Querying as an Enabling Technology in Software Reengineering

Bernt Kullbach Andreas Winter
University of Koblenz-Landau
Institute for Software Technology
Rheinau 1, D-56075 Koblenz, Germany
(kullbach|winter)@informatik.uni-koblenz.de

Conference on Software Maintenance and Reengineering (CSMR ’99)

Querying as an Enabling Technology in Software Reengineering*

Bernt Kullbach Andreas Winter
University of Koblenz-Landau
Institute for Software Technology
Rheinau 1, D-56075 Koblenz, Germany
(kullbach|winter)@informatik.uni-koblenz.de

Abstract

In this paper it is argued that different kinds of reengineering technologies can be based on querying. Several reengineering technologies are presented as being integrated into a technically oriented reengineering taxonomy. The usefulness of querying is pointed out with respect to these reengineering technologies.

To impose querying as a base technology in reengineering examples are given with respect to the EER/GRAL approach to conceptual modeling and implementation. This approach is presented together with GReQL as its query part. The different reengineering technologies are finally reviewed in the context of the GReQL query facility.

1 Introduction

Reengineering may be viewed as any activity that either improves the understanding of a software or else improves the software itself [2].

According to this view software reengineering can be "partitioned" into two kinds of activities. The first kind of activities is concerned with understanding such as source code retrieval, browsing, or measuring. The second kind of activities aims at evolutionary aspects like redocumentation, restructuring and remodularization. We will refer to the former kind of activities as **understanding** and to the latter as **renovation** in the following. Understanding and renovation refer to both, whole software systems and single programs or source code fragments.

Both of the two classes of reengineering activities may be further subdivided into several types of reengineering techniques as shown in figure 1. Understanding covers base technologies like browsing, measurement, and cross referencing, as well as advanced technologies like slicing, object recovery or design recovery. Accordingly, renovation technology can be subdivided into remodularization, restructuring, redocumentation, data reengineering and so on. This subdivision of reengineering technology is not necessarily disjoint. Especially understanding techniques provide the basis for renovation tasks.

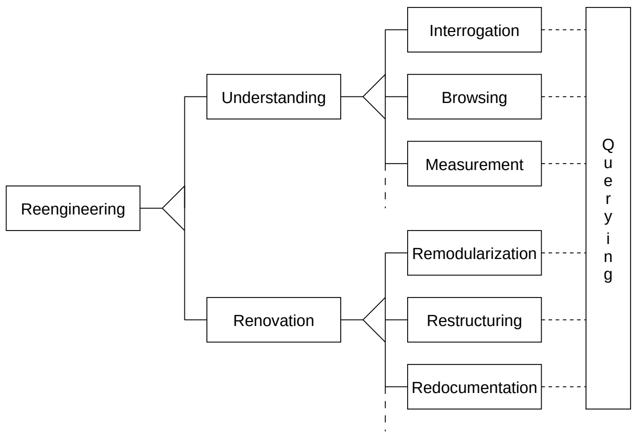

Figure 1. A reengineering taxonomy

In the following it will be argued that understanding as well as renovation technology to a large extent can be based on querying source code representations. This does not cause any conflict with the fact that querying is conventionally used in interactive program understanding. We will refer to this interactive program understanding technique as *interrogation* whereas the terms *query* or *querying* are used to denote the referring base technology. The need for interrogation tools has e. g. been reported by Biggerstaff in terms of a "conceptual grep" [3]. Prakash and Paul also noticed that there is a need for an interactive query facility [37]. Querying not only lets a user interactively retrieve a source

*This work has partially been performed within the GUPRO (Generic Understanding of PROgrams) project which has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie, national initiative on software technology, No. 01 IS 504. Information on GUPRO including the technical reports cited in this paper is available from `http://www.uni-koblenz.de/~ist/gupro.html`

code representation. It can also be used within most of the above-mentioned reengineering tasks. This is witnessed by a lot of work that has been performed in the software reengineering domain.

In order to convey our message this paper is organized as follows. The next section identifies uses of queries in understanding and renovation. Our query tool approach is presented in section 3 within a common framework. The use of this query approach in reengineering is outlined in section 4. Here the use of querying in program understanding and software renovation is described. The paper ends with a conclusion.

2 Querying and reengineering technology

Querying constitutes a base technology which can be efficiently used in most reengineering applications.

As said before interactive program understanding is the conventional application domain for query tools. Many **interrogation** approaches have been proposed while being based on different conceptual modeling techniques, data structures and analysis mechanisms.

Paul and Prakash have proposed a Source Code Algebra (SCA) a as basis for querying abstract syntax tree like program representations [37]. The SCA query evaluator is embedded in the ESCAPE prototype query system which is based on an object-oriented database source code repository. An object-oriented database is also used as part of the Refine toolset [39] to represent source code information. Here, syntax-tree representations can be queried (and transformed) using program specification and pattern matching capabilities. Jarzabek has proposed a Prolog-based static program analyzer (SPA) which is based on the program query language PQL [26]. ASTLOG [11] also has defined a Prolog-based query language which is intended for analyzing abstract syntax tree representations. Within the OMEGA experimental system [32] a relational model of a Pascal-like language called "Model" has been implemented. QUEL is used as query language. The C information abstraction system (CIA) [7] also uses a relational database to store extracted information. Information is retrieved using the INGRES query language. An information abstractor for the C++ programming language is also available [22].

Besides the use of queries in interrogation the other reengineering technologies are often also based on query facilities.

Browsing can be used to explore connections between related parts of a system and multiple system views [8]. If browsing is driven by a conceptual model every navigation step may be viewed as a query with respect to the object currently in focus. So it can be straightforwardly determined which path can be followed from a certain object. Browsing may be additionally integrated with query in that an entry point for a browsing session can calculated by a query.

In **software measurement** it is tried to map certain characteristics of software systems onto numerical values. Many kinds of metrics have so far been proposed addressing different types of software characteristics [25]. From a query point of view a metric an aggregate function that counts occurrences of certain software artifacts, takes average values, calculates quotients, and so on. A query-based approach used for software measurement has e. g. been proposed by Mendelzon and Sametinger [34] who use the Hy+ system together with the underlying Graphlog visual query language [10] to investigate object-oriented systems.

Cross referencing refers to finding out relationships between the components of a software system. In this context one is especially interested in call and use relationships. Cross references may be straightforwardly established using queries that relate two types of objects in a certain way.

Slicing as originally introduced by Weiser [44] was based on iterative solution of dataflow equations. Newer approaches do operate on dependence graph representations [24]. Especially in this graph-based context slicing may be supported by queries i. e. a query may be used to identify a subgraph that corresponds to a slice with respect to a given vertex in a control flow based representation.

Object recovery, aims at synthesizing objects from procedural code. A lot of work has been investigated in this reengineering technology. Object recovery is normally used to migrate procedural systems into object-oriented e. g. to transform procedural COBOL-II into OO-COBOL [20]. Because an object normally is a kind of regular repository substructure it may also be identified by queries. The same thing holds if a design has to be recovered from a system. In **design recovery**, higher abstractions of a system are generated, normally using domain knowledge or external information. Such external information can be provided by so called clichés e g. in form of graph patterns [46].

Similar to the program understanding techniques the software renovation techniques may be also based on query mechanisms. Especially the analysis components of renovation technology are candidates for query technology. But also synthesis resp. transformational components may have a query part.

Restructuring normally refers to changing the source code control structure in order to make a software easier to understand and easier to change [1]. Although this reengineering technology is rather old [4] it is today still needed e. g. in maintenance of legacy COBOL [43]. Because restructuring like control flow normalization includes a significant analysis share query technology may help a lot here.

In **remodularization** it is tried to change the module structure of a system according to common criteria like information hiding [36]. This reengineering technology is mostly based on cluster analysis [40] [45]. Queries may be used here in several ways. Especially the analysis of mavericks can be performed using queries.

Redocumentation means creating or updating information about the source code of a subject system [2]. Redocumentation originally concerned the embedding of comments [23]. Nowadays designs or specifications have to be considered, too. Redocumentation can also be essentially supported by queries. Strictly speaking any information that can be queried from a software can be annotated as a comment.

In order to show how reengineering technology may be supported by querying we will give some query examples in section 4.

3 The query approach to reengineering

In the following we will introduce our approach to querying as being embedded into a common framework.

The query approach shall be introduced along with the three step framework to source code that has been introduced by Tilley [42]. This approach proposes model, extract and abstract as the characteristic phases in source code analysis. **Modeling** refers to constructing a model of an application domain using conceptual modeling techniques [5]. **Extraction** means gathering data from the subject system using an appropriate extraction mechanism and **abstraction** refers to creating abstractions from these data that facilitate the actual reengineering task to be performed.

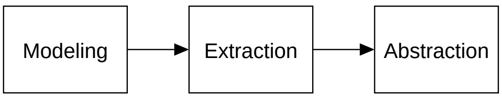

Figure 2. The three step approach to source code analysis

As a consequence to this three step approach a query facility has to come along with an adequate **formal basis**. This has to cover all phases of the source code analysis process in figure 2 in a **consistent and seamless** manner. So extraction should be done according to a conceptual model defined in the modeling phase and the abstraction facility i. e. the query engine has to work on a repository structure defined by the model. A formal basis is also important if a query facility shall be extended or if it has to be embedded into other applications. In addition a query facility has to be **powerful** enough to support a given task. E. g. if it is required to pursue indirect function calls, a language has to allow the closure of the call relationship to be calculated efficiently.

3.1 The *EER/GRAL* approach

In the following the query approach that has been used in the *GUPRO* project [16] will be sketched. This approach provides a seamless and consistent framework for querying source code representations.

In *GUPRO* **modeling** is enabled by the definition of graph classes. Graphs constitute a vivid formal mathematical model as well as an efficient data structure with time-tested algorithms providing a seamless approach to modeling and implementation [18][19]. Classes of graphs are specified using extended entity-relationship (EER) diagrams [6] that can be annotated by additional constraints in the $\mathcal{Z}$-like *GRAL* specification language [21]. Such *EER/GRAL* models are used to specify the underlying graph data structure. This consists of a rather general kind of graphs, called *TGraphs* [15]. These are directed, typed, attributed, and ordered. Entity types in the model refer to vertex types of a *TGraph* while relationship types refer to edge types. There is support for attribute structures as well as for advanced modeling concepts like generalization and aggregation. The semantics of the EER models is formally defined by specifying the class of graphs that suit to a given EER model [13]. The graph data structures are stored in the *GraLab* graph repository [14].

An example of an EER model is presented in figure 3. Here a fraction of the abstract syntax of the C programming language is shown. The complete declaration part is omitted due to its size and complexity.[1]

The **extraction** of source code information into the *GraLab* graph repository is enabled by parsers that for the most part are generated using the *PDL* parser generator [12]. *PDL* extends the Yacc parser generator [27] by EBNF syntax and by notational support for compiling textual languages into *TGraphs*.

In *GUPRO* **abstraction** is gained using the *GReQL* query language [28], which seamlessly suits the overall approach. Within the *GUPRO* project a generic toolset for program understanding has been developed. This can be parameterized by a specification of the actual maintenance problem, i. e. an *EER/GRAL* conceptual model, in order to derive concrete program understanding tool instances. An instance of the *GUPRO* toolset has been especially tailored to the multi-language software environment of a German insurance company [31]. This toolset provides the maintenance engineer with query and browsing facilities that can be used to explore cross references between the job con-

[1] In the EER dialect used vertex types are represented by rectangles, edge types are represented by (directed) arcs. Generalization is depicted by the usual triangle notation but also by graphically nesting object types. Within both notations an abstract generalization is symbolized by hatching. Aggregation is depicted by a rhomb at the vertex type rectangle. Relationship cardinalities are given by an arrow notation at the participating vertex types.

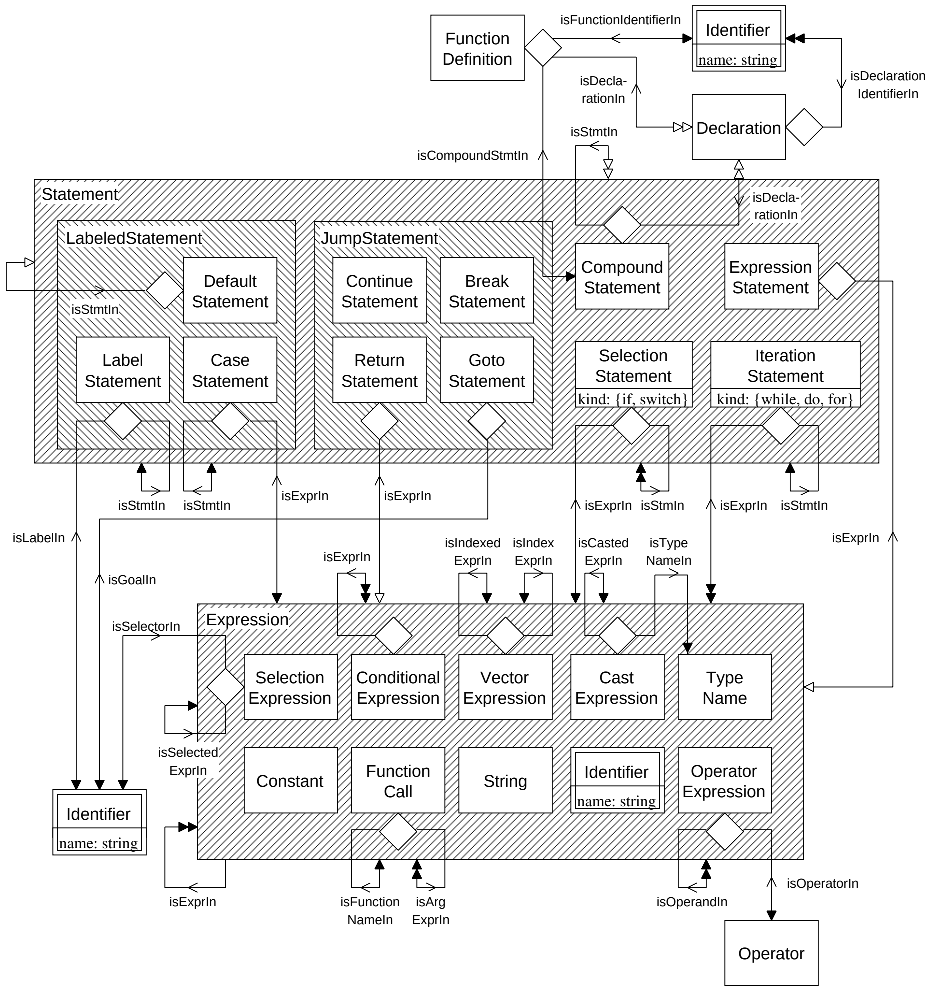

Figure 3. Concept model of the C programming language (extract)

trol languages, programming languages, and database languages.

Within *GUPRO* the extract-transform-rewrite *ETR* approach [17] represents a conceptual framework for software renovation that allows source codes to be consistently changed on a schema level. Within the prototype implementation form-oriented manual changes have been explored. Support for arbitrary automatic changes as they are needed in complex renovation tasks are possible too.

The results of the *GUPRO* project are related to general reengineering technology in section 4. Both, understanding an renovation techniques are based on the *GReQL* query facility which shall now be introduced.

3.2 The *GReQL* query language

In order to retrieve information about software systems the *TGraph* repository is analyzed using the *GReQL* (*GUPRO* Repository Query Language) query language. *GReQL* is an expression language that is especially suited to querying graph structures. Predicates in *GReQL* can be formulated using first order logic. Predicates may also contain path expressions to describe regular path structures i. e. sequences, alternatives and iterations (including transitive and reflexive closures) of paths in the repository. Path expressions can be used to collect sets of objects that can be reached via a specific kind of path from a designated object as well as to test whether a path exists between two objects.

The most important language element in *GReQL* is the FWR expression (FWR = FROM-WITH-REPORT). Within the `FROM` part the variables to be used in a query are declared by specifying their name and type. In the `WITH` clause the set of possible variable assignments is restricted to those specified by a predicate. The expressions specified in the `REPORT` part of a query are calculated and returned by the query. Because FWR expressions return values, i. e. FWR expressions may be nested.

A simple example of a *GReQL* query is shown in figure 4. Within the outer FROM part a variable a with type A is introduced. The outer WITH part restricts the possible assignments to a to those objects with value 42 for attribute x. The REPORT part specifies the name attribute of a to be considered together with the result of an inner FWR expression. This introduces a second variable b of type B which is restricted to those objects being related to a by a possibly empty sequence of edges of type C and a single edge of type D in opposite direction. The name attribute of each such object b is reported.

```
FROM   a : V{A}
WITH   a.x = 42
REPORT a.name,
       FROM   b : V{B}
       WITH   b -->{C}* <--{D} a
       REPORT b.name
       END
END
```

Figure 4. A simple *GReQL* example

A query in *GReQL* is evaluated by an EVAL/APPLY mechanism using an automaton-driven strategy for calculating path expressions efficiently (with respect to repository content). Queries can be statically optimized [38].

3.3 Types of interfaces

The *GReQL* query language is accessible through different kind of interfaces each providing a certain level of comfort and functionality. According to Codd [9] three types of query interfaces may in general be distinguished. A low-level **programming language interface** that is used by professionals e. g. to write application programs that operate on the data, or, in the current context, are embedded into other reengineering techniques. A programming language interface normally provides the most expressive power together with the lowest level of comfort. The second type of interface is a high-level, **stand-alone query interface**. This is normally used by technical or semi-technical users for adhoc retrieving the data. Non-technical users are in general confronted with additional **user-friendly interfaces**. These include form- or screen-oriented interfaces as well as natural language front ends.

In the context of *GReQL* there is support for each of these interface types. A **programming language interface** to *GReQL* (referred to as *inlineGReQL*) is available as an appropriate C++ class. *InlineGReQL* can be used by any program. A **stand-alone query facility** is available with the *GUPRO* query user interface. This provides the user with textual editing facilities and with support for loading and saving of queries and query results. The query user interface supports the *GReQL* query language to its full extent. A **user-friendly interface** to the *GReQL* query facilities comes along with *MeGGI* (Menu Guided *GReQL* Interface) [41]. *MeGGI* is a query interface that lets the user click his or her queries guided by schema information. The user is enabled to specify paths in the repository, logical combinations, aggregate functions, output options and simple constraints.

4 Applications for queries in reengineering

In the following it shall be shown how reengineering technology is supported by querying within our approach. Therefore some query examples for understanding and renovation techniques mentioned in section 2 are given as *GReQL* queries to the *GUPRO* repository. Furthermore it is shown how the extraction step of the query framework in figure 2 is supported by queries.

4.1 The use of queries in understanding

As pointed out in section 2 query mechanisms can be a useful support in understanding technology.

In *GUPRO* **interrogation** is enabled by a query user interface as well as by a user friendly interface as described in section 3.3.

In **browsing** query technology can be straightforwardly used to determine paths and goals for navigation. A hypertext-like browsing component has been developed as part of the GUPRO project [16]. This interacts with the query user interface such that the results of an interrogation are used as an entry point for browsing. So interrogation results can be viewed in terms of source code and they can be used as the basis for further investigations.

As said before in **software measurement** certain characteristics of a software are aggregated to numerical values. As part of the participation in the source code analysis engineering demonstration project [47] a large set of the metrics has been implemented for the C programming language [30] by applying *GReQL* queries to the repository.

As an example the number of decisions [33] shall now be introduced as a rather simple metrics. With respect to the C programming language (cf. figure 3) this can be expressed by the query given in figure 5.

```
cnt (  FROM    i : V{IterationStatement}
       REPORT  i   END ) +
cnt (  FROM    s : V{SelectionStatement}
       REPORT  s   END ) +
cnt (  FROM    c : V{ConditionStatement}
       REPORT  c   END )
```

Figure 5. Calculating the number of decisions

In this query the cnt aggregate function is used to count the relevant objects in the repository. The arithmetic operators + is used to calculate the intended result.

Cross references are of major importance in the understanding of programs and system. In the context of the C instance of the GUPRO toolset (cf. figure 3) e. g. indirect calls can be queried as shown in figure 6.

```
FROM     caller, callee : V{Identifier}
WITH     caller
         (
           -->{isFunctionIdentifierIn}
           <--{isCompoundStmtIn}
           <--{isStmtIn}*
           <--{isExprIn}*
           <--{isFunctionNameIn}
         )+
         callee
REPORT   caller.name AS Caller,
         callee.name AS Callee
END
```

Figure 6. Determining indirect call relationships

Here the relationship between a caller object and a callee object is established by the path expression in the WITH clause. Because indirect calls have to be considered as well, the whole path expression is iterated.

Other program understanding technologies like **slicing**, **object recovery**, or **design recovery** may based on the same query mechanisms. An adequate backend has to be provided for visualizing resp. saving the referring query result. Some serious effort has already been undertaken in basing slicing on query technology. In this context queries are used to infer additional edges resulting in a program dependence graph (PDG). Based on a PDG representation queries can again be used to calculate slices.

```
FROM     v, w :  V{PDGNode}
WITH     v.linenumber = 1249 AND
         v <--{PDG}* w
REPORT   w
END
```

Figure 7. Computing a backward slice

In figure 7 a backward slice is computed for the statement or expression in line 1249 in that all vertices in the PDG representation from that the corresponding vertex can be reached are reported.

4.2 The use of queries in renovation

Software renovation has been introduced as improving a software system in order to increase its quality, understandability and maintainability. Restructuring, remodularization, and redocumentation have been presented as renovation technologies.

Within our approach the renovation aspect of reengineering is represented by the extract-transform-rewrite cycle [17]. A source document is parsed into its internal graph representation. An **extract** operation on this representation is performed. The extract information can be **transformed** automatically or by form-based textual editing. A modified extract structure is integrated with the original source in a **rewrite** step. A final unparse step yields a source code document that reflects the change(s) performed. Especially extracting but also transformation and rewriting are essentially based on *GReQL* queries. The *ETR* cycle has been implemented as a prototype for the C programming language.

To illustrated the use of queries in the context of the *ETR* approach the form-based renaming of identifiers is used as an example. Again we refer to the conceptual model in figure 3. The query in figure 8 may be used to extract the identifiers that are locally defined in a function named printHeaderLabels. The path expression of that query starts with the object representing the referring function. It collects all identifier objects that are defined in a declaration that belongs to the function block or a block nested in this.

```
FROM     f :  V{FunctionDefinition},
         i, j :  V{Identifier}
WITH     f <--{isFunctionIdentifierIn} i AND
         i.name = 'printHeaderLabels' AND
         f <--{isCompoundStmtIn}
           <--{isStmtIn}*
           <--{isDeclarationIn}
           <--{isDeclarationIdentifierIn} j
REPORT   j
END
```

Figure 8. Extraction of local identifiers

If an identifier shall be renamed then it has to be ensured that no identifiers of the same name exist within the same scope and name space. Also no identifier from an outer scope must be overwritten if it is used in the same or an inner scope. This second condition may be checked using the query in figure 9 which collects all identifier objects that may cause a referring rename conflict. The query is strongly simplified by the use of inferred edges that relate identifier objects, scopes and name spaces. These inferred edges have been defined by queries [35].

A conflicting object has to belong to an outer scope, it has to belong to the same name space, it has to be used in the same or an inner scope, and it has to have the same name.

There is evidence that other renovation techniques as remodularization, restructuring, redocumentation that have not been implemented so far can also be supported by queries.

```
FROM     i, j : V{Identifier}
WITH     i -->{belongsToScope}
           (<--{isContainedInScope} )+
           <--{belongsToScope} j AND
         (i --> {belongsToNameSpace}
           <--{belongsToNameSpace} j) AND
         (i -->{isUsedInScopeOf}
           <--{belongsToScope} j}) AND
         (i.name = j.name)
REPORT   j
END
```

Figure 9. Checking a conflict in renaming

4.3 The use of queries in extraction

As soon as a reengineering technology is confronted with multiple languages or multiple files or systems the parsing strategy has to include some support for integration. If a repository is filled incrementally or if it has to be updated with new source code versions local updates become necessary. Within an update the referring components have to be identified and removed first. It has to be guaranteed that no other components are affected by a removal. Now the newly parsed component has to be integrated into the repository. Such an integration is normally based on some kind of anchoring objects. Additionally the relationships to the components in the repository are inferred from the existing information. Within our approach this general parsing strategy is essentially based on queries to the repository [29].

In order to motivate an integration example the model from figure 3 shall now be extended with embedded SQL as depicted in figure 10. Here an SQL statement is modeled as a subtype of a C statement. It refers to some DB2 table via the *usesTable* relationship.

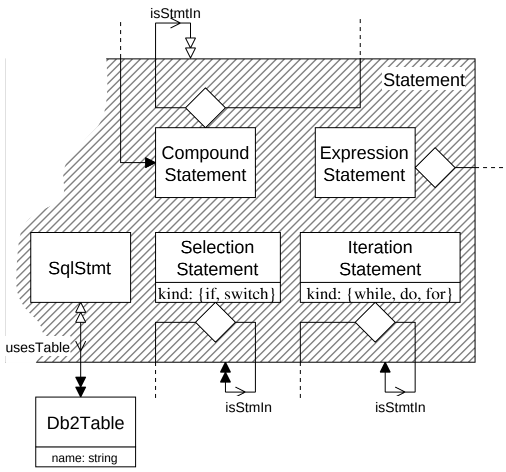

Figure 10. Embedding SQL with C

If multiple C sources are to be parsed into the repository it has to be ensured that the objects of type *Db2Table* which refer to the same DB2 table are merged into each other. In parsing this can be described using the merge rule in figure 11.

```
USING  anchor
FROM   new : V{Db2Table}
WITH   anchor -->{}* -->{usesTable} new
REPORT SET
       FROM   old : V{Db2Table}
       WITH   old.name = new.name
       REPORT old
       END,
       new
END
```

Figure 11. Merging a program into a system

Starting with a designated anchor object `anchor` of a newly parsed C source this query collects all *Db2Table* objects contained in the newly parsed graph and reports all other *Db2Table* objects from the repository having the same name. In an integration step these have to be merged in pairs.

5 Conclusion

In this paper we worked out the usefulness and importance of querying in reengineering technology. In this context a general reengineering taxonomy has been presented that subdivides existing reengineering technology into understanding technology and renovation technology. We tried to identify the query aspects of the existing reengineering technology from these two branches.

Our general approach to graph-based conceptual modeling and implementation has been presented together with *GReQL* as the accompanying query facility. The application of *GReQL* within reengineering technology has been shown with respect to the understanding branch as well as with respect to the renovation branch.

Acknowledgement

We would like to thank all people working in *GUPRO*. Special thanks to Jürgen Ebert for valuable discussions that improved this work very much.

References

[1] R. S. Arnold. Software Restructuring. *Proceedings of the ACM*, 77(4):607–617, April 1989.

[2] R. S. Arnold. A Roadmap Guide to Software Reengineering Technology. In *Software Reengineering*. IEEE Computer Society Press, 1993.

[3] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering*, pages 482–498. IEEE Computer Society Press, Apr. 1993.

[4] C. Böhm and G. Jacopini. Flow diagrams, Turing machines, and languages with only two formation rules. *Communications of the ACM*, 9(5):366–372, May 1966. Presented as an invited talk at the 1964 International Colloquium on Algebraic Linguistics and Automata Theory.

[5] M. Brodie, J. Mylopoulos, and J. W. Schmidt, editors. *On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases and Programming Languages*. Springer, New York, 2 edition, 1986.

[6] M. Carstensen, J. Ebert, and A. Winter. Deklarative Beschreibung von Graphsprachen (Erweiterte Kurzfassung). In F. Simon, editor, *Tagungsband zum Workshop "'Deklarative Programmierung und Spezifikation"', der GI-Fachgruppe 2.1.4 Alternative Konzepte für Sprachen und Rechner, 9.-11. Mai 1994, Bad Honnef*, number Bericht 9412. Kiel, September 1994.

[7] Y.-F. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, Mar. 1990.

[8] L. Cleveland. A program understanding support environment. *IBM Systems Journal*, 28(2):324–344, 1989.

[9] E. F. Codd. Seven steps to rendezvous with the casual user. In J. W. Klimbie and K. L. Koffeman, editors, *Data Base Management*, pages 179–200. North-Holland, 1974.

[10] M. Consens and A. Mendelzon. The G^+/GraphLog Visual Query System. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 19(2):388–388, June 1990.

[11] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, pages 229–242, Berkeley, Oct.15–17 1997. USENIX Association.

[12] P. Dahm. Parser Description Language — An Overview. In *[16]*, pages 137–156. 1998.

[13] P. Dahm, J. Ebert, A. Franzke, M. Kamp, and A. Winter. TGraphen und EER-Schemata — Formale Grundlagen. In *[16]*, pages 51–66. 1998.

[14] P. Dahm and F. Widmann. Das Graphenlabor. Fachberichte Informatik 11/98, Universität Koblenz-Landau, Institut für Informatik, Koblenz, 1998.

[15] J. Ebert and A. Franzke. A Declarative Approach to Graph Based Modeling. In E. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graphtheoretic Concepts in Computer Science*, number 903 in LNCS, pages 38–50, Berlin, 1995. Springer.

[16] J. Ebert, R. Gimnich, H. Stasch, and A. Winter, editors. *GUPRO — Generische Umgebung zum Programmverstehen*. Koblenzer Schriften zur Informatik. Fölbach, Koblenz, 1998.

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

[18] J. Ebert, A. Winter, P. Dahm, A. Franzke, and R. Süttenbach. Graph Based Modeling and Implementation with EER/GRAL. In B. Thalheim, editor, *15th International Conference on Conceptual Modeling (ER'96), Proceedings*, number 1157 in LNCS, pages 163–178, Berlin, 1996. Springer.

[19] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schürr. Building integrated software development environments part I: Tool specification. *ACM Transactions of Software Engineering and Methodology*, 1(2):135–167, Apr. 1992.

[20] H. Fergen, P. Reichelt, and K. P. Schmidt. Bringing Objects into COBOL, MOORE - A tool for migration from COBOL85 to object-oriented COBOL. In *Proccedings of the Conference on Technology of Object-Oriented Languages and Systemes (TOOLS 14)* , pages 435–448. Prentice Hall, Santa Barabara, August 1994.

[21] A. Franzke. GRAL: A Reference Manual. Fachbericht Informatik 3/97, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, 1997.

[22] J. E. Grass. Object-oriented design archaeology with CIA++. *Computing Systems*, 5(1):5–67, Winter 1992.

[23] K. Heninger, J. Kallander, D. Parnas, and J. Shore. Software Reuqirements for the A-7E Aircraft. NRL Memorandum Report 3876, Nov. 1978.

[24] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.

[25] D. Ince. An Annotated Bibliography of Software Metrics. *ACM SIGPLAN Notices*, 25(8):15–23, Aug. 1990.

[26] S. Jarzabek. PQL: A language for specifying abstract program views. In W. Schäfer and P. Botella, editors, *Software Engeneering - ESEC '95. Proceedings*, volume 989 of *LNCS*, pages 324–342, Berlin, 1995. Springer.

[27] S. C. Johnson. YACC — Yet another compiler - compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, N.J., 1975.

[28] M. Kamp. GReQL-Sprachbeschreibung. In *[16]*, pages 173–202. 1998.

[29] M. Kamp. Managing a Multi-File, Multi-Language Software Repository for Program Comprehension Tools – A Generic Approach. In U. D. Carlini and P. K. Linos, editors, *6th International Workshop on Program Comprehension*, pages 64–71, Washington, June 1998. IEEE Computer Society.

[30] B. Kullbach. Approaching WELTAB III using GUPRO. The 6th Reengineering Form, March 8-11, Firenze, Italy, 1998, 1998.

[31] B. Kullbach, A. Winter, P. Dahm, and J. Ebert. Program Comprehension in Multi-Language Systems. In *Proceedings of the 5th Working Conference on Reverse Engineering 1998 (WCRE'98)*, 1998. to appear.

[32] M. A. Linton. Implementing Relational Views of Programs. *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Developement Environments*, pages 132–140, May 1984.

[33] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[34] A. Mendelzon and J. Sametinger. Reverse Engineering by visualizing and querying. *Software—Concepts and Tools*, 160(4):170–182, 1995.
[35] A. Panse. Konzeption und Realisierung eines Reengineering-Werkzeugs. Eine Fallstudie des ETR-Zyklus. Diplomarbeit, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, Januar 1998.
[36] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.
[37] S. Paul and A. Prakash. A Query Algebra for Program Databases. *IEEE Transactions on Software Engineering*, 22(3):202–217, Mar. 1996.
[38] D. Polock. Ein statischer Optimierer für GRAL- und GReQL-Ausdrücke. Diplomarbeit D 414, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, September 1997.
[39] Reasoning Systems. REFINE User's Guide, 1989.
[40] R. W. Schwanke. An Intelligent Tool for Re-engineering Software Modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, May 1991.
[41] N. Südkamp and R. Gimnich. Benutzeroberflächen für den *GUPRO*-Prototyp. In *[16]*, pages 205–218. 1998.
[42] S. R. Tilley. *Domain-Retargetable Reverse Engineering*. PhD thesis, Department of Computer Science, University of Victoria, January 1995.
[43] M. van den Brand, A. Sellink, and C. Verhoef. Control Flow Normalization for COBOL/CICS Legacy Systems. In P. Nesi and F. Lehner, editors, *Proceedings of the 2nd Euromicro Conference on Software Maintenance & Reengineering*, pages 11–19, Los Alamitos, 1998. IEEE Computer Society.
[44] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, Mar. 1981.
[45] T. A. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings of the 4th Working Conference on Reverse Engineering*, pages 33–43, Los Alamitos, California, 1997. IEEE Computer Society Press.
[46] L. M. Wills. Using Attributed Flow Graph Parsing to Recognize Clichés in Programs. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. Fifth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci.*, volume 1073 of *Lecture Notes in Computer Science*, pages 170–184. Springer, 1996.
[47] WorldPath Information Services. Reverse Engineering Demonstration Project. http://www.worldpath.com/reproject/, 1998.