and *Ref*) and two types of nodes (*Proc* for *P* and *Q* and *Var* for *V* and *W*) along with node and edge attributes named *File* and *Line*.

Figure 1 can be interpreted as follows. On line *42*, procedure *P*, which is stored in file *main.c*, calls procedure *Q*, which is stored in file *test.c*. *P* references variable *V* which is declared on line *225*. *Q* references variable *W* which is declared on line *316*. Edge *(P, Q)* has type *Call* and a *Line* attribute whose value is *127*. Generally, a node or edge can have any number of attributes. There can be more than one edge between nodes, as will be explained below.

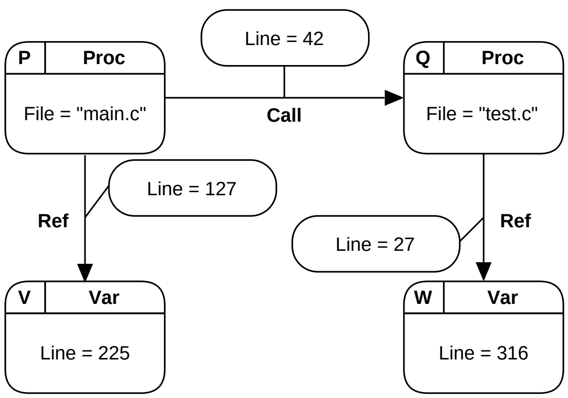

Figure 1. Sample typed graph with attributes

Attributed typed directed graphs (or simply typed graphs, for short) are convenient for representing a rich class of information to be exchanged about computer programs. Their clear mathematical foundation gives them generality and flexibility, independent of any particular application. This is analogous to the advantage of the mathematical foundation of relational data bases, which is distinct from the schema or data of a particular data base. In fact, any typed graph can be directly transcribed into a relational data base and vice versa. Typed graphs, or to be more precise, directed graphs with typed nodes and edges, with attributes on both nodes and edges, provide the mathematical foundation of TA and TGraphs. Typed graphs with corresponding operators have been axiomatized by [43], and formalised as Relation Partition Algebra (RPA) [26]. A more general class of typed graphs is given by TGraphs [7]. TGraphs (attributed, typed, directed and ordered graphs) have been formalized using the Z notation [42]. In the next section we will now show how typed graphs can be represented in XML.

3 GXL: A Standard Exchange Format in XML

Graphs such as the one in figure 1 can be represented as an XML document [46]. XML offers a adaptable standardized markup language for describing documents according to a given document structure (DTD). For example, the second line of the XML stream in figure 2, node *P* is specified to have type *Proc* and to have an attribute named *File* whose value is *main.c*. In a similar way, this XML stream specifies the other nodes and edges in figure 1.

```
<gxl>
<node id="P" type="Proc">
    <attr name="File" value="main.c" />
</node>
<node id="Q" type="Proc">
    <attr name="File" value="test.c" />
</node>
<node id="V" type="Var">
    <attr name="Line" value="225" />
</node>
<node id="W" type="Var">
    <attr name="Line" value="316" />
</node>
<edge begin="P" end="Q" type="Call">
    <attr name="Line" value="42" />
</edge>
<edge begin="P" end="V" type="Ref">
    <attr name="Line" value="127" />
</edge>
<edge begin="Q" end="W" type="Ref">
    <attr name="Line" value="316" />
</edge>
</gxl>
```

Figure 2. Graph in figure 1 represented in XML (GXL document). The nodes (*P*, *Q*, *V*, and *W*) and edges *(P, Q)*, *(P, V)*, and *(Q, W)* are represented along with their types and attributes.

Figure 2 is an example of use of GXL. GXL is a slightly modified version of GraX [10], expanded for convenient representation of the information in TA streams as well as for representation of PROGRES graphs. More sophisticated features of GXL like schema references, composite attribute structures, edge identifiers, and edge ordering are not explained in this small example. For more details see the GXL-DTD extract in section 7.

4 Scaling Up to Handle Large Software

To be industrially useful, GXL must scale up to handle software systems comprised of hundreds of thousands or millions of lines of code. TA, TGraphs and PROGRES, which are the main notations that were merged to create GXL, have been used extensively in analyzing such software systems. We list examples of these analyses to indicate that GXL can handle such cases:

TA has been applied to the following systems:

1. IBM's TOBEY code optimizer [16]. This is a 250,000 source line program written in the PLIX dialect of PL/I.
2. IBMs AS400 Cobol front end. This is a 300,000 source line C program.
3. IBM's IOC (IBM Open Class library for C++). This is a 1,000 class library to support general development in C++.
4. MITEL's SX2000 telephone switch. This is a Pascal program with 3,000,000 lines of code.
5. Linux kernel. This is the 700,000 source line open source Unix clone [2]. The Linux kernel has been analyzed using CFX, using SNIFF+ and using TA for interconnection.
6. GCC C++ compiler. This is the 1,000,000 line open source C++ optimizing compiler.
7. Mozilla (Netscape) web browner. This commercial system was made open source.
8. Apache web server. This is the 80,000 source line open source web server, which is the most widely used web server.
9. VIM (VI Improved) text editor. This is the 150,000 source line open source implementation of the VI (VIsual editor).

In the analysis of each of these systems, TA was the basis for exchanging information between the parser, manipulation and analysis tools.

GUPRO [8] is a generic software reengineering workbench. Software systems are represented by TGraphs matching a conceptual model (graph class). GUPRO is generic in that sense, that these conceptual models are adapted to the concrete needs of certain reengineering problems. This allows to use common components to analyze or to manipulate TGraphs scaled to the given problem. GUPRO tools were applied to the following software systems:

1. KAP. This is a cash box system maintained by IBM Deutschland Informationssysteme GmbH. IBM bought the sources from a third party and was interested in estimating the maintainability of this C code.
2. Volkfürsorge Application Landscape [23, 27]. Altogether this MVS system consists of 25000 JCL, 5700 CSP, 4000 COBOL, 3800 Delta COBOL, 6000 PL/I, and 1000 assembler programs and various databases with more than 3000 entity types. The software system at Volksfürsorge, a large German insurance company, was analyzed at an architectural level.
3. Software Security Systems (together with Bundesamt für Sicherheit in der Informationstechnik). These are various C/C++ systems which have to be certified to meet various software security requirements, i. e., for encoding and decoding classified data.
4. SQLTABS. This describes the application landscape of an Austrian bank. This C/C++ system consists of about 3000 modules and 16000 methods or procedures.

Analogous to the TA approach, the analysis of these systems was based on TGraphs. Suitable parsers translate the sources into TGraphs, which were analyzed by using the GReQL graph query language [22] and graph algorithms implemented with the GraLab graph library [6]. Furthermore TGraphs are widely used in the field of visual languages. KOGGE [11] is a MetaCASE tool for generation graphical editors for these languages. An overview of applications based on TGraphs including graph algorithmic examples is given in [12].

PROGRES [39] is a graph grammar based approach to graph transformation. In the domain of software engineering, it has been used to build various forward and reengineering tools, for example:

1. Gebühren Einzugszentrale (GEZ) and Aachener und Münchener insurance company. These systems are large COBOL applications. Both systems were transferred into object based systems [4] and into distributed JAVA applications [38].
2. IPSEN CASE tool [32]. This is a research prototype of an integrated software development environment with round-trip engineering capabilities (incremental propagation of changes between various types of analysis, design, and programming documents) which was realized at the RWTH Aachen.
3. Database reengineering tool VARLET [21]. This tool was built at the University of Paderborn and transforms relational database schemata into object-oriented database schemata using graph transformation technology.
4. Software performance modeling and analysis tools [37] at Carleton University Ottawa. Graph transformation based tools derive software performance models and typical message exchange patterns from software architectures and runtime traces.

These examples show the practical relevance and industrial strength of the main graph approaches combined into GXL. This provides evidence that GXL can cope with large industrial software systems.

5 Interconnecting Software Extractor, Manipulator and Analysis Tools

We can categorize the software tools of interest into three rough classes:

1. extractors, which collect information about the software,
2. manipulators, which change the form of this data, and

3. analyzers which determine and display information derived from the data.

As an example of an extractor, the ACACIA C++ front-end from AT&T Labs [3] parses C++ programs and stores the resulting information in a relational data base. GUPRO uses a YACC/LEX like parser generator PDL [5] for generating parsers delivering TGraphs. Furthermore, there exist parser front ends providing abstract syntax trees e. g. EDG [13] for C/C++ or ASIS (ADA Semantic Interface Specification) [20]. An overview to various parsing techniques used in reengineering is given in [45]. As an example of (2), the PROGRES system [39] transforms graph representations e. g. in order to translate COBOL programmes into OO-Cobol programmes [4] or to translate monolithic OO programmes into distributed client/server applications on top of CORBA [38]. In [9] a TGraph based approach on semantics preserving code manipulation is presented. As an example of (3), an analyzer tool may compute the McCabe complexity of procedures represented in the data [41]. RPA [35] [26] uses a relational approach to analyzing software architectures. A general graph query approach to software analysis is followed in GUPRO [8]. As another example, the Rigi system [48] displays graph information, as colored, nested, box and arrow diagrams.

It is a goal of GXL to allow such tools to be conveniently interconnected. To do this, extractor tools would emit their data as GXL stream. Manipulator tools would input a GXL stream, manipulate the corresponding graph, and output a new GXL stream. Analyzers would input a GXL stream and create corresponding reports and visualizations. Of course, these tools can use their own internal representations, including internal data bases, in-memory data structures, and file formats. GXL would be used for external communication.

We will now discuss how the notations combined into GXL have been used to interconnect many software analysis tools, to indicate how GXL should be useful in a similar way. Here is a list of tools that have been interconnected in various ways using PROGRES, TA, or TGraphs:

1. Acacia C++ parser tool kit for C++ and graph drawing, by AT&T labs. The analysis is at the external declaration level. There is a command line interface that allows extraction of facts from Acacia's data base of facts about an parsed C++ program. This interface has been used by Eric Lee to create a corresponding TA stream, used in turn for analyzing Mozilla source code using PBS tools.
2. ASIS, ADA (Semantic Interface Specification) [20] provides a standardized interface to ADA ASTs. Ongoing work at University of Koblenz transfers ADA ASTs into TGraphs to offer a common base for analysis of ADA systems.
3. Bauhaus reengineering tool set, by University of Stuttgart. The Bauhaus front end parses C and stores the result in a data structure with an API. Mike Godfrey has extracted facts from Bauhaus analysis and transcribed them to TA for viewing by LSEDIT.
4. CPPanal by Harry Sneed. These tools extract source code information on an architectural level from large software systems and stores them in SQL tables. By using GraX these tables are transfered into TGraphs for further analysis with GUPRO tools.
5. Cxref, by Andrew Bishop. Cxref inputs C and produces documentation (in LaTeX, HTML, RTF or SGML) including a cross-references. Mike Godfrey has extracted facts from Cxref and transcribed them to TA.
6. CFX (C Fact eXtractor) C parser from the PBS tool kit, by the University of Toronto. This is a C parser that outputs facts about the C program in RSF, which is then transcribed to TA.
7. Dali reverse engineering tool kit, by Software Engineering Institute. This tool kit combines various other tools, including the Rigi viewer, and has used TA for analysis of Linux.
8. Datrix C++ parser from the Datrix tool kit, by Bell Canada. This front end creates a complete AST (abstract syntax tree) in extended TA from C++ source programs. The parser serves as the basis for Bell Canada's extensive set of software quality analysis tools.
9. DaVinci [30] from University Bremen is a graph visualization system. There are filters which transform TGraphs into the internal representation of DaVinci. Furthermore in GenVis [18] DaVinci was combined with GReQL. By using GReQL large graphs can be scaled down to smaller subgraphs that can be visualized with DaVinci.
10. EDGE from the University of Karlsruhe is a graph browser. EDGE has been used to display and layout PROGRES graphs.
11. G2HTML from University of Koblenz is a filter that translates TGraphs into HTML documents. These HTML documents are used in GUPRO to browse through large graphs.
12. GCC by GNU. GCC is an open source C/C++ compiler which provides accessing internally constructed abstract syntax trees. Ongoing work at University of Koblenz transfers these ASTs into TGraphs to offer a common base for further code analysis.
13. Graphlet from University of Passau [17]. This is a graph layout tool. Like DaVinci, Graphlet was used for visualization of small TGraphs.
14. Grok from the PBS tool kit, by the University of Waterloo. This a graph manipulation system, which based on Tarski's algebraic operators for typed graphs (rela-

tions composition, union, intersection, etc., along with transitive closure) [19]. Grok reads and writes TA as well as other notations including RSF triples.

15. GUPRO browser, by University of Koblenz. This browser provides navigation through multi-file and multi-language source codes according to a given conceptual model. This browser operates on TGraphs, created by suitable parsers.
16. GUPRO parsers, by University of Koblenz. To represent multi-language software systems in a common repository, parsers for these languages have to interoperate. In [23] the TGraph based approach for managing multi-language repositories in GUPRO is given.
17. GUPRO Query, by University of Koblenz. This is a universal graph query system. In the domain of software reengineering it is used for querying various software systems represented by TGraphs.
18. JViews. These Java software components from ILOG have been used used to display PROGRES graphs and serves as the user interface of generated diagram editors.
19. LSEDIT (LandScape EDITor) from PBS tool kit, by Universities of Toronto. This is a software architecture navigator. It inputs architectural descriptions in TA. It will visualize any graph presented in TA.
20. Matlab Stateflow by Mathworks and CTE by Daimler Chrysler. These tools are linked together using the PROGRES graph transformation engine in order to build an integrated model-based testing workbench.
21. PLIX parser, by University of Victoria. This is a front end that analyzes a PL/I dialect used at IBM, and produces RSF triples, which are to TA.
22. RIGI viewer, by University of Victoria. TA has been transformed to RSF, which is read by the RIGI graph viewer.
23. RigiParse C parser from Rigi tool kit, by University of Victoria. This is a front end for C, which produces RSF triples, which in turn have been transformed to TA.
24. SNIFF+, by Wind Rivers Systems, Inc. SNIFF+ supports navigation through facts about a C program. Jeromy Carriere of the Software Engineering Institute used SNIFF+'s API to extract facts about Linux, stored these facts in TA, and then analyzed them using a relational data base.
25. TKSee (software comprehension tool), by University of Ottawa. This is a navigator for facts about programs. It uses a schema called TA++ specified in TA for a generalized model of procedural languages. It inputs facts about programs (notably, MITEL Pascal programs) in TA and supports flexible interactive querying of these facts [29].

We could also list the extensive set of software systems which have been interconnected using RSF, which is closely related to TA. Our purpose in presenting this list of interacting tools is to demonstrate both the necessity of interoperable tools in reengineering and the practicality of using GXL to enable interoperability between existing software tools.

6 Levels of Software Analysis

There is a spectrum of levels of granularity at which programs are analysed, from the low level details of expressions up to high level architectural features [24, 10, 1]. We will illustrate this by considering the end points of this spectrum, and will show how TA and TGraphs have been used at these levels, to indicate how GXL can as well be used at these various levels.

6.1 AST Level

At the lowest or most detailed level, essentially all information about the source program is extracted. This is commonly called the AST (Abstract Syntax Tree) level.

As an example at the AST level, the Datrix parser analyzes C++ programs and emits an AST for them in TA [28]. Figure 3 shows a simplified version of the typed graph produced by the Datrix parser for a C++ assignment statement.

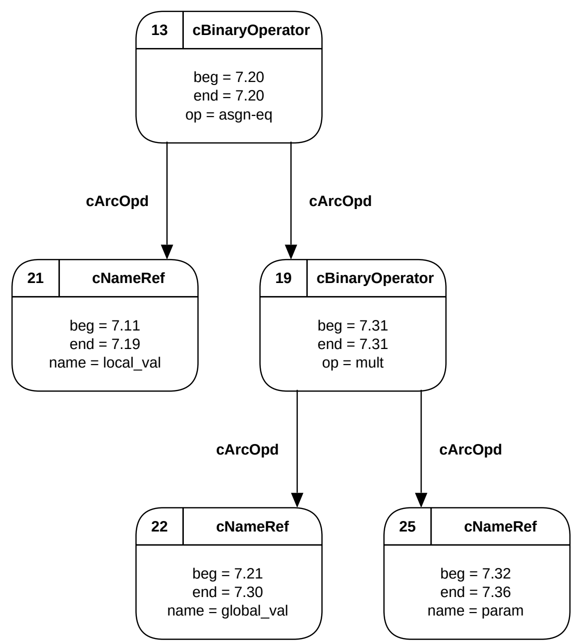

Figure 3. Datrix AST (extract) for the C++ statement: $local\_var := global\_var * param$

In the Datrix C++ AST (figure 3), the node identifiers are distinct integers (*13*, *21*, *19*, *22* and *25*). The named reference nodes (type *cNameRef*) have three attributes: *beg*, *end*, and *name*. The *beg* and *end* attributes give the location of the reference in a source file; for example, *local_var* occurs in file number *7*, beginning at character *11* and ending at character *19*. The binary operator nodes (type *cBinaryOperator*) have an operator attribute *op*; in particular, node *13*'s operator is *asgn-eq*, i. e., it represents an assignment. Node *19*'s operator is *mult*, i. e., multiply. (The documentation of the Datrix parser gives the description of the entire AST for C++ in TA [28].) Although the AST in general has many types of edges, this fragment has only one kind of edge, namely the operand arc (*cArcOpd*). It is straightforward to specialize the Datrix C++ AST to handle Java, and Bell Canada proposes to do so.

In a similar way TGraphs have been used for fine grained modeling of C programs. Whilst the Datrix C++ parser provides a tree-like representation, the C parsers used in GUPRO supply directed, node and edge attributed, node and edge typed, ordered, acyclic graphs. In these graphs objects (such as variables), which are modeled as leaves in ASTs, are only represented once. Different occurrences (declaration, definition, usage) of those objects are modeled by edges of different types (cf. [10] for examples). These TGraphs provide the base for data- and control flow analysis and the calculation of program slices for software security certification.

6.2 Architectural Level

The highest level, called the architectural level, commonly concentrates on entire source files, and clusters files into subsystems and subsystems into larger subsystems. The facts (edges and nodes) that define this clustering into subsystems do not come from code itself, but rather are provided manually or by special clustering tools.

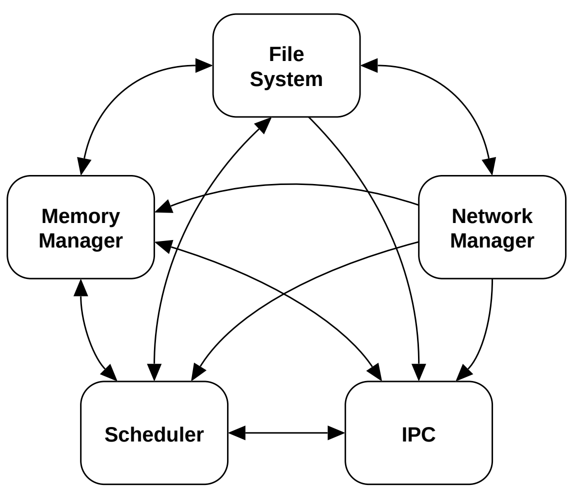

Figure 4. Architectural level, showing the top level subsystems of the Linux kernel

As an architectural example, figure 4 shows the top level subsystems of Linux [2]. Although only one type of edge (Call) is shown in figure 4, Bowman's analysis of Linux at the architectural level actually contains three types of edges: *Call*, *Reference* and *ImplementBy*. (*ImplementBy* means there is a header in one subsystem whose body is in another subsystem.) The edges in figure 4 were determined from lower level edges extracted by CFX. These edges were "lifted" by a Grok fact manipulator to the level of subsystems.

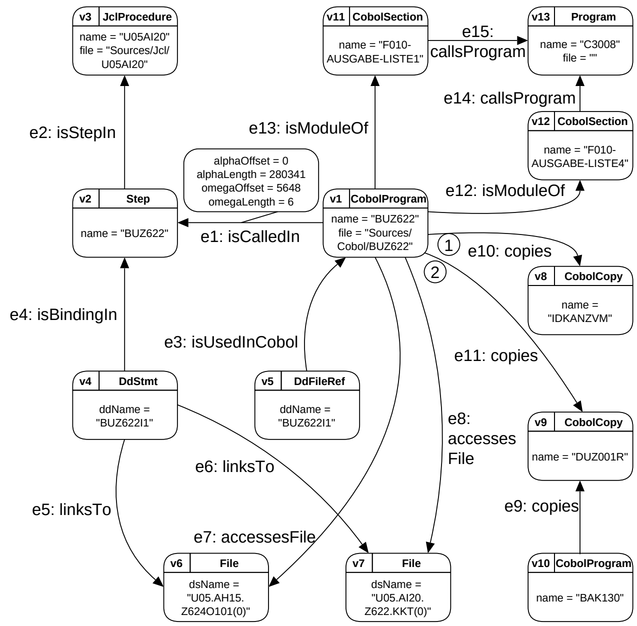

Figure 5. Architectural level, showing an extract from the Volksfürsorge application landscape

Figure 5 shows a directed, node and edged attributed node and edge typed TGraph depicting an extract of the software application landscape of Volksfürsorge [27]. The TGraph shows parts of the embedding of CobolProgram *BUZ622* into the whole system. It is called by Step *BUZ622*, accesses two files, includes two CobolCopies and contains two CobolSections which call an external Program. The ordering of the two copies edges incident to CobolProgram *BUZ622* indicates that CobolCopy *IDKANZVM* is included before *DUZ001R*. All vertices in this example are attributed with names (*name*, *ddname*, *dsname*) of the modeled source code artifacts. Furthermore, all edges carry coordinate attributes (only depicted for edge *e1*) that link to the concrete positions in the source code. TGraphs like the one shown in figure 5 were derived by various parsers extracting architecture related information from sources of the participating languages. Subgraphs for each parse unit were integrated into the total graph [23].

These examples demonstrate using TA and TGraphs for analyzing software systems at different levels of abstraction. We have discussed the levels of facts used in the analysis of programs, to show that GXL is suited to handling these various granularities.

7 Specifying Syntax of XML Stream in DTD

We will now explain how the syntax of GXL XML streams is specified. The standard notation for specifying the syntax of an XML stream is DTD (Document Type Definition) [46]. DTD is used to specify the nesting of tagged blocks, as well as parameters within tags.

Figure 6 gives a simplified version of the DTD for GXL (cf. [10]). The complete GXL DTD representing the current state of standardizing GXL can be found at `http://www.gupro.de/GXL`. In the DTD, parameters marked as #REQUIRED must be given in the XML stream, while parameters marked as #IMPLIED are optional. #CDATA means character string data and #NMTOKEN denotes names. An ID locates a part of the XML that can be referenced (by an IDREF) later in the XML stream.

```
<!ELEMENT gxl (node | edge)*>
<!ATTLIST gxl
    schema      NMTOKEN       #REQUIRED
    edgeids     (true | false) "false" >
<!ELEMENT node (attr)* >
<!ATTLIST node
    id          ID            #REQUIRED
    type        NMTOKEN       #IMPLIED
    edgeorder   IDREFS        #IMPLIED >
<!ELEMENT edge (attr)* >
<!ATTLIST edge
    id          ID            #IMPLIED
    type        NMTOKEN       #IMPLIED
    begin       IDREF         #REQUIRED
    end         IDREF         #REQUIRED >
<!ELEMENT attr EMPTY >
<!ATTLIST attr
    name        NMTOKEN       #REQUIRED
    value       CDATA         #IMPLIED >
```

Figure 6. DTD (Simplified Specifying the Syntax of the XML Stream for GXL)

The first line of figure 6 specifies that a stream for GXL consists of zero or more nodes and edges. The third line specifies that there must be a *schema* (this specifies the types of nodes and edges, etc., as explained in section 7.2). The fourth line requires that the parameter *edgeids* be set to *true* or *false* (default), indicating whether edges in the graph have identifiers on their edges (cf. the graph given in figure 5). Next, each node is specified to have zero or more attribute elements (attr), as a well as parameters called *id*, *type*, and *edgeorder*. For example, in figure 1 (see also figure 2), the top left node has *id = "P"* and *type = "Proc"*. When edge identifiers are used, the *edgeorder* parameter for nodes can be used to specify the order of edges entering and leaving a node. This is necessary, e. g. for representing the order of including copy books in Cobol (cf. figure 5). Next, an edge is specified to have zero or more attributes, as well as parameters called *id*, *type*, *begin* and *end*. For example, the top edge in figure 1 (see also figure 2), has a beginning (*begin*) node of *P*, and *end* node of *Q* and a *type* of *Call*. The *id* parameter for an edge is optional. It is essential when there is more than one edge of a given type between the same two nodes. The final part of the DTD specifies that each attribute (for a node or for an edge) has a name and a value. For example, the top left node in figure 1 has an attribute named *File* whose value is *main.c*.

The DTD in figure 6 has been simplified for presentation in several ways. Attributes can actually be structured values, and not just strings. Attributes for a given node or edge can also be given separately from the description of the node or edge by means of a nodeextension or an edgeextension. In this case, nodes are referenced by their *id* whereas edges are referenced (dependent on *egdeids*) either by their *type* with their *begin* and *end* attributes, or by their *id*. The sketched extension mechanism allows to exchange different aspects (views) of one program graph representation as different XML streams. For instance it is possible to exchange different layouts of the same program graph in the form of different GXL extensions of the same basic GXL stream.

A DTD specification such as this one can be used to check the format of the XML stream. The University of Koblenz has developed a generic parser for GXL based on this DTD, that invokes pluggable semantic actions for processing nodes, edges, and attributes. There are also various tools grouped around XML for analyzing and manipulating XML documents. DOM (Document Object Model, `http://www.w3.org/DOM/`) offers among other things a standardized tree-based interface for accessing and manipulating XML documents. Other approaches offer event based interfaces for parsing XML documents reacting on tags of XML document, e. g. SAX (Simple API for XML, `http://www.megginson.com/SAX/`) or non validating parsers, which ignore DTDs e. g. LARK (`http://www.textuality.com/Lark`). Furthermore XML documents can easily be transferred into other textual formats by XSL (Extensible Stylesheet Language). These tools support straightforward implementation of translation from GXL to existing formats such as TA, TGraphs, PROGRES RSF, and RPA.

7.1 Multi-Edges and Ordering

Sometimes it is natural to have more than one edge from a given node x to another given node y. Typed graphs allow more that one edge between nodes x and y, if the edges have different types. RPA [35], which deals with typed graphs, formalizes the notion of counts on edges, which can be used to record multi-edges of the same type from x to y. These counts can be modeled in (unextended) typed graphs by having an edge attribute named count. Note that this extension does not handle the situation of different attributes on the multi-edges.

TGraphs [7] have been designed to handle a very general and flexible class of graphs, incorporating the features of almost all graph models. TGraphs are typed graphs that allow multiple edges of a given type between two nodes. This is possible in TGraphs because the edges necessarily have an identifier *id*. GXL has been specifically designed to allow for this possibility. As an example, these multi-edges are useful so that several distinct edges of type Call can link procedure *P* to procedure *Q*. These edges would be distinguished by distinct identifiers.

TGraphs also provide modeling of ordered graphs with specified orders for nodes, edges, and edges incident to a given node. Modeling with ordered graphs is necessary for various graph algorithms preserving a predefined order of traversal. In reengineering ordered graphs are useful for modeling sequences of relationships like ordering parameter lists or ordering includes (cf. figure 5). The order of incident edges is given by the optional *edgeorder* attribute.

7.2 Specifying Schema of Graph by Class Diagrams

Since software is written in various programming languages and since we deal at various levels of granularity, it is inevitable that the types and attributes in our typed graphs vary according to our purpose. GXL deals with this variability the same way that relational data bases deal with it, namely, by means of schemas that specify the form of the graph data. In the case of facts about software, our schemas are quite changeable; for example, we change the schema when we lift facts between levels of abstraction. This is different from classical data bases, which have schemas that rarely change.

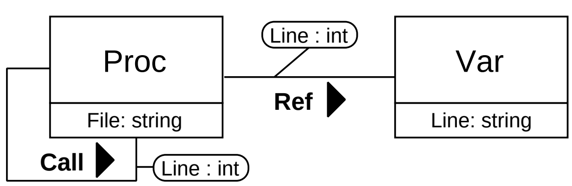

Figure 7. A schema for graphs like the one in figure 1

A common way to specify the schema for a typed graph is by means of (Extended) E/R (Entity/Relation) diagrams or UML class diagrams [10]. Figure 7 gives an UML diagram, or schema, that specifies the general form of the graphs like the one in figure 1. This figure specifies that *Proc*'s can call *Proc*'s and can reference *Var*'s. It specifies that *Proc*'s have a *File* attribute, *Var*'s have a *Line* attribute and that *Call* and *Ref* edges have a *Line* attribute.

Since class diagrams represent structured information, they can be represented as a typed graph as well. Regarding a graph schema defining a schema for schemata (metaschema) each class diagram can be mapped to a graph. [10] proposes such a metaschema which also allows inheritance of node and edge types. By representing schemata by graphs, we can directly represent them in GXL, as XML streams. This allows us to transmit the schema in XML along with the graph itself from one tool to another. This in turn makes tools independent of structural details such as the specific types and attributes in a graph, yielding tools which are parameterized by the schema.

We have described two levels of structure for GXL. The first contains the GXL stream to represent a typed graph having arbitrary types of nodes and edges and arbitrary attributes. The second contains, the GXL stream to represent the graph structure (i. e. the types of nodes and edges and particular attributes of this graph) given by an UML class diagram. These two levels reflect the fact that many tools are designed to handle arbitrary typed graphs, but they can be conveniently configured to deal with particular kinds of typed graphs as specified by the class diagram.

8 Criteria for Success of an Standard Exchange Format

Criteria for the success for software exchange formats have been given in [31, 25, 1]:

A. works for several levels of abstraction (e. g. AST, structure, architecture),
B. works for several languages (e. g. C, C++, Java, PL/I, RPG),
C. works for multi-million lines of code (e. g. 3 to 10 MLOC),
D. maps between entities/relationships and source code statements (e. g. line number or AST node),
E. works for static and dynamic dependencies,
F. is incremental (i.e., one subsystem at a time can be added),
G. has universality (e. g. used by others, IEEE standard),
H. has extensibility.

We feel that GXL does a good job of satisfying these criteria. By means of examples, we have explained how GXL can meet criteria A, B and C. Attributes can be used to give mappings (criterion D). Dynamic dependencies can be collected, as edges, by a profiler and stored in GXL, analogous to the way in which static dependencies are collected by a parser (criterion E). Each subsystem can have its own graph, represented in its own GXL stream; these can be merged by a graph manipulator such as Grok to create a combined graph, also represented in GXL (criterion F). Our plan is to encourage use of GXL by others and to have a recognized standard, e. g. an IEEE standard (criteria G). At ICSE 2000 Workshop on Standard Exchange Formats (WoSEF 2000) GXL was accepted as a possible standard graph exchange format by numerous reengineering and graph transformation research groups from industry and academics [40].

Since GXL can handle arbitrary typed graphs, whose form is given by schemas, it can handle a rich class of usages, both inside and outside the re-engineering community (criterion H). The GXL format proposed in this paper is not only restricted to tools in reengineering. Being a general graph exchange format, GXL can also be applied to other areas in software engineering like interchanging models between CASE tools for exchanging graphs between graph transformation systems. GXL has been designed in such a way that future extensions are feasible for handling different types graphs, such as hypergraphs or hierarchical graphs which are quite popular in the visual language as well as in the graph transformation community.

It should be mentioned that the approach to information exchange we are proposing has some shortcomings, which we will now discuss briefly. The size of exchanged GXL files will be somewhat large, due to XML syntax and the length of tags and parameter names. Perhaps alternate, short tag names should be allowed. Standard compression techniques will be helpful. But in contrast to exchange formats based on XMI [34] GXL is substantially less verbose. For every schema, XMI documents require an own DTD which can be generated automatically from an UML class diagram using the Meta Object Facility (MOF) [33]. Whilst these DTDs define an deep hierarchy of nested tags, GXL only uses one small DTD with few tags. API to access information about a program is sometimes considerably more efficient and convenient than a stream such as GXL. It is our position that such APIs are necessary components of analysis systems, and will exist to provide rapid access to information which is exchanged using a stream notation such as GXL.

In the end, GXL will be successful when many tools tools are interconnected by it in a way that provides valuable combined services. This will happen if the teams that develop these tools work together in a spirit of cooperation. Our plan is to begin by using GXL to interconnect the GUPRO tools (based on TGraphs), the PBS and related tools such as the Datrix C++ parser (which currently use TA) and the PROGRES tool. We then hope to expand, with the help of other teams, to interconnect with other well known tools. We invite other teams to participate.

9 Conclusion

This paper proposes that software analysis tools should use a common Software Exchange Format (SEF) to facilitate the interoperation of these tools. A particular graph-based, XML-based SEF, called GXL is proposed for this purpose. This SEF is a slightly modified version of the GraX notation, designed to conveniently handle transcribed TA, RSF, RPA and PROGRES data streams. These approaches are widely used for exchanging reengineering data. The formal definitions of the typed graphs in TA, PROGRES and GraX are essentially the same, and can be smoothly meshed. SEFs such as TA have been proven by using to them in the analysis of large software systems. This usage indicates that GXL can scale up to handle large industrial software. It is time to advance the state of practice in software analysis tools by standardizing on an exchange format such as GXL.

References

[1] I. Bowman, M. Godfrey, and R. Holt. Conecting Architecture Reconstruction Frameworks. in *First International Symposium on Constructing Software Engineering Tools (CoSET99)*. 1999.

[2] I. Bowman, R. Holt, and N. Brewster. Linux as a Case Study: Its Extracted Software Architecture. in *International Conference on Software Engineering*. Los Angels, 1999.

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

[4] K. Cremer. *Graphbasierte Werkzeuge zum Reverse Engineering und Reengineering*. DUV, Wiesbaden, 2000.

[5] P. Dahm. Parser Description Language, An Overview. in [8], 137–156. 1998.

[6] P. Dahm, J. Ebert, and C. Litauer. Das Graphenlabor. in [8], 67–84. 1998

[7] J. Ebert and A. Franzke. A Declarative Approach to Graph Based Modeling. in E. Mayr, G. Schmidt, and G. Tinhofer, (eds.). *Graphtheoretic Concepts in Computer Science*, LNCS 903. Springer, Berlin, 38–50. 1995.

[8] J. Ebert, R. Gimnich, H. H. Stasch, and A. Winter, (Hrsg.). *GUPRO — Generische Umgebung zum Programmverstehen*. Fölbach, Koblenz, 1998.

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

[10] 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, 89–98. 1999.

[11] J. Ebert, R. Süttenbach, and I. Uhe. Meta-CASE in Practice: a Case for KOGGE. in A. Olivé and J. A. Pastor, (eds.). *Advanced Information Systems Engineering, 9th international Conference, CAiSE'97*, LNCS 1250. Springer, Berlin, 203–216. 1997.

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

[13] Edison Design Group, Inc. *C++ Front End. Internal Documentation (Version 2.43)*. http://www.edg.com/cpp.html, 1999.

[14] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook on Graph Grammars: Applications, Languages, and Tools*, volume 2. World Scientific, Singapore, 1999.

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

[16] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

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

[18] H.-J. Hofmann. GenVis - Ein generisches Werkzeug zur Visualisierung von Programmstrukturen. Diplomarbeit, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, April 1998.

[19] R. Holt. Structural Manipulations of Software Architecture using Tarski Relational Algebra. in [47], pages 210–219. 1998.

[20] ISO/IEC15291. Information technology – Programming languages - Ada Semantic Interface Specification (ASIS). `http://www.acm.org/sigs/sigada/wg/asiswg/`, 1999.

[21] J. Jahnke and A. Zündorf. Applying Graph Transformations to Database Reengineering. in [14], 267–286. 1999.

[22] M. Kamp. GReQL, Eine Anfragesprache für das GUPRO-Repository, Sprachbeschreibung (Version 1.2). in [8], 173–202. 1998.

[23] M. Kamp. Managing a Multi-File, Multi-Language Software Repository for Program Comprehension Tools – A Generic Approach. in U. De Carlini and P. K. Linos, (eds.). *6th International Workshop on Program Comprehension*. IEEE Computer Society, Washington, 64–71. June 1998.

[24] R. Kazman, S. Woods, and J. Carrière. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. in [47], 154–163. 1998.

[25] R. Koschke, J.-F. Girard, and M. Würthner. An Intermediate Representation for Integrating Reverse Engineering Analyses. in [47], 241–250. 1998.

[26] R. Krikhaar. Reverse Architecting Approach for Complex Systems. in *Proceedings of the International Conference on Software Maintenance (ICSM '97)*. IEEE Computer Society Press, Los Alamitos, 4–11. 1997.

[27] B. Kullbach, A. Winter, P. Dahm, and J. Ebert. Program Comprehension in Multi-Language Systems. in [47], 135–143. 1998.

[28] C. LeDuc and B. Lague. DATRIX Abstract Semantic Graph Reference Manual. Bell Canada, 1999.

[29] T. Lethbridge and 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.

[30] M. Werner and M. Fröhlich. daVinci V2.0.x Online Documentation. `http://www.tzi.de/~davinci/docV2.0/`, June 1996.

[31] 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.

[32] M. Nagl, (ed.). *Building Thightly-Integrated (Software) Development Environments: The IPSEN Approach*, LNCS 1170. Springer, Berlin, 1996.

[33] Meta Object Facility (MOF). `http://cgi.omg.org/cgi-bin/doc?ad/99-06-05.pdf`, June 1999.

[34] XML Meta Data Interchange (XMI). `http://cgi.omg.org/cgi-bin/doc?da/99-10-02.pdf`, October 1999.

[35] 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.

[36] F. N. Paulish. *The Design of an Extendible Graph Editor*, LNCS 704. Springer, Berlin, 1991.

[37] D. Petriu and X. Wang. Deriving Software Performance Models from Architectural Patterns by Graph Transformations. in H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, (eds). *Theory and Application of Graph Transformations - Proc. 6th Int. Workshop TAGT'98*,LNCS 1764. Springer, Berlin, 475–488. 1999.

[38] A. Radermacher. *Tool Support for the Distribution of object-based applications*. PhD thesis, RWTH Aachen, March 2000.

[39] A. Schürr, A. J. Winter, and A. Zündorf. PROGRES: Language and Environment. in [14], pages 487–550. 1999.

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

[41] I. Sommerville. *Software Engineering*. Addison-Wesley, London, 1995.

[42] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, 2 edition, 1992.

[43] A. Tarski. On The Calculus of RelationsThe Journal of Symbolic Logic. *The Journal of Symbolic Logic*, 6(3):73–81, 1941.

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

[45] M. van den Brand, A. Sellink, and C. Verhoef. Current Parsing Techniques in Software Renovation Considered Harmful. in *Proceedings of 6th international Workshop on Program comprehension*. IEEE, Los Alamitos, 108–117. 1998.

[46] Extensible Markup Language (XML) 1.0. W3c recommendation, W3C XML Working Group, `http://www.w3.org/TR/1998/REC/xml/19900210`, February 1998.

[47] *Fifth Working Conference on Reverse Engineering*. IEEE Computer Society, Los Alamitos, 1998.

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