Comparing Graph-based Program Comprehension Tools to Relational Database-based Tools

Carola Lange[1]
Institute for Software Technology
University of Koblenz-Landau
Rheinau 1, D-56075 Koblenz
Germany
clange@uni-koblenz.de

Harry M. Sneed
Case Consult GmbH
Flachstraße 13
D-65197 Wiesbaden
Germany
harry.sneed@caseconsult.com

Andreas Winter
Institute for Software Technology
University of Koblenz-Landau
Rheinau 1, D-56075 Koblenz
Germany
winter@uni-koblenz.de

Abstract

In this paper we compare the experiences of applying the graph-based GUPRO approach to experiences in applying ANAL/SoftSpec – an approach based on relational databases. We present the results of a case study in which GUPRO has been applied to a multi-language software system for stock trading (GEOS). Comparing the results of the case study with experiences of applying ANAL/SoftSpec to GEOS we show that the graph-oriented approach enables an efficient way of source code analysis and program understanding.

Keywords: comparison of program comprehension tools, relational database repository, graph-based repository, querying repositories.

1. Introduction

During the last few years lots of program comprehension tools have been developed in research as well as in industry. To show their usefulness, it is apparently necessary to apply these tools to various maintenance problems from the field and evaluate them in real use cases. Comparing the application of different tools to equal or similar tasks leads to a better consideration of the pros and cons of the underlying tool approaches. Therewith, knowledge about their usability in simplifying the task of understanding complex software systems can be gained. Further steps in adapting the tools to the demands of the maintenance programmers can be made, facilitating an ongoing improvement of both approaches and tools.

Bellay and Gall presented a comparison of four different Reverse Engineering Tools (Refine/C, Imagix 4D, Rigi, Sniff+) on WCRE 1997 [1]. The main goal was to investigate the differences between the tools with respect to the generation of graphical reports. Based on 80 assessment criteria Bellay and Gall conclude that it is difficult to analyze embedded systems with current reverse engineering tools and that in particular the graphical views and layouts need considerable improvement. They also state that mixed-language support is a necessity for real-world applications.

Elliot Chikofsky organized a Reverse Engineering Demonstration Project [6]. In a cooperative study among commercial and non-commercial research groups the results and value of their methods and tools in analyzing the WELTAB III Election System were demonstrated.

Susan Elliott Sim and Margaret-Anne Storey organized two consecutively structured tool demonstrations at CASCON 1999 and WCRE 2000 [20]. Several teams were given a set of architectural analysis and maintenance tasks to perform on a common subject system. This comparison provided the tool developers with insights into their own tools and gave them the opportunity to view other tools, their ability and performance to provide support for equal tasks. They also point out that tool evaluation is necessary to enhance the technology transfer and widen the acceptance of program comprehension tools in industry.

In contrast to Bellay and Gall's study, which emphasizes tool visualization, we focus on comparing different repository structures. Source code analysis can be performed either text-based or repository-based. Both types of approaches were tested and evaluated on the structured tool demonstrations. As an example, for textual based analysis, the simple editor facilities and basic UNIX-tools like *grep* and *find* were used. The experiences in applying Unix tools to analyze XFig are described in [24]. Software development teams, which use repository structures based on graphs or binary relations representing source code artifacts, also participated in the demonstration (PBS

[1] Currently on a scholarship program at the Artificial Intelligence Center of The University of Georgia (August 2000 – May 2001).

(Portable Bookshelf) [7] and Rigi [18]). The graphical code browsing and program comprehension tool TkSee [21], which uses repository structures based on relational databases was evaluated during the CASCON 1999 workshop. The Relation Partition Algebra (RPA) described by [13] is another repository-based approach using binary relations for data representation. Relational database management systems provide the underlying repository structure for further program comprehension tools. An approach based on relational databases was first introduced by Linton in [17]; yet, his results regarding response time performance were unsatisfactory using RDBMS available in the early 1980s. More recent RDBMS based approaches are described in [5] and [24]. Other repository structures are based on logic oriented data representations [3], [11], [4], LISP Images [19], syntax trees [26], or hybrid knowledge bases [10].

This paper succeeds [24], which describes the results of applying the CPPANAL source code analyzer to the sources of the stockbroker trading system GEOS. Their approach is based on relational databases. Here, we present the results of applying the graph-based GUPRO approach [2] to GEOS. Thus, this paper compares the application of a graph-based approach to program comprehension with a relational database approach. It also describes the ways in which two different tool sets can be combined into an interoperable reengineering workbench.

The subsequent parts are organized as follows: Section 2 gives a characterization of the reengineering project at the stock broker in Vienna and first insights into the maintenance tasks and inquiries of the maintenance programmers. The approach pursued in this case study and the tools used are described in section 3. Finally section 4 gives insights about the actual experiences that were made by applying the tools.

2. Reverse Engineering of *GEOS*

GEOS (Global Entity Offering System) is a stock trading application system developed in Vienna as a standard software package to be sold to banks. It has been under development for five years. There are currently four banks in Austria and two banks in Germany using the system.

GEOS consists of 8 subsystems plus a Java gateway to the internet. There are different subsystems for order processing, conditions management, depot administration, portfolio management, risk management, trading, clearing, and investment management. Each of these systems has a C++ frontend to serve the distributed graphical user interfaces and a C backend to process the centralized relational databases. The Java Internet gateway offers a bypass to the C backend components via a wrapper layer. Analyzing such multi-language systems naturally requires an integrated examination of all subsystems.

The system is constantly growing. Currently it has 6,279 source files and 2,364,652 lines of code [22]. Systems of this size and complexity are inherently difficult to document, because of the great number of entities and the even greater number of relationships. The nature of object-oriented software also leads to a greater number of interactions between separately compiled modules since methods in one class refer to methods in another.

When developing GEOS there was no CASE tool used to document the design. The code was produced based on the requirement specification in a semi-formal functional specification language, which does not provide any information about the technical architecture. Thus, the only descriptions of the programs are the sources themselves and the comments contained therein.

For this reason, it was decided to use reverse engineering tools to recapture design information from multi language sources and to store it in a software repository. The key technology used here is static analysis. Through static analysis, it is possible to generate entity and relationship tables from each source member, depicting variable references, database accesses, file usage, class inheritance and other relevant cross references. These intermediate cross reference tables for each module are then processed to populate the repository in a relational database [23].

There are currently more than 1600 components, 3000 modules, 2200 classes, 30,000 interfaces, 34,000 functions, 290,000 function calls, 192,341 data declarations and 895,110 data references. This gives an idea of the magnitude of the documentation problem. Maintainers want to have specific answers to specific questions [9]. Unfortunately the set of questions a maintenance programmer will ask, in order to comprehend software systems, cannot be foreseen. Thus, program comprehension tools have to provide a powerful query mechanism to investigate software systems [2]. Regarding the architectural level, these questions are classifiable into five main types: relations between modules and functions, include relationships, call relationships, inheritance relationships and metrics. Here are some examples of maintainers' specific questions:

- How does a certain module interact with other elements in the system and which functions belong to a particular module?
- Which modules include a certain header file?
- Which function calls a particular function and which function is called by a certain function?
- Which classes specialize a certain (super) class?

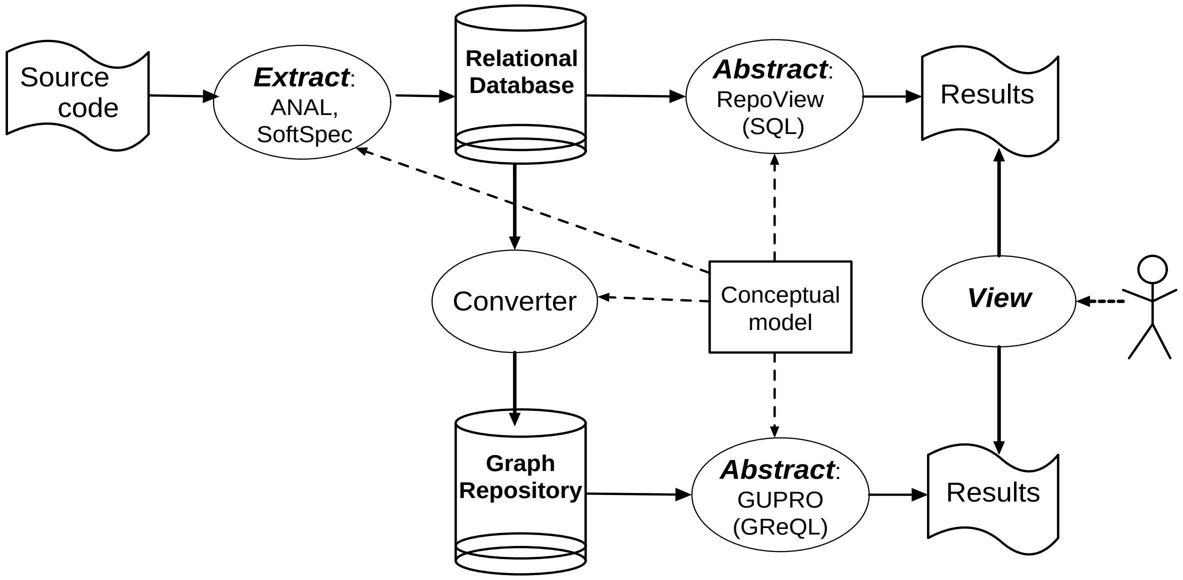


Figure 1: Approach

- How many other functions are called by a particular function, or what is the average of called functions of a particular module?

Scanning through complex diagrams, whether on paper or on a graphical user interface, is no efficient way to comprehend large software systems. Directly querying the data repository has been shown to be an enabling technology in this field [14]. Standard SQL queries have already been performed on the database repository [23]. The GUPRO (Generic Unit for Program Understanding, [2]) approach is now being used, while higher performance due to the use of graph-based query technology is expected. Furthermore it should be easier to formulate complex queries such as querying of transitive closures.

3. Approach and Tools

The two approaches compared in this paper follow the Extract-Abstract-View metaphor, where source code is extracted into an analyzable representation. Abstractions are calculated by suited analysis features and queries, which are visualized afterwards. The data that is extracted from code artifacts is defined in a conceptual model. This also controls the abstraction in the way that certain abstractions of the data can be queried according to the conceptual model.

The overall approach is depicted in figure 1. Tables were extracted from the GEOS source code with respect to a conceptual model (***extract***). The main tools responsible for the extract process are a parser component (ANAL) and a component that creates the relational database repository (SoftSpec). These tools exist for JAVA, C/C++, and IDL source code. The repository can be queried (***abstract***) with the SoftSpec component RepoView using standard SQL statements. It is possible to select relations from a single table via a SELECT..FROM.. WHERE statement or to make a join of several tables by means of a nested SELECT statement. The results of the query are displayed (***view***) in an Excel type table where they can be viewed or printed out.

By transforming the database repository into a graph, whose vertices and edges depend on the underlying ***conceptual model***, further abstractions can be performed. Transforming contents of relational databases into graphs is straightforward; the database is read out into a graph structure according to the conceptual model.

The GUPRO approach provides an adaptable and extensible workbench for program analysis. GUPRO is strongly based on graph technology. Source code is parsed into graph structures, which are accessible by graph algorithms and a general graph query language GReQL [14]. With GReQL graphs are queried (***abstract***) according to entities (nodes) and associations (edges) represented in the conceptual model. GReQL query results can be viewed in ASCII- or HTML-tables (***view***). GUPRO additionally provides source code browsers [25] with scaleable representations of source code through the use of folding techniques. Folding is a technique that can be used for structuring text documents. In GUPRO, folding is used to represent the replacements of the C preprocessor and other user defined replacements [15]. Of

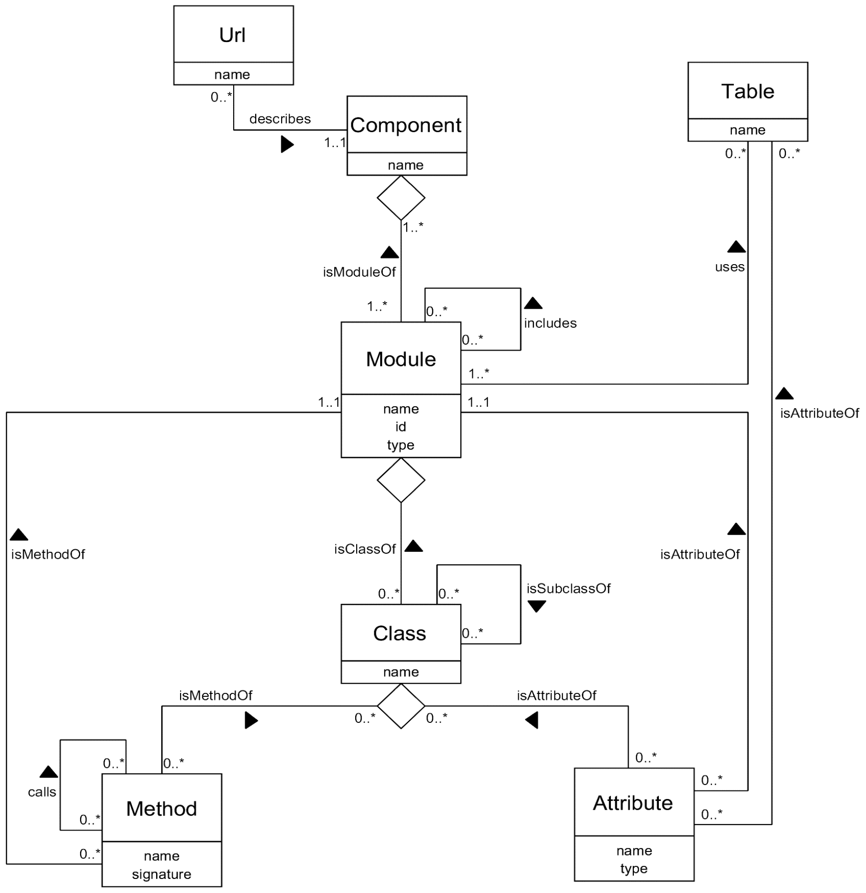


Figure 2: Conceptual Model

course, the representation of results in tables and – in terms of source code – in browsers is integrated.

In this paper we focus on the process of abstraction through querying relational databases or graph-based repositories. In the following section we sketch the underlying conceptual model, which provides combining ANAL/SoftSpec and GUPRO. Afterwards, the extract, the abstract, and the view components in both approaches are described in further detail.

3.1 Conceptual Model

The authors developed the underlying conceptual model through discussions in joint meetings. It describes the amount of data, which is exchanged between ANAL/SoftSpec and GUPRO. The conceptual model, depicted in the UML class diagram in figure 2, defines the common repository structure for analyzing GEOS on an architectural level. This conceptual model defines both, the relational data base structure used in ANAL/SoftSpec and the graph-based structure used in GUPRO. The conceptual model represents the co-operation of modules, classes, and methods in order to enable the analysis of associations between different entities.

The GEOS system is separated into various *components* that contain multiple *modules*. For external documentation components are associated to further documents, accessible by URLs. In the case of C++ and Java modules these are further decomposed into *classes*, which are collections of *methods* and *attributes*.

For non-object oriented languages, functional components (here also called *methods*) and *attributes* are directly associated with modules. The databases accessed by GEOS modules are represented by their *tables*, including the herein referenced *attributes.* The associations render an include hierarchy (*includes*) between modules, a class hierarchy (*isSubclassOf*) between classes, and the call relations (*calls*) between methods.

3.2 Extract: Generating the GEOS repository

In order to extract information from GEOS source code, first, a series of extraction tools (ANAL) is applied, which parse the code, extract the relevant information, and create an intermediate program description. In a second step, further tools (SoftSpec) process this intermediate description to create the actual relational databases. The resulting repository database consists of 18 binary relationship tables plus a document text table containing module structures and comments, for the several language systems (C, C++, JAVA, RDBMS).

In this case study we have combined two separate tool approaches. Therefore, some kind of transformer was necessary to map the different data structures. For this reason the relational database was converted to the standard exchange format GXL [8], which can be used as input for the GUPRO tools. In this way, the benefits of already existing source code parsers could be combined with the benefits of GUPRO analyzing and program understanding techniques.

GEOS consists of 8 subsystems. The examples in this paper are taken from the investment banking service subsystem Nostro that is evenly divided between C++ and C. The resulting database consists of 18 tables with combined 28,890 rows. The graph, representing the tables of the subsystem Nostro, has a size of 795 KB, with 8223 nodes and 20,815 edges.

3.3 Abstract

Once the database repository has been loaded by the SoftSpec tool, there are two possible ways to use it: the adhoc query of cross references, and the generation of standard system documents. These documents depict the different hierarchies and networks from the repository, e.g. function call trees, class inheritance trees, and function sequence diagrams.

Querying the GEOS repository is accomplished by RepoView using any standard SQL SELECT statement supported by the IBM Universal DB2 database system. For example, the database repository contains a table Func_Plus, which relates to each function (Func_Name) all called functions (Target_Name). A typical SQL query (CheckErrOut.sql), which lists all functions that invoke the function 'CheckErrOut', is the following:

```
SELECT Func_Name
FROM Func_Plus
WHERE Target_Name =“CheckErrOut“
```

SQL also provides further queries, which join several tables. A detailed description of experiences in querying the GEOS repository with SQL can be found in [23].

GUPRO is based on graph technology, and adaptability is given by graph-based conceptual modeling. In contrast to object-oriented modeling, where associations are treated as references from one association end to the other, and which can only be navigated in one direction, in graph-based modeling, edges are considered as first-class citizens. Thus, graph-based modeling provides navigation of associations in both directions – in and against the orientation of an edge type. Embedding these features into graph query-languages allows powerful navigation through graph structures. Here the graph representation of the tables was queried with the query language GReQL, a language especially suited to graph querying. The FWR (FROM WITH REPORT) expression is the most important language element. Within the `FROM` clause variables are declared; in the `WITH` clause the set of possible variables can be restricted by first order logic predicates. They can also contain regular path expressions including (reflexive) transitive closures. The REPORT clause specifies the representation of query results [12] [14].

The following query (CheckErrOut.grq) reports – just as the above mentioned SQL query CheckErrOut.sql – the names of all functions, which are directly calling the function with name 'CheckErrOut'. The types of nodes and edges used in the GReQL query refer directly to those depicted in figure 2.

```
FROM
callee:V{Method},caller:V{Method}
WITH callee.name = 'CheckErrOut'
     AND callee <--{calls} caller
REPORT caller.name
END
```

The FROM part declares two variables, `caller` and `callee`, of node type `Method`. The WITH part restricts the possible assignments to `callee` to those methods with a name attribute equal to 'CheckErrOut'. Furthermore, the possible assignments to `caller` are restricted to those linked to `callee` by an outgoing edge of type 'calls' in the second part of the WITH clause. The REPORT part specifies to report the value for the attribute name of each method which fulfills the WITH clause.

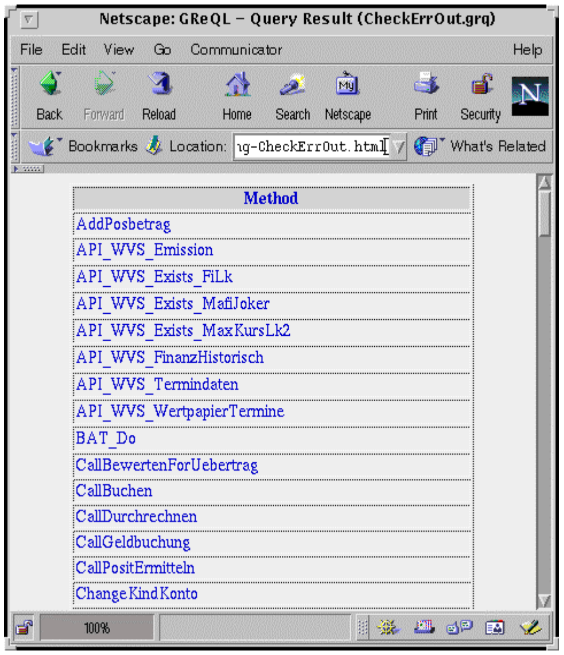
GreQL is an expression language; predicates can be formulated using first order logic and may also contain path expressions to describe regular path structures, such as sequences, alternatives and iterations, also including reflexive and transitive closures. Queries are efficiently evaluated by an automaton driven calculation of the path expressions. The challenge of applying GReQL to a real-world reengineering case was to evaluate how these means can be used in order to improve program understanding. A further reason for applying GReQL to this case was to see whether the graph-based approach could keep up with the original approach of reengineering GEOS by using relational databases to represent entities and relations of the system.

3.4 View

The original viewing aspects of the SoftSpec tools were limited to the table rows as they are stored. The user gets the SQL column name at the top of each column and, thereunder, a list of all selected values in that column ordered by primary key. If a value is missing in a row, the column entry for that row is empty.

In this case study, GReQL queries were evaluated with the command line GReQL tool CLG. CLG supports three different output formats of the query results: ASCII tables, HTML tables and comma-separated format. A more user-friendly graphical user interface in C++ for querying and browsing graphs as well as supporting loading, editing and saving of queries and their results is currently under construction [25].

Figure 3 depicts the HTML result of the formerly mentioned GReQL query CheckErrOut.grq, which lists the names of all methods that are directly calling CheckErrOut.



Method
AddPosbetrag
API_WVS_Emission
API_WVS_Exists_FiLk
API_WVS_Exists_MafiJoker
API_WVS_Exists_MaxKursLk2
API_WVS_FinanzHistorisch
API_WVS_Termindaten
API_WVS_WertpapierTermine
BAT_Do
CallBewertenForUebertrag
CallBuchen
CallDurchrechnen
CallGeldbuchung
CallPositErmitteln
ChangeKindKonto

Figure 3: GReQL query result

4. Experiences in applying GReQL

In this section we describe the actual case study – the kinds of queries that were performed in order to answer the maintenance programmer's questions. In addition, new insights into the applicability of GReQL with respect to time performance and mightiness of the language constructs are presented.

As introduced in section 2, the queries performed on the Nostro graph are classifiable into five main types: relations between modules and functions, include relationships, call relationships, inheritance relationships, and metrics. See [16] for a detailed description of all performed queries and their results. Each type of query, along with a typical example, is described in subsections 4.1 to 4.5. Section 4.6 gives an overview of the time performance of GReQL in comparison to similar SQL queries. Finally, the applicability of the query technique in real-world use cases to gain a deeper understanding of large software systems is discussed in section 4.7.

4.1 Module – Function relationships

The maintenance programmer first has to gain an overview of the modules and their methods in order to have an overview of the whole system. Queries such as the following can be classified as queries concerning module-function relationships:

- Which functions belong to a particular module?
- Which modules contain functions being called by a particular function?
- Which modules use a particular function?

For example, the Nostro system contains a central module which can be recognized by its attribute id = 0. In order to discover which other elements of the system are related to this specific module, you simply traverse all incoming and outgoing edges of this module node and list the corresponding nodes. The following GReQL-query (listNeighbors.grq) reports the name of every node that has a direct link to the module with id = 0:

```
FROM m:V{Module}
WITH m.id = 0
REPORT
 FROM v:V{}
 WITH v <->{} m
 REPORT v.name
 END
END
```

The first FROM-WITH part declares the variable `m` of type Module and restricts its assignments to those module nodes with attribute id = 0. The nested FROM-WITH-REPORT clause specifies to report the names of all nodes that have a direct relation to `m`. The query result of listNeighbors.grq shows that the module with id = 0 is directly associated to 1409 other elements of the system, i.e. to 18.3 % of all system elements.

4.2 Include relationships

The Nostro system contains about 600 modules. Hence, keeping track of the include relationships between these modules is an important aspect of understanding how the individual parts of the software system work together. Typical queries about include relationships are the following:

- Which modules are included by a particular module?

- Which modules include a particular module?
- Which modules are included by the header files of a particular module?

For example, the attributes id, name and type of each module that includes the header file 'nndnostr' are reported by the following query (includedByMod.grq):

```
FROM inc:V{Module}
WITH inc.name = 'nndnostr'
REPORT
   FROM m:V{Module}
   WITH inc <--{includes} m
   REPORT m.id, m.name, m.type
   END
END
```

The outer FWR clause declares a variable `inc` and restricts its assignments to module nodes with name 'nndnostr'. The inner FWR clause declares a second variable of node type module and reports the attribute values of those modules that include `inc`. The result of includedByMod.grq lists 43 different modules that include module 'nndnostr'.

4.3 Call relationships

More than 3000 functions belong to the approximately 600 modules of the Nostro system. Queries about which function is invoked by a particular function, and which function calls a particular function help understanding which parts of the system will be influenced by or depend on changes in the functionality of certain procedures. As an example, the following GReQL query lists for each function all functions that are invoked by the function (directlyCalledFuncs.grq):

```
FROM caller:V{Method}
REPORT caller.name,
   FROM callee:V{Method}
   WITH caller -->{calls} callee
   REPORT callee.name
   END
END
```

The result of directlyCalledFuncs.grq lists, in about 7075 lines, the name of each function (`caller`) together with the set of functions (`callee`) which are directly called by `caller`.

GReQL also enables the user to query transitive closures. The following query (callFunctions.grq) applies the '*' operator to list for each function all functions that are called directly or called indirectly by a called function.

```
FROM caller:V{Method}
REPORT caller.name,
   FROM callee:V{Method}
   WITH caller -->{calls}* callee
   REPORT callee.name
   END
END
```

The result of callFunctions.grq contains about 150,000 rows, listing for each function `caller` every function whose change could have an impact on `caller`. Since edges can be navigated in both directions the direction of the arrow can be changed in order to enumerate for each function *f* all functions whose change might have an impact on *f*.

The ability of GReQL to query transitive closures is one of its significant advantages in comparison to SQL. Even though newer SQL standards support the querying of transitive closures, having to process a non-predictable amount of joint operations limits the possible time performance. The processing of transitive closures in GReQL can be performed efficiently by directly traversing the graph structure.

4.4 Inheritance relationships

In understanding object-oriented programs, new difficulties arise due to the possibility of applying (multiple) inheritance and polymorphism. Queries about generalization or specialization of a particular class in the system are classified as queries about inheritance relationships. Typical queries are as follows:

- Which classes are specializations of a particular class?
- Which classes are super classes of a particular class on a distinct level of generalization?
- Which classes are affected from multiple inheritance?

The following GReQL query (highestSuperclass.grq) reports the 'highest' super class of each class, i.e. it reports the name of each class and the name of the most general super class of that class.

```
FROM c,super : V{Class}
WITH c -->{isSubclassOf}* super
AND
outDegree{isSubclassOf}(super) = 0
REPORT c.name, super.name
END
```

The FROM part declares two variables, `c` and `super,` of type class. The WITH part restricts the possible assignments such that `c` has to be a subclass of `super` on arbitrary level and that `super` must not be a subclass of any other class. The REPORT part specifies to report the name of class `c` and the name of its highest super class. The result of highestSuperclass.grq lists more than 2000 class name pairs.

4.5 Metrics

In order to gain insights into the size of the system and the proportion of the different types, various kinds of metrics were calculated. They can be classified into measurements counting specific types of nodes or edges, queries calculating the number of edges adjacent to the different types of nodes and metrics calculating averages. The following are queries typically classified as metrics:

- What is the number of components, modules, classes, methods or attributes?
- How many function calls are implemented in the system, a particular module, class or method?
- What is the average number of functions called by each function?
- What is the average number of classes that belong to a module?

These are only a small selection of possible metrics; a larger variety of software metrics is described e.g. in [27]. To give a metric example in GReQL, the following query (avgCalledFunc.grq) returns the average number of functions called by each function.

```
avg(
FROM m:V{Method}
REPORT outDegree{calls}(m)
END)
```

For the Nostro subsystem this query reports that, on average, 2.34 functions are called by each function.

4.6 Performance

The time performance of GReQL queries was measured and compared to the performance of similar SQL queries running on the corresponding relational databases. In order to analyze the time performance for each task of the command line GReQL tool CLG, four different times were measured: the duration to load the graph, the time to evaluate the GReQL query and the times to format and to print the output file. Loading the graph file (795 KB) with the CLG tool on a Pentium III (600 MHz, 256 MB RAM) takes 3.02 seconds. For standard queries it takes about 4 seconds to process the whole query, i.e. to load the graph, evaluate the query and print the output file. Hence, most of the execution time is "wasted" by reloading the graph for each query.

Table 1 lists the time needed to evaluate the query and the total time needed to perform the graph loading, query evaluation and output formatting for a series of typical GReQL queries. The respective times for SQL queries are also listed, measured in 0.5 seconds and performed on an AMD K6 PC with 300 MHz and 128 MB RAM.

For example, retrieving all methods that are directly calling function 'CheckErrOut' by a GReQL-query takes in total 3.46 seconds. Running the corresponding SQL query on the Nostro tables takes about 4 seconds. Comparing the total times of each GReQL and SQL query it can be seen that the query execution times are only slightly different. GReQL has an advantage in being able to traverse outgoing edges of any type, e.g. for listNeighbors.grq, whereas several tables have to be looked up and joined to retrieve this information from the database repository.

Table 1: Time performance comparison

Query	GReQL total time (sec)	GReQL evaluation time (sec)	SQL total time (sec)
listNeighbours	3.510	0.110	4
directlyCalledFuncs	4.830	0.880	3.5
callFunctions	21.200	10.550	–
cntComponents	3.300	0.0	4.5
cntModules	3.300	0.0	3.5
cntClasses	3.290	0.0	3.5
cntFunctions	3.300	0.060	4
cntAttributes	3.360	0.060	3.5
CheckErrOut	3.460	0.110	4

In order to perform the SQL query directlyCalledFuncs, only the Func_Plus table has to be searched. But in GUPRO the whole graph has to be loaded. If multiple queries used the same already loaded graph, 3 seconds could be saved for the execution of each GReQL query (assuming the current Nostro graph is used). Then the much shorter GReQL evaluation times could be compared to the performance of SQL on the corresponding relational databases. This would result in a significant difference in the time performance, which can easily be seen from the particular entries in table 1.

The GReQL query callFunctions.grq as introduced in section 4.3 calculates the transitive closure of called functions for each function of the Nostro system. Since, the SQL standard, used in RepoView, does not support querying transitive closures no corresponding SQL query could be performed on the database repository.

We are aware of the fact that these measurements and their comparison can only be tentative, because the two query languages were evaluated on different processor types. But, the results indicate that GReQL provides the maintenance programmer with a comparable and – due to the ability of querying transitive closures – with even more advanced query functionality than SQL. Furthermore, GReQL query evaluation times keep up with the standardized repository query mechanisms of SQL, and further improvements are expected, provided that multiple queries use the same already loaded graph.

4.7 Applicability

We have shown that querying large software systems represented as graphs provides the maintenance programmer with various efficient opportunities for information retrieval. In this last section we briefly discuss the general necessity and usefulness of querying to the maintenance of large multi-language software systems like GEOS.

The primary contribution of the query facility in connection with the GEOS software repository is to support the impact analysis of change requests and error corrections [28]. To trace the source of an error it is necessary to navigate through the entities comprising a software system and to identify side effects before the correction is carried through. For instance, if an error occurs in a derived class, then it is prudent to examine the base classes of that class before coming to any conclusion.

The greatest source of errors in GEOS now are second level defects, a side effect of correcting other errors, so any effort to avoid such undesired side effects is well worth it. Before implementing a change request it is now imperative to first come up with a cost estimate. The only way to achieve this is by assessing the impact domain of the change, i.e. all classes, interfaces, methods and attributes, affected. This can best be done by querying the repository and not by scanning through various documents. Maintenance programmer questions like "What functions are called by the function to be changed and what functions call it?" can be answered by a directed query in seconds. Scanning through graphical documents in a CASE Tool takes hours; scanning through paper documents may take days. Therefore, utilizing querying technology is particularly efficient for answering questions regarding side effects of changing and further development of legacy software. Maintenance programmers using GUPRO in the context of former case studies stated clearly, that diagrams do not support their every day work; but query results in the form of source code and tables are much more suited to support their work and give answers to their daily questions [2].

Once affected elements have been identified it is possible to estimate the rate of change in percentage of the whole. This percentage is then taken from the Function-Points or Object-Points of the total impact domain to come up with a Function-Point or Object-point count of the change. This count can then be converted into man-days via the maintenance productivity tables. The use of impact analysis to estimate maintenance costs, as well as to identify intersecting change requests, has been covered in previous papers [29]. By means of a query language it is possible to improve the accuracy of the impact analysis while at the same time reducing the time required.

5. Conclusion

In this paper, we have presented the results of a case study, in which the graph-based GUPRO approach and ANAL/SoftSpec, an approach based on relational databases, were combined. In this way, the benefits of already existing source code parsers could be combined with the benefits of GUPRO analyzing and program understanding techniques in order to reengineer GEOS, a large software system.

It was shown that the graph query language GReQL provides as much functionality as the standard relational database query language SQL. In addition, GReQL enables the maintenance programmer to formulate regular path expressions in order to declaratively describe cross references. GReQL path expressions also support first order logic and transitive closures, which can be queried with respect to the underlying conceptual model.

GReQL queries are efficiently evaluated by an automaton driven calculation of the path expressions, but it was shown that most of the time is needed to load the associated graph in standard queries. Hence, it still remains to be evaluated to what extent the time performance of GReQL can be improved by performing multiple queries while loading the graph only once. Since, in this case study, the time measurements to compare the performance of GReQL and SQL were made on different types of processors, further studies have to be performed, in order to find out more exactly which approach is the most efficient.

One benefit of GReQL is the ability to directly traverse the graph, whereas in SQL different tables might have to be joined while executing the query. This contributes to the fact that queries which use only one table are generally more efficient to execute as an SQL query on the database repository, and queries related to many different types of system elements generally have shorter execution times in GReQL.

6. Acknowledgements

We would like to thank the GUPRO team for their assistance in performing this case study. And we want to thank Jürgen Ebert, Bernt Kullbach and Keith Leatham, in particular, for their valuable comments, which helped improving this paper.

References

[1] B. Bellay, H. Gall: **A comparison of four Revese Engineerin Tools**, presented at the 4th Working Conference on Reverse Engineering (WCRE '97), Amsterdam, The Netherlands, 1997.

[2] J. Ebert, R. Gimnich, H. H. Stasch, A. Winter, **GUPRO - Generische Umgebung zum Programmverstehen**, Fölbach, 1998, Koblenz.

[3] G. Canfora, A. Cimitile, U. de Carlini, **A Logic-Based Approach to Reverse Engineering Tools Production**, IEEE Transactions on Software Engineering, 18(12): 1053-1063, December 1992.

[4] G. Canfora, L. Mancini, M. Tortorella, **A Workbench for Program Comprehension during Software Maintenance**, in Proceedings of the 4th Workshop on Program Comprehension, March 29-31 1996, Berlin, Germany, pp. 30-39, IEEE Computer Society Press, Los Alamitos, 1996.

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

[6] **Reverse Engineering Demonstration Project**, 1998, http://www.pathbridge.net/reproject/ (10/11/2000), presented at the 6th Reengineering Forum, Florence, Italy, March 1998.

[7] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong., **The Software Bookshelf**, IBM Systems Journal, Vol. 36, No. 4, pp. 564-593, November 1997.

[8] R. C. Holt, A. Winter, A. Schürr: **GXL – Toward a Standard Exchange Format**, Proceedings WCRE 2000, IEEE Computer Society Press, Los Alamitos, pp. 162-171, 2000.

[9] P. Hsia, A. Gupta., C. Kung, J. Peng, S. Liu, **A Study on the effect of architecture on the maintainability of object-oriented systems** in Proceedings of IEEE-ICSM-95, Opio, France, Computer Society Press, Oct. 1995, p. 4.

[10] S. Jarzabek, PQL: **A language for specifying abstract program views**, in Proceedings of the 5th European Software Engineering Conference (ESEC '95), Springer, Berlin, pp. 324-342, September 1995.

[11] S. Jarzabek, T. P. Keam, **Design of Generic Reverse Engineering Assistant Tools**, in Proceedings of the Second Working Conference on Reverse Engineering (WCRE '95), IEEE Computer Society Press, pp. 61-70, July 1995.

[12] M. Kamp, GReQL: **Eine Anfragesprache fuer das GUPRO-Repository - Sprachbschreibung (Version 1.2)**, Fachbericht Informatik 14/98, University of Koblenz-Landau, Fachbereich Informatik, 1998.

[13] R. Krikhaar, **Software Architecture Reconstruction**, PhD thesis, University of Amsterdam, 1999.

[14] B. Kullbach, A. Winter, **Querying as an Enabling Technology in Software Reengineering**, in P. Nesi, C. Verhoef, Proceedings of the 3nd European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 1999, Los Alamitos, pp 42-50.

[15] B. Kullbach, A. Winter, **Visualisieren von Macros durch Folding**, in: J. Ebert, B. Kullbach, F. Lehner (Eds.) Fachberichte Informatik, Universität Koblenz-Landau, Germany, 2000.

[16] C. Lange: **GReQL applied to reengineer GEOS**, Technical Report, to appear, Institute for Software Technology, University of Koblenz-Landau.

[17] M. A. Linton, **Implementing Relational Views of Programs**. In Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium n Practical Software Development Environments, pp. 132 - 140, May 1984.

[18] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl**., A reverse engineering approach to subsystem structure identification**, Journal of Software Maintenance: Research and Practice, 5(4), pp. 181-204, December 1993.

[19] P. Newcomb, **Legacy System Cataloging Facility**, in Proceedings of the Second Working Conference on Reverse Engineering (WCRE '95), IEEE Computer Society Press, pp. 52-60, July 1995.

[20] S. E. Sim, M. A. Storey: **A Structured Demonstration of Program Comprehension Tools**, Proceedings WCRE 2000, IEEE Computer Society Press, Los Alamitos, pp. 184-193, 2000.

[21] J. Singer, T. Lethbridge, N. Vinson: **An Examination of Software Engineering Work Practices**, presented at CASCON '97, Toronto, Canada, 1997.

[22] H. Sneed: **Measurement and Assessment of complex, distributed, object-oriented Application systems** in Proceedings of ESCOM-SCOPE Workshop, ESCOM Working Group, Hercmonceaux, England, April, 1999, pp. 101–109.

[23] H. Sneed, T. Dombovari: **Comprehending a complex, distributed, object-oriented software System - a Report from the Field**, in Proceedings of IEEE- IWPC-99, IEEE Computer Society Press, Pittsburgh, May, 1999, pp. 218–225.

[24] A. Tateishi, A. E. Walenstein: **Unix Tools for the XFig Structured Demonstration**, Proceedings WCRE 2000, Panel on structured Tool Demo, pp. 203–206, 2000.

[25] V. Hong: **Funktionsprototypischer Entwurf und Evaluation eines windowsbasierten GUPRO-Frontends**, Master Thesis to appear, Institute for Software Technology, University of Koblenz-Landau..

[26] C. H. Wells, R. Brand. L. Markosian, **Customized Tools for Software Quality Assurance and Reengineering**, in Proceedings of the Second Working Conference on Reverse Engineering (WCRE '95), IEEE Computer Society Press, pp. 71-77, July 1995.

[27] H. Zuse: **Software complexity: measures and methods**, Berlin, New York, W. de Gruyter, 1991.

[28] M. Fyson, C. Boldyreff: **Using Application Understanding to support Impact Analysis**, Journal of Software Maintenance, Vol. 10, No. 2, March 1998, pp. 93-110.

[29] H. Sneed: **Estimating the Costs of Software Maintenance Tasks**, in Proceedings of International Conference on Software Maintenance, IEEE Press, Opio, France, Oct. 1995, pp.168-181.