

Graph Technology in Reverse Engineering The TGraph Approach

Jürgen Ebert **Volker Riediger**

University of Koblenz-Landau
Institute for Software Technology
Universitätsstr. 1
56070 Koblenz, Germany
ebert,riediger@uni-koblenz.de

Andreas Winter

Johannes-Gutenberg-University Mainz
Institute for Computer Science
Staudingerweg 9
55128 Mainz, Germany
winter@uni-mainz.de

Abstract: TGraphs are directed graphs with typed, attributed, and ordered nodes and edges. These properties leverage the use of graphs as models for all kinds of artifacts in the context of software reengineering. TGraphs are accompanied by metamodel based technologies to define, manipulate, analyze, query, visualize, and transform graphs.

This paper summarizes the work accomplished around TGraphs and graph based modeling over the last decade. We introduce TGraphs as a versatile and expressive formalism and demonstrate their well-suitedness for solving reverse engineering problems.

1 Introduction

Reverse Engineering is the process of analysing given systems to identify its components and their interrelationships and to create an explicit representation in another form or at a higher level of abstraction [CC90]. As such it is a central and an almost indispensable activity in software reengineering.

During the last decade various techniques supporting reverse engineering were developed. The *Extract-Abstract-View-Pattern* [Ti95] provides a coarse reference architecture for tools implementing these techniques. Facts on source code artifacts are *extracted* into a software model. Using various analysis techniques, these data are *abstracted* to provide a deeper understanding of the software systems. Abstractions of the system are then *viewed* by appropriate visualisation means.

A major challenge in developing reverse engineering tools is to provide an efficiently analysable structure to represent abstract software models [CCdC92]. To deal with industrial scale software, these structures have to cope with huge amounts of strongly connected data. Several *representation paradigms* for abstract software models can be identified, the most popular being logics and logical databases (DATA.Tool [CCdC92], CodeQuest [HVd06]), sets and relations (RPA [OvFK98], SWAG Kit [Hol98], Crocopat [BNL05],) relations in the sense of relational databases (CIA [CNR90], CIAO [CFKW95], SoftANAL [SD99]), hybrid structures combining SQL-databases with internal structures (Dali [KC99]), and graphs (Bauhaus [RVP06], Columbus [FBTG02], GUPRO [EKRW02], RE-

forDI [Cre00], Rigi [Won98], Shrimp [SM95]). All these different forms of code representation have their advantages and their disadvantages.

In the following, we claim that *graphs play a central role* in the sense that they have proven to be a versatile and expressive formalism for fact representation and fact processing. At the same time, graph representations can easily be transformed to other representation formalisms [EKW99]. The latter property makes them well suited to act as means for data exchange between reengineering tools. Thus, the GXL Graph eXchange Language [HSEW06], the widely accepted XML dialect for reengineering tool interoperability, uses graphs as common structure.

The type of graphs discussed in the following are *TGraphs* [EF95], i. e. directed graphs whose vertices and edges are typed, attributed, and ordered. Originally, TGraphs were defined to represent abstract syntax of functional programming languages [Ebe85], and were later applied to the representation of visual languages in the KOGGE MetaCASE-tool [ESU97]. Their properties leverage the use of graphs as models for all kinds of artifacts. They are especially suited to software reengineering purposes, since reengineering artifacts are almost always discretely structured artifacts. Additionally, TGraphs are accompanied by *metamodel based technologies* to define, manipulate, analyse, query, visualise and transform TGraphs.

This paper summarises the use and impact of the *TGraph Technology in Reverse Engineering* and its application in the *GUPRO (Generic Understanding for PROgrams)* program comprehension framework [EKRW02]. The description uses a simple Java program as running example and introduces the modeling, extraction, abstraction, and visualisation support used in GUPRO and applied in various reverse engineering projects during the last decade. The paper also acknowledges the numerous theses written by our students as important bricks of the whole approach.

Section 2 shortly introduces TGraphs using abstract syntax graphs as an example and lists the fact extractors used to transform source code to TGraphs. Section 3 focuses on the query language GReQL and describes how querying is being used as an enabling technology in reverse engineering. Section 4 deals with the visualisation of content stored in TGraphs and introduces several browsing views. Section 5 explains the modeling theory behind TGraphs. The schema dialect grUML and its formal semantics is sketched shortly. This section also describes the (J)GraLab API for schema and graph implementation. Finally, section 6 gives an overview on reverse engineering applications that have been tackled with the TGraph approach.

2 Representation of source code artifacts in graphs

A first step in reverse engineering activities is the preparation of source code, which is usually the only reliable basis for analysing legacy systems. Depending on the objectives of the analysis, all relevant *facts* contained in sources are *extracted* and stored in appropriate repositories. Here, TGraphs are used for representing these facts.

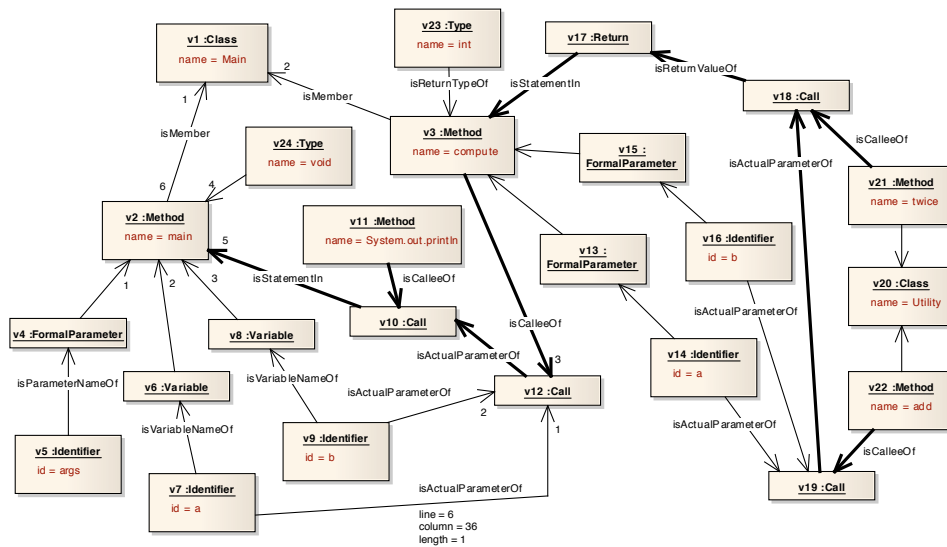


Figure 1: Java program of Listing 2 represented as TGraph (simplified)

Figure 1 shows an extract of a *sample TGraph* representing the Java code depicted in Figure 2. Its fine grained structure is intended to facilitate understanding the relationships between all code elements. The graph shows usages of all TGraph features.

```

1 public class Main {
2     public static void main(String[] args) {
3         int a = 26;
4         int b = -5;
5         System.out.println(compute(a, b));
6     }
7     private static int compute(int a, int b) {
8         return Utility.twice(Utility.add(a, b));
9     }
10 }
11
12 public class Utility {
13     public static int add(int x, int y) {
14         return x + y;
15     }
16     public static int twice(int x) {
17         return 2 * x;
18     }
19 }

```

Figure 2: Java Code

(1) Nodes and edges are *typed*, e.g. nodes of type Method represent methods and isReturn-type-edges connect methods with their appropriate return Types. (2) Various nodes are *attributed*, e.g. Method nodes carry their method's names. Edges also carry attributes which link to source code. In Figure 1 those attributes are only shown for the isActualParameterOf edge, modeling variable a's occurrence as the first parameter in the call of compute. (3) This example also shows *ordering* of incidences. The small numbers reflect the succession of the actual parameters in the call. (4) All edges are *directed* and point to those nodes which subsume other code concepts, e.g. all edges representing containment of methods to a given class are directed towards the Class node. Each *entity* to be modeled is only represented once by one single node. The TGraph approach uses edges to model *occurrences* of elements in another. The occurrence of an entity at some location in the source code is depicted by an edge connecting the corresponding node to the node representing the location.

Each *entity* to be modeled is only represented once by one single node. The TGraph approach uses edges to model *occurrences* of elements in another. The occurrence of an entity at some location in the source code is depicted by an edge connecting the corresponding node to the node representing the location.

senting its location of use. Variable `a`, declared in method `main`, is represented by node `v7`. Declaration and initialization (line 3) and use (line 5) of `a` are modeled by edges. Since formal parameter `a`, declared in `compute` (line 7), is a different entity, it is represented by node `v14`, having the same name.

Although the graph is pruned for visualisation reasons, it allows to motivate and demonstrate the use of TGraphs in reverse engineering and program comprehension. For GUPRO there is an extractor for Java [BV08] which *locally* transforms Java code to fine-grained abstract syntax graphs. The complete TGraph representing the code of Figure 2 has 78 vertices and 119 edges, covering methods and declarations completely and a more detailed representation of the abstract syntax. Additionally, the extractor can work in the so-called *complete* mode where the resulting graph also includes the relevant parts of the Java Runtime Environment and possibly imported third-party binary libraries. Those parts are computed by Java reflection and enlarge the graph to 7,132 nodes and 19,948 edges.

Analysing real-world programs leads to even bigger graphs. JGraLab [JGr], the underlying TGraph implementation, is capable of efficiently handling graphs with millions of nodes and edges. Figure 3 shows the node and edge counts of TGraphs generated by extracting fine grained representations of ANTLR, an open-source parser generator, and the JGraLab library itself [BV08]. Interestingly, the node count in complete mode can be substantially lower than in local mode while the edge count increases. This is a result of a far more sophisticated type analysis done by the Java extractor which merges nodes representing the same object, and then links more occurrences of program elements to their definitions.

Program	#Classes	LoC	Mode	Nodes	Edges
ANTLR	216	55,725	local	133,557	241,298
			complete	178,624	383,131
JGraLab	626	232,947	local	685,715	971,397
			complete	576,269	1,008,979

Figure 3: Graph size of real programs

Further fact extractors for converting source code to TGraphs were developed for a wide range of programming languages and languages systems. Fact extraction on an architectural level was provided in the initial GUPRO project with IBM scientific center and Volksfürsorge insurance company [KWDE98]. These fact extractors coped with a typical multi languages system covering COBOL, CSP, MVS/JCL, PSB, SQL (DDL and DML), and IMS-DBD sources on a coarse-grained level. Graph-merging techniques were applied here to enable incremental extension and update of a TGraph representing (parts of) the software landscape of Volksfürsorge [Kam98]. Fine-grained extractors on the abstract syntax level exist for Ada [KS01], C [Rie01]. Further extractors for C++ are based on the Columbus framework [FBTG02] and use their GXL export to generate TGraphs.

At the beginning of the decade, efforts on enabling interoperability between reengineering tools resulted in an XML based interchange format. The GXL Graph eXchange Language [HSEW06] is formally based on TGraphs, complemented by some straightforward features to support the representation of hierarchical graphs and hypergraphs. The univer-

salinity of TGraphs facilitated the development of a very generic exchange format, covering all representations practically used in reverse engineering tools. At the Dagstuhl Seminar on *Interoperability of Reengineering Tools* [EKM01] GXL was ratified as standard interchange format for exchanging reengineering related data and meanwhile is implemented by most of the tools sketched in Section 1.

3 Analyzing TGraphs

TGraphs constitute a well-defined formal *mathematical model* as well as an efficient *data structure* [Ebe87] providing a seamless approach for graph-based modeling and implementation. Many reverse engineering techniques can be based on *graph analysis* using graph algorithms and/or *graph querying*. On TGraphs, querying is used as enabling technology for many reverse engineering techniques.

The GraLab *Graph Libraries* for C++ [DW98] and Java [Kah06, JGr] provide efficient support for manipulating TGraphs including creating and accessing nodes and edges, setting and getting node and edge attributes, accessing node and edge types, traversing the graph, manipulating the order of incidences, retrieving incident edges for given nodes, etc.

Querying of TGraphs is realised by *GReQL* (Graph REpository Query Language) [KW99]. GReQL is a declarative expression language making extensive use of regular path expressions to denote relations between nodes and edges. It is designed as a pure query language, keeping the inquired graph unchanged. GReQL is generic in the sense that queries might refer to a given graph schema (cf. Section 5).

Figure 4 depicts a *sample GReQL query* and the corresponding query result with respect to the Java program in Figure 2 and the Java graph in Figure 1. It calculates all caller-callee pairs in the Java fragment. GReQL queries basically consist of three parts: the **from**-clause declares the relevant graph elements, the (optional) **where**-clause specifies additional constraints for the declared graph elements, and the **report**-clause describes the appearance of the query result.

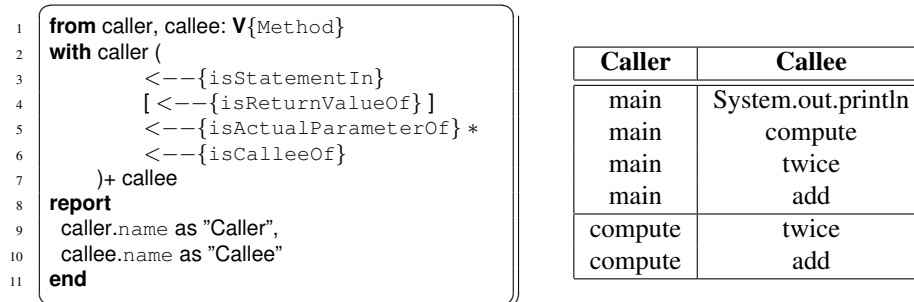


Figure 4: A simple GReQL Query

The path predicate in lines 2-7 uses a regular structured pattern describing the connection between caller and callee. This pattern summarizes all occurrences of method invocation

(direct call and invocation by method parameters or in return statements) used in the simple code example. To find a directly called method, one has to follow an `isStatementIn` edge in reverse direction leading to a `Call` node. This node is connected by a reverse `isCalleeOf` edge to the called method. Methods called in arguments of other method calls require additional `isActualParameterOf` edges. Here, calls in return statements follow an optional, reverse `isReturnValueOf` edge from a `Return` node to a `Call` node representing (parts of) the return parameters. Deriving also indirect method calls requires to calculate the transitive closure over this path pattern, depicted by the “+” in line 7. The relevant edges for calculating caller-callee pairs are marked in Figure 1 by darker arrows.

GReQL uses first order predicate logic over finite sets and constructive set theory. All GReQL queries are algorithmically accessible, i.e. they can be evaluated on a given TGraph in polynomial time. GReQL provides many relevant graph predicates (e. g. structural graph properties) as well as aggregation functions (e. g. *average*, cardinalities) and further graph specific functions (e. g. *degree*, *alpha*, *omega*). Polynomial execution time is guaranteed since quantifiers range only over finite domains and since all basic functions and predicates have either constant or linear complexity.

Experiences with GReQL in several projects (cf. Section 6) show that graph queries are a useful and versatile basis for analysing graphs. Thus, many aspects of software reverse engineering can be easily based on GReQL querying:

Software metrics help to measure and compare software characteristics, which are expressible in numbers. GReQL provides aggregation functions to calculate e. g. the number of edges pointing from call nodes to method nodes, indicating the fan out of that method. Counting graph elements combined with common algebraic operations can generally be used to specify software metrics. [Kie97] shows the successful application of GReQL to define software metrics for COBOL systems.

Cross References are mappings between certain graph objects. The interrelation between these objects can be specified by path-predicates. The query in Figure 4 is a sample for calculating cross references between methods. Since edges in TGraphs are traversable in both directions, cross references can be accessed from both sides. In the Volksfürsorge Project [KWDE98] these features were used to detect programs which included certain COBOL copy books.

Program Slicing has the goal of deriving minimal subparts of given programs that control the value of a variable at a given statement (backward slice) or are controlled by it (forward slice). Following the approach of [OO84] the computation of slices can be reduced to queries [Sch07].

Refactoring denotes the transformation of source code to make it more readable and maintainable without changing its semantics. Refactorings are thus doable by transforming the syntax graph of the program where usually many context conditions have to be tested to make the refactoring action secure. Queries are used to detect “bad smells” and ensure proper replacements [Fli06].

Preprocessor Statements make fine grained analysis of source code, for example in COBOL, C, C++, and PL/I systems, a very complicated task. While source code, visible to

the reverse engineer, contains *preprocessor input*, the program processed by the compiler is the *preprocessor output*. The relation between preprocessor input and output is characterized by non-reversible textual transformations which generally ignore the syntax of the programming language itself. TGraph and query based solutions for various preprocessor problems were developed in [KR01, Rie04]. For the C language, a special preprocessor creates the ordinary preprocessor output together with a so-called *fold graph*. This graph represents all preprocessor actions, from file inclusions to macro definitions and macro expansions. By specialized graph algorithms and graph queries, connecting the fold graph with the abstract syntax graph, even for preprocessed languages, results of fine grained analysis can be visualized in original source code.

Impact Analysis tries to estimate the effort for implementing feature requests, change requests, or bugfixes in large software systems. Clearly, querying with transitive closure computation can be used to compute the source locations subject to change. Additionally, not only the fact that a piece of code has to be touched but also the concrete *path* of changes in a project and the *length* of that path are important information to facilitate impact analysis. Efficient computation of such paths specified by regular path expressions was realized in [Ber03].

Cluster Analysis is a useful technique in *architecture recovery*. Cluster analysis algorithms traditionally work with matrix or table data. The decisions for membership of objects in certain clusters are based on distance measures. Unfortunately, most cluster analysis frameworks only define rather simplistic distance functions which are not sufficient for reverse engineering problems. Techniques to use TGraphs directly as data source for cluster analysis in the YALE¹ (Yet Another Learning Environment) [Rap] experimentation framework were developed in [Ber06]. Additionally, the problem to relate results of cluster analysis back to graph entities was solved.

[KW99] shows a wide range of GReQL queries used in reengineering in general and [LSW01] shows a set of queries used for comprehending C/C++/RDBMS-based systems.

Today, GReQL is provided by various interfaces: the GUPRO Reverse Engineering Workbench enables interactive querying. GReQL is also part of the (J)GraLab graph libraries to facilitate GReQL-based reasoning in C++ [DW98] and Java programs [Kah06, JGr]. For automated analysis, GReQL queries and graph manipulations can be part of programs in the scripting language *GReQLScript* [Kla07].

4 Visualisation

TGraphs can be visualised directly by using standard graph visualisation tools for rendering. For example, TGraphs can be directly exported as GraphViz [Gra] source files (including some layout adjustments); and GraphViz is capable of handling GXL files. Due to the huge size of TGraphs representing real software systems visualisation of all nodes and edges is often inadequate. The information stored in graphs has to be condensed, selected and appropriately presented to the software engineer. Graph visualisation in reverse

¹The YALE project was recently renamed to RapidMiner.

engineering also deals with presenting the source code behind the graphs. Usually maintenance programmers think in source code, not in graphs. Thus, graph visualisation in reverse engineering has to provide means to present mappings between graphs and code.

Again, querying can be used to compute the relevant parts of a software graph to be visualised. The results of GReQL queries can be rendered in several useful ways:

Table Views like the one in Figure 4 show the results in textual form. Additionally, the textual results can be exported in XML for processing in external tools.

Source code views visualise query results directly as highlighted regions in source code. The GUPRO program understanding workbench also includes displays of preprocessor actions by folding on arbitrary level of detail. Navigation in source files of huge projects is controlled by graph contents and context sensitive schema based queries. Figure 5 shows a GReQL query for a C program, the tabular query result, and a source code view of the result. To facilitate the visualisation of preprocessor input and output, preprocessor macro calls enclosed in triangles and can be expanded and folded [Rie04].

Graph views can be useful to visualise various aspects of a software system. Adequate presentation of graphs is strongly influenced by the modelling domain the graph is applied in, and is often dependent on the information in the graph elements as well as the graph structure. Domain specific graph layout specifications based on metamodels and graph queries [SRW06] allow flexible customised graph views. Arbitrary information can determine the presentation of graph elements, e.g. using software metrics to determine the size of nodes, or using type information to change colours.

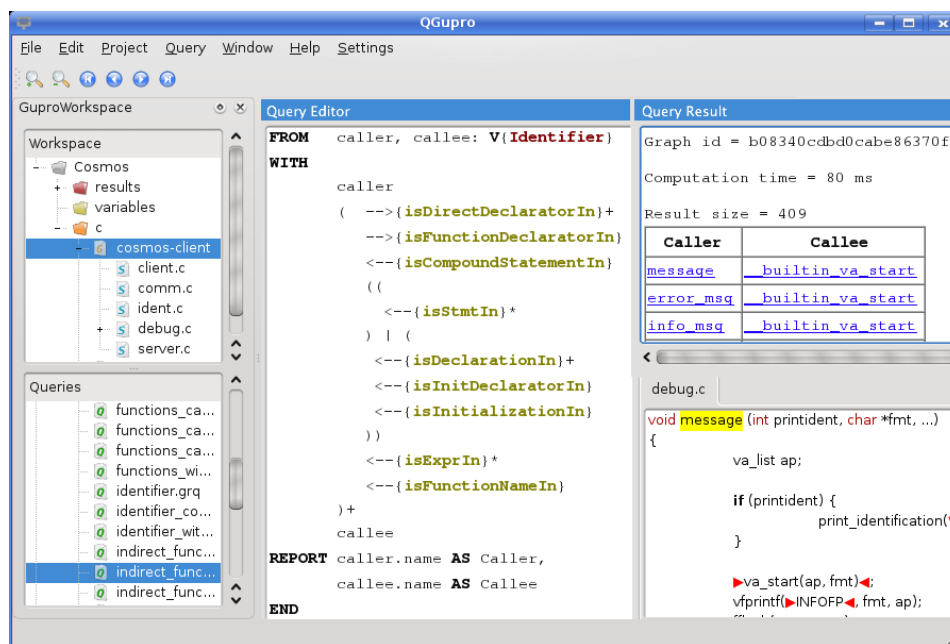


Figure 5: Screenshot of the GUPRO workbench

5 Graph based metamodelling

TGraphs are a very general representation means for representing program and reverse engineering information from *different viewpoints* and on *different levels of granularity*. They can be used for a fine-grained representation of the abstract syntax of concrete programs (cf. Figure 1), but they can also be used to represent more coarse-grained information like networks of method calls, containment structures, architectural views, and other higher-level views. Furthermore, they are also able to describe heterogeneous systems consisting of artifacts in different languages [KWDE98].

This versatility is achieved by adapting TGraphs to the respective viewpoint using *meta-models*. MOF (Meta Object Facility) [OMG06] provides an UML-based approach to meta-modelling. A MOF-like metamodeling hierarchy is used to specify classes of TGraphs formally. This formalization was termed EER/GRAL in the nineties ([EWD⁺96]) and was based on extended entity relationship diagrams and the Z-inspired constraint language GRAL [EF95]. Today, a sublanguage *grUML* (graph UML) of UML is used for this purpose, and constraints are formulated as boolean GReQL queries [BER⁺08].

Using grUML enables to define *classes of TGraphs* formally by *schemas*. grUML is a subset of UML class diagrams which has a formal TGraph semantics, i. e. grUML contains (only) those elements of UML which can be interpreted in graphs. Classes correspond to node types, associations correspond to edge types, specialization and generalization lead to type hierarchies, and attributes refine the information on node or edge types, respectively. Furthermore, multiplicities correspond to degree restrictions.

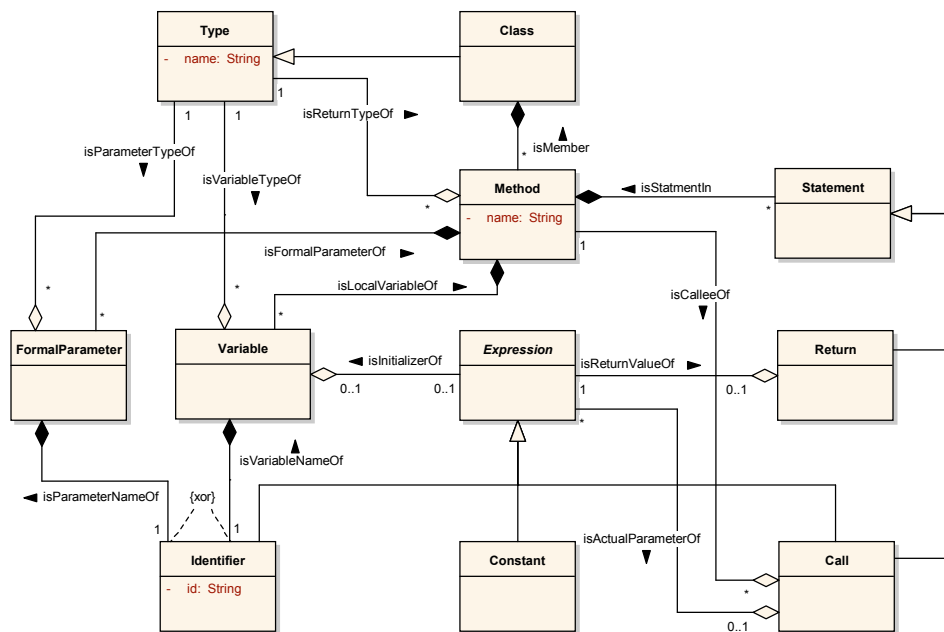


Figure 6: Simplified Java MetaModel for the graph in Figure 1

Figure 6 shows a simplified *sample grUML schema* which specifies the class of TGraphs for Java used in section 2. This schema describes classes (Class) as compositions of methods (Method) whose signatures are also modeled in detail. For simplicity's sake, statements and expressions are not elaborated further than needed for Figure 1. The complete, fine-grained syntax schema used by the Java extractor [BV08] is far more complicated and contains 89 node classes and 180 edge classes.

Schemas as metamodels written in grUML provide the ability to formally define all inputs, intermediate forms, and final results of reverse engineering algorithms and thus give a basis for defining and discussing the different forms of data used and produced. Since grUML metamodels define the data structure used by GraLab, grUML specifications are not only conceptual descriptions but seamlessly also type constraints kept by the implementation. The modeling power of this approach is slightly higher than EMOF [OMG06]. Thus, any EMOF-compatible tool can be used on top of GraLab.

Metamodeling with grUML leads to a similar *instantiation hierarchy* like in MOF [OMG06], but due to the elaborated structure of TGraphs a different metameta-model is being used. This metameta-model has a formal TGraph semantics. Its advantage is the fact that any metamodel according to it precisely defines a graph class and can simultaneously be used as a GraLab schema. The semantic description of grUML is based on the metameta-model and is compositional in the sense that the graph class is defined by composing partial instances of metameta-model classes to graphs.

Because TGraphs are strongly schema based, and the underlying schema language grUML has a formally defined semantics, TGraphs are an ideal means to support model driven engineering applications.

Model Driven Engineering (MDE) is a modern approach for software engineering, based on metamodeling. Various models are transformed into several intermediate models, and finally into source code. In near future, also software developed with MDE methods will be subject to reverse engineering.

Thus, on the one hand, reverse engineering technology should be capable of dealing with such software systems, and on the other hand, reverse engineering techniques should make (more) use of model driven approaches. TGraph technology can be used in both, in a model-driven development manner and for representing, analyzing and reverse engineering MDE systems.

TGraphs in MDE. The JGraLab technology enables model driven engineering: TGraphs can be specified by creating grUML schema diagrams with UML tools, and subsequently transforming these models into executable Java code. In a first step, the grUML model is stored as *XMI file*. XMI (XML Metadata Interchange [OMG07]) is a standardised format based on MOF for UML model exchange purposes. This model is transformed by an XML transformation into a TGraph schema file the so called *TG format*, a simple textual representation of grUML schemas. Finally, the JGraLab code generator transforms this schema into an object oriented API for graphs which allows convenient access in terms of the modeled domain.

MDE with TGraphs. MDE systems require flexible and customisable comparisons and transformations of models. Both problems can be solved with TGraphs and JGraLab: for the *model comparison part* we developed an integration with *SiDiff* [WN05], a generic UML model difference algorithm developed at the University of Siegen, Germany. Models kept in TGraphs can be compared in a user defineable way, e. g. to compute similarities or version differences. The *transformation part* is addressed by a direct coupling of the MOLA transformation engine [KBC04] created at the University of Latvia. Transformations specified in MOLA (Model Transformation Language) are compiled to Java executables which operate directly on JGraLab TGraphs. This MDE tool set is applied in the ReDSeeDS project (Requirements Driven Software Engineering System [ReD, Smi07]) where TGraphs are used as central model and fact repository.

6 Applications

TGraphs and their accompanying software comprehension techniques were used in various projects. Thanks to the metamodel-based genericity of the TGraph approach, these techniques were easily applicable to various reverse engineering and program analysis projects.

In the middle of the 1990s, the **Volksfürsorge insurance company** in Hamburg operated a heterogeneous software landscape consisting of about 6,000 units of PL/I, about 4,000 units of COBOL, and about 5,700 CSP applications having more than 75,000 components. These units were connected by approx. 25,000 JCL procedures and visualised using about 5,000 MFS-descriptions (Message Format Service). Database definitions are given in nearly 1,000 IMS-DBD units which are connected to the whole system by use of more than 1,800 PSB-specifications [KWDE98]. A major problem in maintaining this system was to detect and to consider interrelationships between artifacts of different programming languages. Funded by the German Federal Research Ministry (BMBF) and in cooperation with IBM Scientific Center, an appropriate GUPRO toolset was developed. The structure of the Volksfürsorge system was defined by an appropriate metamodel covering all relevant concepts of the participating programming languages [DFG⁺98]. An integrated set of fact extractors [Kam98] provided up to date export of software facts to the TGraph repository. Using the first version of GReQL, which was initially developed in this collaboration, various queries were provided to detect cross references in the Volksfürsorge system [KWDE98] and help maintenance programmers to better appraise impacts of software changes.

Major contributions of collaborations with the **German Federal Office for Software Security** (BSI), Bonn, to the application of TGraph technology in program analysis, was its application to *security analysis*. These works required fine grained TGraph representations of C and Ada programs. Adequate metamodels and related parsers for C [Rie01] and Ada [KS01] were developed. Analysis of preprocessed code and the adequate visualisation of query results required means to analyse systems on preprocessor output, but presenting the results on preprocessor input (cf. Source View in Section 4). The *fold-approach* [KW00, Rie04], based on a mapping between TGraphs representing preprocessor output (code graph) and macro structure (fold graph) enabled scalable views on preprocessed languages.

Analysing the **GEOS stock trading system** covering 2,364,652 lines of code in 6,279 source files made it possible to compare the TGraph-based query engine to Harry Sneed's SQL-based program analysis workbench CPPAnal [SD99]. Both systems did not differ in functionality, but the GReQL inherent ability to directly traverse TGraphs leads to much faster evaluation of those queries relating many different types of code components [LSW01] than comparable SQL queries.

TGraph-based metamodeling technique was also applied in collaboration with **Debeka Group**, Koblenz, to define the structure of JCL Jobs [Wid01]. This metamodel became part of the Debeka software repository structure implemented in ZEDER [JB06], and is used in a huge migration project.

Using **GXL**, the TGraph modeling and analysis facilities were also applied to provide interoperability between software engineering tools. In [WHW02] GReQL-based queries were combined with Ric Holts *grok* engine, implementing Tarski-Algebra [Hol98]. Parsing support for C++ provided by Columbus [FBTG02] was also made available to TGraph based software analysis. Exporting Bauhaus resource flow graphs [RVP06] to GXL and filtering them according to architectural viewpoints enabled the visualisation of reconstructed software architectures by UML diagrams using IBM's software architect [WW05].

7 Conclusion

The previous sections introduced the *TGraph approach* and showed various applications to solve reverse engineering problems. TGraph technology provides a versatile graph model and comes with elaborated grUML metamodeling facilities to enable extensive adaptability. Its implementation and analysis support by GraLab and GReQL make it practically available for the development of reverse engineering tools.

Next to their application in reverse engineering, TGraphs and their associated techniques were also successfully applied in defining metamodels for visual languages [Win00] and creating and using the KOGGE-MetaCase-Tool [ESU97]. Currently, they are applied as central fact repository in the ReDSeeDS Requirements Driven Software Development System [ReD, Smi07].

The TGraph related development reported in this paper evolved over the years and was driven by varying requirements from the different project contexts. Since seamlessness has been the main goal, all design decisions were done in the light of keeping the approach consistent and well-balanced. There are still some open requests for extending the approach, the most relevant being the claim for distributed graphs, hierarchical graphs and/or hypergraphs. The extension of the approach in these directions is the main challenge for the future.

Acknowledgments. We like to use this short recapitulation of our work on the occasion of the 10th Workshop Software Reengineering to express our gratitude to all who supported us in developing the TGraph technology. Thanks go to all our students, for valuable discussions on various aspects of graph theory, graph based modeling, and graph querying and their application to reverse engineering. We are also indebted to our stu-

dents for their considerable contribution to the development of GraLab, GReQL, and the GUPRO tools. We are grateful to all former and current members of our group who provided important contributions to enable and realize our ideas. Special thanks have to go to our project partners and funding organizations for giving us the chance to realise our approaches and for reminding us to develop practically usable and reasonable techniques.

References

- [Ber03] U. Berg. Berechnung von Pfadmengen in der Graph-Anfragesprache GReQL. Diplomarbeit, Universität Koblenz-Landau, Institut für Softwaretechnik, 2003.
- [Ber06] T. Bernd. Softwareclustering im Reverse Engineering. Diplomarbeit, Universität Koblenz-Landau, Institut für Softwaretechnik, 2006.
- [BER⁺08] D. Bildhauer, J. Ebert, V. Riediger, H. Schwarz, and S. Strauß. grUML — An UML-based Modeling Language for TGraphs. to appear in *Arbeitsberichte Informatik*, Universität Koblenz-Landau, 2008.
- [BNL05] D. Beyer, A. Noack, and C. Lewerentz. Efficient Relational Calculation for Software Analysis. *IEEE Transactions on Software Engineering*, 31(2):137–149, February 2005.
- [BV08] A. Baldauf and N. Vika. Java-Extraktor für GUPRO. Studienarbeit, Universität Koblenz-Landau, Institut für Softwaretechnik, 2008.
- [CC90] E. J. Chikofsky and J. H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, pages 13–17, Januar 1990.
- [CCdC92] G. Canfora, A. Cimitile, and U. de Carlini. A Logig-Based Approach to Reverse Engineering Tools Production. *IEEE Transactions on Software Engineering*, 18(12):1053–1064, December 1992.
- [CFKW95] Y.-F. Chen, G. S. Fowler, E. Koutsofios, and R. S. Wallach. Ciao: A Graphical Navigator for Software and Document Repositories. In *International Conference on Software Maintenance, Proceedings, IEEE Computer Society*, pages 66–75. 1995.
- [CNR90] 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.
- [Cre00] K. Cremer. *Graphbasierte Werkzeuge zum Reverse Engineering und Reengineering*. DUV, Wiesbaden, 2000.
- [DFG⁺98] P. Dahm, J. Fricke, R. Gimnich, M. Kamp, H. H. Stasch, E. Tewes, and A. Winter. Anwendungslandschaft der Volksfürsorge, Grobgranulares Konzeptschema und Anfragemöglichkeiten. In *[EGSW98]*, pages 91–120. 1998.
- [DW98] P. Dahm and F. Widmann. Das Graphenlabor. In *[EGSW98]*, pages 67–84. 1998.
- [Ebe85] Jürgen Ebert. Graph Implementation of a Functional Language. In H. Noltemeier, editor, *Proceedings of the Workshop on Graphtheoretic Concepts in Computer Science (WG '85)*, pages 73–84, Linz, 1985. Trauner Verlag.
- [Ebe87] J. Ebert. A Versatile Data Structure for Edge-oriented Graph Algorithms. *Communication of the ACM*, 30(6):513–519, June 1987.
- [EF95] J. Ebert and A. Franzke. A Declarative Approach to Graph Based Modeling. In *E. Mayr, G. Schmidt, G. Tinhofer (eds.): Graphtheoretic Concepts in Computer Science, Springer, LNCS 903*, pages 38–50. 1995.
- [EGSW98] J. Ebert, R. Gimnich, H. H. Stasch, and A. Winter, editors. *GUPRO — Generische Umgebung zum Programmverstehen*. Fölbach, Koblenz, 1998.
- [EKM01] J. Ebert, K. Kontogiannis, and J. Mylopoulos. Interoperability of Reverse Engineering Tools. Dagstuhl Seminar Reports, Seminar 01041, 2001.
- [EKRW02] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO – Generic Understanding of Programs, An Overview. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.
- [EKW99] J. Ebert, B. Kullbach, and A. Winter. GraX – An Interchange Format for Reengineering Tools. In *6th Working Conference on Reverse Engineering, Proceedings, IEEE Computer Society*, pages 89–98. 1999.

- [ESU97] J. Ebert, R. Süttenbach, and I. Uhe. Meta-CASE in Practice: a Case for KOGGE. In A. Olivé, J. A. Pastor (eds.): *Advanced Information Systems Engineering, Springer, LNCS 1250*, pages 203–216. 1997.
- [EWD⁺96] J. Ebert, A. Winter, P. Dahm, A. Franzke, and R. Süttenbach. Graph Based Modeling and Implementation with EER/GRAL. In B. Thalheim, editor, *Conceptual Modeling — ER'96*, LNCS 1157, pages 163–178. Springer, 1996.
- [FBTG02] R. Ferenc, À. Beszèdes, M. Tarkiainen, and T. Gyimòthy. Columbus - Reverse Engineering Tool and Schema for C++. In *18th International Conference on Software Maintenance, Proceedings*, pages 172–181. IEEE Computer Society, 2002.
- [Fli06] S. Flick. Das Dagstuhl Middle Metamodel im Kontext sprachunabhängigen Refactorings, Diplomarbeit, Universität Koblenz-Landau, Institut für Softwaretechnik, 2006.
- [Gra] Graphviz - Graph Visualization Software. <http://www.graphviz.org>.
- [Hol98] R. C. Holt. Structural Manipulations of Software Architecture using Tarski Relational Algebra. In *5th Working Conference on Reverse Engineering, Proceedings, IEEE Computer Society*, pages 210–219. 1998.
- [HSEW06] R. C. Holt, A. Schürr, S. Elliott Sim, and A. Winter. GXL: A Graph-Based Standard Exchange Format for Reengineering. *Science of Computer Programming*, 60(2):149–170, April 2006.
- [HVd06] E. Hajiyev, M. Verbaere, and O. de Moor. CodeQuest: Scalable Source Code Queries with Datalog. In Dave Thomas, editor, *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, LNCS 4067, pages 2–27, Berlin, Germany, 2006. Springer.
- [JB06] M. Schulze J. Bach. Migration des Debeka-Software-Repositorys auf ein RDBMS. *Softwaretechnik-Trends*, 26(2), Mai 2006.
- [JGr] JGraLab: The Java Graph Laboratory. <http://jgralab.uni-koblenz.de>.
- [Kah06] S. Kahle. JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen. Diplomarbeit, Universität Koblenz-Landau, Institut für Softwaretechnik, 2006.
- [Kam98] 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, editors, *6th International Workshop on Program Comprehension, Proceedings*, pages 64–71. IEEE Computer Society, 1998.
- [KBC04] A. Kalnins, J. Barzdins, and E. Celms. Model Transformation Language MOLA. *Lecture Notes in Computer Science*, 3599:14–28, 2004.
- [KC99] R. Kazman and J. Carrière. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Automated Software Engineering*, 6(2):107–138, April 1999.
- [Kie97] O. Kienitz. Software Metriken. Studienarbeit S 475, Universität Koblenz-Landau, Institut für Softwaretechnik, Januar 1997.
- [Kla07] I. Klassen. GReQL-Script. Studienarbeit, Universität Koblenz-Landau, Institut für Softwaretechnik, 2007.
- [KR01] B. Kullbach and V. Riediger. Folding: An Approach to Enable Program Understanding of Preprocessed Languages. In *8th Working Conference on Reverse Engineering, Proceedings*, pages 3–12. IEEE Computer Society, October 2001.
- [KS01] B. Kullbach and G. Schmitz. Dokumentation des Ada-Parsers für GUPRO. Projektbericht 9/01, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 2001.
- [KW99] B. Kullbach and A. Winter. Querying as an Enabling Technology in Software Reengineering. In P. Nesi and C. Verhoef, editors, *Proceedings of the 3rd European Conference on Software Maintenance and Reengineering*, pages 42–50. IEEE Computer Society, 1999.
- [KW00] B. Kullbach and A. Winter. Visualisierung von Macros durch Folding. In J. Ebert, B. Kullbach, and F. Lehner, editors, *2. Workshop Software Reengineering (Bad Honnef, 11./12. Mai 2000), Fachbericht Informatik 8/2000, Universität Koblenz-Landau*. 2000.

- [KWDE98] B. Kullbach, A. Winter, P. Dahm, and J. Ebert. Program Comprehension in Multi-Language Systems. In *5th Working Conference on Reverse Engineering, Proceedings, IEEE Computer Society*. Los Alamitos, 1998.
- [LSW01] C. Lange, H. Sneed, and A. Winter. Comparing Graph-based Program Comprehension Tools to Relational Database-based Tools. In *9th International Workshop on Program comprehension, Proceedings*, pages 209–218. IEEE Computer Society, 2001.
- [OMG06] OMG. Meta Object Facility Core Specification, Version 2.0. Technical report, 2006.
- [OMG07] OMG. MOF 2.0 / XMI Mapping Specification, V2.1.1. <http://www.omg.org/technology/documents/formal/xmi.htm>, 2007.
- [OO84] K. J. Ottenstein and L. M. Ottenstein. The Program Dependence Graph in a Software Development Environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM.
- [OvFK98] 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.
- [Rap] RapidMiner (YALE): Open-Source data mining with the Java software RapidMiner (YALE). <http://rapid-i.com/content/blogcategory/10/69/>.
- [ReD] ReDSeeDS: Requirements Driven Software Development System. <http://www.redseeds.eu>.
- [Rie01] V. Riediger. The GUPRO C Parser. Projektbericht 5/01, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 2001.
- [Rie04] V. Riediger. *Die Präprozessor-Problematik im Reverse Engineering und beim Programmverstehen*. PhD thesis, Universität Koblenz-Landau, November 2004.
- [RVP06] A. Raza, G. Vogel, and E. Plödereder. Bauhaus, A Tool Suite for Program Analysis and Reverse Engineering. In *L. M. Pinho and M. G. Harbour: Reliable Software Technologies, Ada-Europe 2006, 11th Ada-Europe International Conference on Reliable Software Technologies, Proceedings*, pages 71–82. 2006.
- [Sch07] H. Schwarz. *Program Slicing - Ein dienstorientiertes Modell*. Verlag Dr. Müller, 2007.
- [SD99] H. M. Sneed and T. Dombovari. Comprehending a Complex, Distributed, Object-oriented Software System, A Report from the Field. In *7th International Workshop on Program Comprehension, Proceedings*, pages 218–225. IEEE Comp. Society, 1999.
- [SM95] M.-A. Storey and H. A. Müller. Manipulation and Documenting Software Structures Using SHriMP Views. In *International Conference on Software Maintenance, Proceedings, IEEE Computer Society*, pages 275–285. 1995.
- [Smi07] M. Smialek. *Software Development With Reusable Requirements-Based Cases*. Oficyna Wydawnicza Politechniki Warszawskiej, Warsaw, 2007.
- [SRW06] F. Schricker, V. Riediger, and A. Winter. GXL2SVG: Domain Specific Graph Layout. *Softwaretechnik-Trends*, 26(2):63–64, 5 2006.
- [Til95] S. R. Tilley. Domain-Retargetable Reverse Engineering. Phd thesis, Department of Computer Science, University of Victoria, Victoria, January 1995.
- [WHW02] J. Wu, R. C. Holt, and A. Winter. Towards a Common Query Language for Reverse Engineering. Fachberichte Informatik 8/2002, Universität Koblenz-Landau, Institut für Informatik, Koblenz, 2002.
- [Wid01] F. Widmann. Entwicklung von Batch-Jobs bei der Debeka. Konzeptvorschlag für eine Datenstruktur des Repositories. Projektbericht 6/01, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 2001.
- [Win00] A. Winter. *Referenz-Metaschemata für visuelle Modellierungssprachen*. Deutscher Universitätsverlag, Wiesbaden, 2000.
- [WN05] U. Kelter J. Wehren and J. Niere. A Generic Difference Algorithm for UML Models. In *Proceedings of the SE 2005, Essen, Germany*, Essen, Germany, March 2005.
- [Won98] K. Wong. RIGI User’s Manual, Version 5.4.4. <http://www.rigi.csc.uvic.ca/rigi/rigiframe1.shtml?Download>, 30. June 1998.
- [WW05] J. Wolff and A. Winter. Blickwinkelgesteuerte Transformation von Bauhaus-Graphen nach UML. *Softwaretechnik-Trends*, 25(2):33–34, Mai 2005.