Electronic Notes in Theoretical Computer Science 72 No. 2 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume72.html 10 pages

GUPRO – Generic Understanding of Programs

An Overview

Jürgen Ebert [1] Bernt Kullbach [2] Volker Riediger [3] Andreas Winter [4]

Universität Koblenz-Landau, Institut für Softwaretechnik
D-56016 Koblenz, Postfach 201602

Abstract

GUPRO is an integrated workbench to support program understanding of heterogenous software systems on arbitrary levels of granularity. *GUPRO* can be adapted to specific needs by an appropriate conceptual model of the target software.

GUPRO is based on graph-technology. It heavily relies on graph querying and graph algorithms. Source code is extracted into a graph repository which can be viewed by an integrated querying and browsing facility. For C-like languages *GUPRO* browsing includes a complete treatment of preprocessor facilities.

This paper summarizes the work done on *GUPRO* during the last seven years.

1 Introduction

Instead of producing new software systems from scratch, nowadays software development has to deal more and more with understanding and reworking legacy programs which have evolved over times. Studies on the effort on software development prove that the portion spent for software maintenance has increased dramatically from 49% in 1977 [28] to more than 90% in 1995 [29] (cf. [30, p 31]). Incremental software development approaches like Extreme Programming [3] suspend the artifical distinction between software development and software maintenance. Understanding, changing, correcting, and adapting software systems are essential activities during software development. Thus, research on software engineering has to provide methods and tools supporting software evolution.

Software reengineering summarizes all activities that either support the *understanding of software* or *improve the software itself* [2]. Major activities in reengineering deal with

(i) *reverse engineering software systems*, e. g. detecting software components and their interrelationships to provide multiple views of software systems at a higher level of abstraction

[1] email: ebert@uni-koblenz.de, url: www.uni-koblenz.de/∼ebert.
[2] email: kullbach@uni-koblenz.de, url: www.gupro.de/kullbach.
[3] email: riediger@uni-koblenz.de, url: www.gupro.de/riediger.
[4] email: winter@uni-koblenz.de, url: www.gupro.de/winter. Andreas Winter is currently visiting University of Waterloo, School of Computer Science, Waterloo, Canada.

(ii) *comprehending software systems*, e. g. learning what software components do, how they operate, and how they interact,
(iii) *evolving software systems*, e. g. correcting, changing, adapting, and extending software systems.

Activities in reverse engineering and in program comprehension, during isolated maintenance phases as well as throughout incremental software development processes, follow an *Extract–Abstract–View–Metaphor* [27]. Data about source code artifacts are *extracted* into a *software repository*. Using various analysis techniques, these data are *abstracted* to provide a deeper understanding of the software system. Abstractions of the system are then *viewed* by appropriate visualization means.

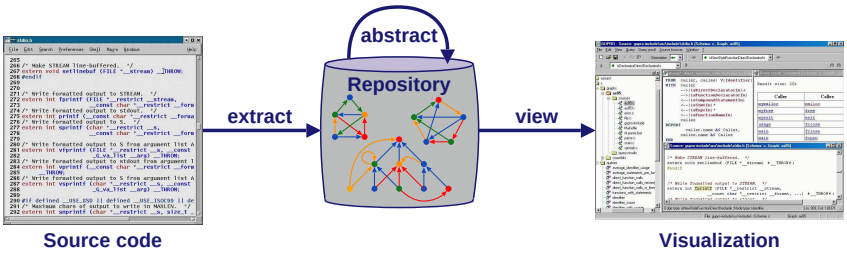

Fig. 1. Reference architecture for Reverse Engineering Tools

The *Extract–Abstract–View–Metaphor* also serves as a reference architecture for reverse engineering tools (cf. figure 1) and provides a classification framework for reverse engineering techniques. Integrated reverse engineering tools like Bookshelf [13], DALI [21], PBS [17], Rigi [36], SoftANAL [27], and SWAGKIT [33] follow this reference architecture. These tools differ in the addressed reverse engineering problem, in the underlying repository techniques, in the granularity of source code representation, and in the analysis and visualization techniques.

This paper summarizes the *GUPRO* [8] reverse engineering approach. The objective of *GUPRO* (Generic Understanding of PROgrams) is to provide an *integrated reverse engineering workbench* supporting multiple program analysis techniques. *GUPRO* is strongly based on *graph technology* [10]. Software artifacts are stored in a *graph repository*, abstraction is done by *graph queries* and *graph algorithms*.

Section 2 introduces the graph-based *GUPRO*-repository. Using the *EER/GRAL* graph based conceptual modeling approach, the structure of the *GUPRO*-repository can be adapted to the special needs of different reverse-engineering problems. Section 3 depicts the population of *GUPRO*-repositories by specialized parser frontends. The abstraction facilities of *GUPRO* based on graph-queries are explained in section 4. Section 5 describes the *GUPRO*-browser for software visualization. A short conclusion in section 6 closes this overview.

2 Representing Source Code

Mostly, documentation of software systems is incomplete, outdated, incorrect, or does not fit the current version of the system to be reverse-engineered. Thus, pro-

gram code is the only reliable source for starting program understanding and reengineering activities.

Depending on the actual reverse engineering problem and the aspired program analysis technique, different (internal) code representations have to be chosen. E. g. describing software architectures, stressing the module- and call structure [5, p. 36f], requires a coarse grained representation of modules, call- and/or include-relationships. In contrast, a detailed dataflow- and controlflow analysis requires a fine grained representation of each source code object (variables, methods, paragraphs) and their occurrences in program statements. Hence, a general reverse engineering workbench has to cope with different data structures or reverse engineering repositories including corresponding analysis operations.

Graphs offer a general data structure since efficient and universal algorithms are known which can be applied to nearly all reverse engineering analysis techniques. An overview on graph-based reverse engineering tools, including their interoperability, is shown in [18]. *GUPRO* 's graph model is given by *TGraphs* , which provide a universal, expressive, and powerful way of modeling (cf. section 2.1). The *EER/GRAL* conceptual modeling approach [10] is used to define graph structures matching the requirements of certain analysis techniques (cf. section 2.2). Implementation support for storing and analyzing *TGraphs* is given by the *GraLab* graph library [6] and by the *GReQL* graph query language (section 4).

2.1 Graphs

Different program analysis techniques require different underlying graph models, e. g. directed graphs, undirected graphs, node attributed graphs, edge attributed graphs, node typed graphs, edge typed graphs, ordered graphs, relational graphs, acyclic graphs, trees, etc. or combinations of these. To support multiple program analysis techniques in one reverse engineering workbench, the underlying graph model has to be as rich as possible to cover most of the required graph models. Such a common graph model is given by *TGraphs* [9]. *TGraphs* are *directed* graphs, whose nodes and edges may be *attributed* and *typed*. Each type can be assigned an individual attribute schema specifying the possible attributes of nodes and edges. Furthermore, *TGraphs* are *ordered*, i. e. the node set, the edge set, and the sets of edges incident to a node have a total ordering. This ordering gives modeling power to describe sequences of objects (e. g. parameter lists) and facilitates the implementation of deterministic graph algorithms. In applying *TGraphs* , not all properties of TGraphs have to be used to their full extent. The individual graph models cited above can all be viewed as specializations of *TGraphs* .

Figure 2 shows a program fragment and its *TGraph* representation on abstract-syntax-graph-level. The functions *main*, *max* and *min* are represented by nodes of type *Function*. These nodes are attributed with the function name. *FunctionCall* nodes represent the calls of functions *max* and *min*. They are associated to the caller by *isCaller* edges and to the callee by *isCallee* edges. *isCaller* edges are attributed with a *line* attribute showing the line number which contains the call.

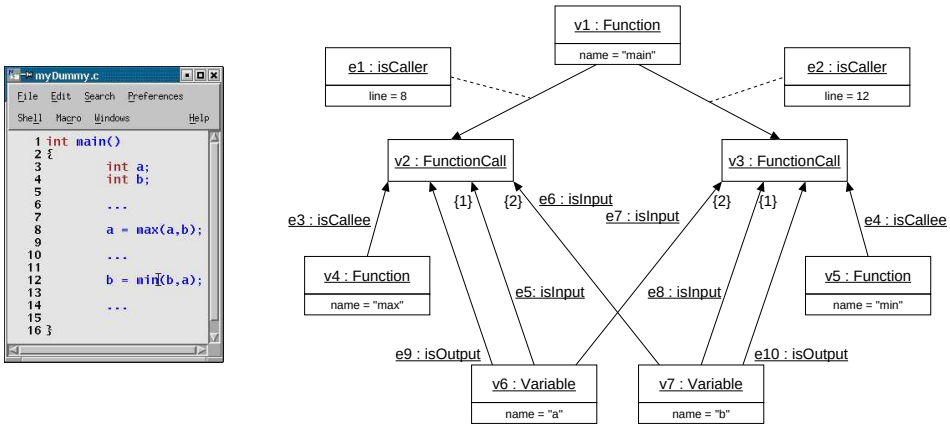

Fig. 2. typed, attributed, directed, ordered graph

Input parameters (represented by *Variable* nodes that are attributed with the variable name) are associated by *isInput* edges. The ordering of parameter lists is given by ordering the incidences of *isInput* edges pointing to *FunctionCall* nodes. The first edge of type *isInput* incident to function call *v2* (modeling the call *max(a,b)*) comes from node *v6* representing variable *a*. The second edge of type *isInput* connects to the second parameter *b* (node *v7*). The incidences of *isInput* edges associated with node *v3* model the reversed parameter order. Output parameters are associated to their function calls by *isOutput* edges.

2.2 Graph-Classes

TGraphs provide a simple structural graph-based means for applying graph algorithms and graph queries. Different reverse engineering tasks require different *TGraph* structures. To implement the required granularity of the source code representation and the aspired reverse engineering techniques, different node-classes, edge-classes, and incidence structures are necessary. Equally, the attribute structure of node- and edge-classes, and the incidence relation between these classes depends on the reengineering problem to be solved. Furthermore, class hierarchies and additional constraints e. g. ordering of incidences or multiplicity constraints have to be described.

These structural data on graphs can be defined by *conceptual modeling techniques*. *GUPRO* follows the *EER/GRAL* approach on graph based conceptual modeling [10]. Class diagrams offer a suited declarative language to specify the required graph classes with respect to a certain reverse engineering problem. Additional constraints are specified by using the *GRAL* graph specifiation language [14].

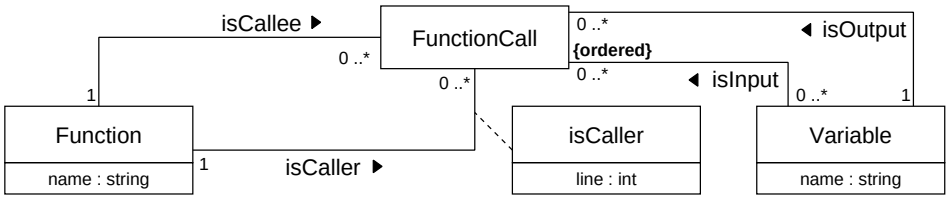

Fig. 3. Graph class definition for graphs like the one in figure 2

Graphs like the one in figure 2 offer a fine grained representation of program structures, focusing on the description of functions, function calls, variables and their interdependencies. Figure 3 shows a possible graph class definition of these graphs, depicted as an UML class diagram [4]. Node classes (*FunctionCall*, *Function*, and *Variable*) are defined by UML classes. Edge classes (*isCallee*, *isInput*, and *isOutput*) are defined by associations. Attributed edge types (*isCaller*) are described by UML association classes.

Genericity in *GUPRO* is strongly provided by using graph classes. Program analysis in *GUPRO* uses a *graph class independent query approach*. Thus, the general query-based analysis facilities can be combined with *problem oriented graph schemas* to offer proper reverse engineering support. Currently *GUPRO* provides graph schemas for several applications including

- architectural analysis of multi-language software in an insurance company (Cobol, PL/I, CSP, JCL, IMS-DB, SQL) [26], [8],
- architectural analysis of Java/C/C++/RDBMS based software of a stock trading system [27],
- fine grained analysis of C [31], [32] and Ada programs [24] in the context of security analysis and certification, and
- fine grained representation and analysis of JCL-job-descriptions of a Bull-host system [34].

Current work deals with adapting the fine grained C++-Schema developed within the Columbus C++-parser frontend [12].

3 Extracting Facts from Source Code

All analysis activities in reverse engineering and program understanding depend on the data stored in the repository. Thus, populating the repository is a fundamental step in all program understanding and reverse engineering processes. Extracting facts from source code has to be accurate and reliable [11].

The development of fact extractors for reverse engineering is based on knowledge from the area of compiler construction. (cf. [1]). In particular, extracting facts from software systems has to deal with additional problems like multi-language assets or the use of preprocessors.

Software systems usually consist of a huge number of strongly connected sources in different programming languages, data base definitions, JCL-texts, etc. E. g. the software system of a large German insurance company consists of about 25 000 JCL-, 5 700 CSP-, 7 800 COBOL II- and Delta COBOL-, 6 000 PL/1-, 1 000 Assembler-, 100 REXX sources as well as programs written in languages like APL, SAS or Easytrieve. This is complied by 100 data models with about 3 000 entities and 60 000 attributes. Thus, the reengineering repository has to represent large *multi-language software systems* [26]. Due to the size of such software systems, it is not feasible to fill reverse-engineering repositories with all sources at once. Especially a change in the system has to be mirrored in the repository incrementally.

GUPRO-fact extractors for multi-languages systems follow a four step parsing approach [19]. The first step checks if the document is already represented in the

repository in a former version. If so, its facts are removed. The document itself is then parsed in a second step. In the third step, the extracted facts are integrated into the existing repository and a fourth step ensures further integrity constraints. These parsing steps are controlled by a set of graph-queries (cf. section 4), e. g. to discover graph objects to be deleted in step 1 or to be merged in step 3.

Preprocessors are used in many programming environments to increase the expressiveness of languages by supporting macro definition and expansion, conditional compilation and even low level configuration management. But unfortunately, preprocessors significantly complicate program understanding, since *what the user sees is not what the compiler gets*. Especially fine grained data flow and control flow analysis must be based on the preprocessor output, whereas the analysis results have to be presented in the notation used by the programmer, i. e. in preprocessor input.

The *GUPRO* folding approach [23] provides an integrated representation of preprocessor input and output in the reverse engineering repository. Each macro usage is represented by its macro-label and its macro-expansion in a fold structure which is created by a special preprocessor. The fold structure also memorizes the current visualization status of each macro call, i. e. if the macro is expanded or folded. This structure is connected to the representation of the preprocessor output. Thus, program analysis can be done on the preprocessor output, and the analysis results can be presented in any level between the programmers view on the source code to the compilers view on the source code. Since the fold structure is a graph, and *GUPRO* analyzes graphs, it may serve as an analysis subject providing facts about preprocessor usage.

GUPRO currently supports extracting facts on architectural level for multi-language systems consisting of COBOL, CSP, MVS/JCL, PSB, SQL (DDL and DML), and IMS-DBD sources [26]. Parsers for extraction on abstract syntax tree level including the extraction of fold structures are provided for C [32] and Ada [24]. Using the *GXL* Graph Exchange Language [35], standard C++ parser-frontends like Columbus [12] and CPPX [7] can be used within *GUPRO* . Further *GXL* based filters exist for converting a database repository representing a Java/C/C++/RDBMS system on architectural level to the graph based *GUPRO* repository [27].

4 Abstracting From Source Code

Most reverse engineering techniques can be mapped on querying a reverse-engineering repository [25]. *GUPRO* uses a schema-independent querying mechanism. Accordingly, customizing *GUPRO* 's analysis facilities to a special reverse engineering problem only requires to parameterize the general *GUPRO* query engine with the appropriate schemas. The following sections briefly introduce the *GReQL Graph Repository Query Language* [20].

GReQL is a *declarative expression language* which is especially tailored to query graph structures. *GReQL* is designed as a pure query-only language that does not change or extend the queried repository. Usually, a *GReQL* query consists of three parts. In the `FROM` clause the relevant graph elements are declared by specifying a

variable name and a type (nodes and edges are treated equally). Predicates which have to be fulfilled by these objects are specified in the `WITH` clause. Here, *GReQL* provides first order logic on finite sets. *GReQL* supports various graph oriented predicates including regular path expressions, e. g. sequences, alternatives and iterations (reflexive and transitive closure) of paths in the queried graph. The `REPORT` clause describes the appearance of the query result. `FROM-WITH-REPORT` expressions may be nested.

```
FROM caller, callee :  V{Function}
WITH caller --> {isCaller} <-- {isCallee} callee
REPORT caller.name, callee.name
END
```

Fig. 4. A simple *GReQL* query

Figure 4 shows a simple *GReQL* query calculating all caller/callee-pairs in graphs like those specified by the graph class in figure 3. Variables (`caller` and `callee`) of node class `Function` are declared in the `FROM` clause. The path predicate `caller -->{isCaller}<-- {isCallee} callee` in the `WITH` clause indicates that `caller` and `callee` have to be connected by a sequence of an outgoing `isCaller` and an incoming `isCallee` edge. For all `caller`/`callee` pairs matching this path predicate, the `REPORT` clause defines to display the associated `name` attributes. Applied to the graph in figure 2, this query will report $[\![(\texttt{main}, \texttt{max}), (\texttt{main}, \texttt{min})]\!]$.

Figure 5 depicts a more complicated query within the *GUPRO*-tool. For each caller function, it calculates the set of directly and indirectly called functions with respect to a fine grained C schema [31]. The path expression expressing the connection between `caller` and `callee` follows various edges of different edge classes. This path expression contains (reflexive) transitive closures of various edges. To obtain the indirectly called functions, the complete path expression is iterated as well.

Querying graphs with *GReQL* is currently supported by three interfaces. The *GUPRO* -Reverse-Engineering-Workbench enables interactive querying. A standard API for embedding *GReQL* queries in C++-programs comes with the *GraLab* graph library [6]. *CLG* (Command Line *GReQL*) [22] offers a script language version of *GReQL* which provides sequential calculation of *GReQL* queries, including the reuse of intermediate query results in subsequent queries. *CLG* also supports various export formats for query results, e. g. HTML, XML, and CSV.

5 Visualizing Source Code

Supporting program understanding primarily focuses on understanding connections between source code artifacts. Reverse engineers want to query the software system and map the query results into the original source code. Thus, *GUPRO* supports program understanding by visualizing query results in (nested) tables and by displaying source code directly.

The *table view* gives a first overview about those software objects complying with the appropriate query. These query-results are directly linked to the source code, if possible. A simple mouse click on an object in the table view directly opens a

source code browser with a *code view* showing the appropriate occurrence of this source object. But the tables may also contain other query results, e. g. software metrics, which are not directly related to a certain code position.

Figure 5 shows a screenshot of the *GUPRO* Reverse Engineering Workbench. The left panel shows the current project. Current analysis deals with the C system `as85` of the graph class `c` in the folder `c/graphs`. The folder `queries` collects all queries suiting the C schema. The *GReQL* query `indirect_function_calls_nested` is shown in the query editor and the query result is displayed in a table view. The code view in the lower part of figure 5 directly corresponds to the call of function `match` by `p_term`, displayed in the table view.

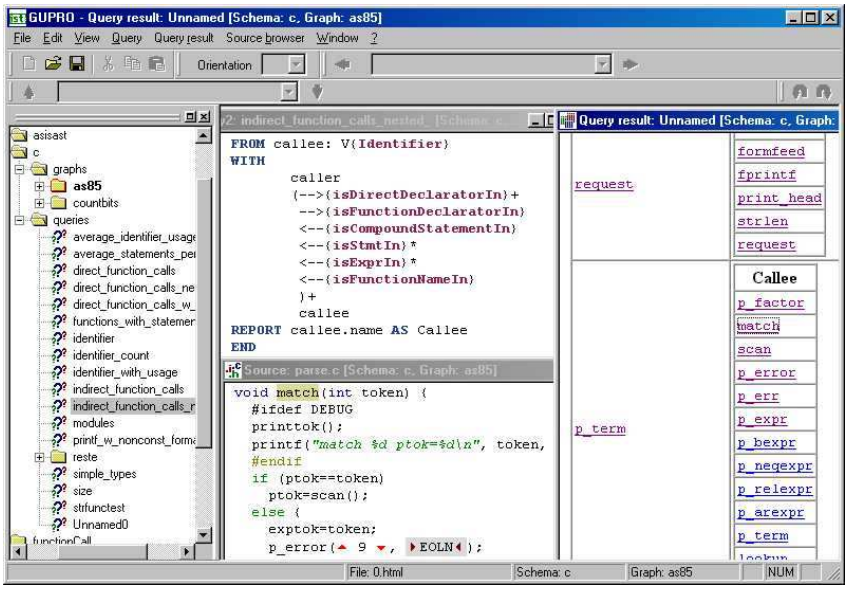

Fig. 5. Visualization in *GUPRO*

Table view and code view are strongly integrated and *GUPRO* users can switch between both visualizations. Likewise, selecting a source code object in the code view provides a starting point for browsing the source code. According to the relations between source code objects represented in the repository, reverse engineers can explore the connected source code objects. Thus the browsing facility supports navigation according to the conceptual model. Furthermore, source code browsing in *GUPRO* is also supported by graph queries. Starting from the selected code object a *GReQL* query calculates all appropriate code objects. These objects again are displayed in a table or highlighted in a code view.

By using the *GUPRO* folding approach (cf. section 3), source code of preprocessed languages can be displayed at any level from preprocessor input, which was originally written by the programmer, to preprocessor output, which is the foundation for source code analysis. The code browser in figure 5 shows an expanded and an unexpanded preprocessor macro. Preprocessor input ▲ 9 ▼ was expanded from macro `E_EXPECT` and the unexpanded preprocessor input ► *EOLN* ◄ might be expanded to `'\n'`. Here, the triangles indicate the fold status of these macros.

Current work on the visualization components of *GUPRO* deals with developing a

graphical view on the repository and the stored software objects. Using *GXL* [35] as input to simple format converters, the underlying graph structures are displayed with standard graph visualizer tools like DaVinci [15] or GraphViz [16].

6 Conclusion

This paper gave a brief overview over the *GUPRO* integrated Reverse Engineering Workbench. It summarized the work done during the last seven years. The aim of *GUPRO* is to support reverse engineering and program understanding of heterogenous software on arbitrary levels of granularity. To achieve this, *GUPRO* is adaptable by a conceptual model of the relevant information. This model implies the structure of the graph-based *GUPRO*-repository. Source code is extracted into the repository and the repository graphs can be viewed by an integrated querying and browsing facility.

GUPRO heavily relies on graph querying and and graph algorithms. For C-like languages *GUPRO* browsing includes a complete treatment of preprocessor facilities.

GUPRO-instances were developed for supporting architectural analysis of heterogeneous multi-language systems used in an insurance company. Further instances are applied support the analysis of C and Ada-Systems with regard to securitiy certification.

Focus of the current work in the *GUPRO* project is the combination of GUPRO with other reengineering tools via *GXL*-interfaces.

References

[1] Aho, A. V., R. Sethi and J. D. Ullman, "Compilers. Principles, Techniques, and Tools", Addison-Wesley, 2000.

[2] Arnold, R. S., *A Roadmap Guide to Software Reengineering Technology*, in: Arnold, R. S., (ed.), "Software Reengineering", IEEE Comp. Soc. Press, 1993.

[3] Beck, K., "Extreme Programming Explained", Addison-Wesley, 1999.

[4] Booch, G., J. Rumbaugh and I. Jacobson, "The Unifi ed Modeling Language User Guide", Addison Wesley, 1999.

[5] Bras, L., P. Clements and R. Kazman, "Software Architecture in Practice", Addison-Wesley, 1998.

[6] Dahm, P. and F. Widmann, *Das Graphenlabor*, in: *[8]*, 1998, 67–84.

[7] Dean, T., A. Malton and R. Holt, *Union Schemas as a Basis for a C++ Extractor*, in: 8th Working Conference on Reverse Engineering, IEEE Comp. Soc., 2001, 59–67.

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

[9] Ebert, J., B. Kullbach and A. Winter, *GraX – An Interchange Format for Reengineering Tools*, in: "6th Working Conference on Reverse Engineering", IEEE Comp. Soc., 1999, 89–98.

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

[11] Elliott Sim, S., R. C. Holt and S. Easterbrook, *On Using a Benchmark to Evaluate C++-Parsers*, "10th International Workshop on Program Comprehension", IEEE, 2002.

[12] Ferenc, R. and A. Beszédes, *Data Exchange with the Columbus Schema for C++*, in: "6th Conference on Software Maintenance and Reengineering", IEEE Comp. Soc., 2002, 59–66.

[13] Finnigan, P. J., R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S.

G. Perelgut, M. Stanley and K. Wong, *The software bookshelf*, IBM Systems Journal 36 1997, 564–593.

[14] Franzke, A., *GRAL: A Reference Manual*, Fachbericht Informatik 3/97, Universität Koblenz-Landau, Fachbereich Informatik, 1997. http://www.uni-koblenz.de/universitaet/fb/fb4/publications/GelbeReihe/RR-3-97.ps.gz

[15] Fröhlich, M. and M. Werner, *daVinci V2.0.x Online Documentation*, http://www.tzi.de/davinci/docs/.

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

[17] Holt, R. C., *PBS: Portable Bookshelf*, http://www.turing.toronto.edu/pbs.

[18] Holt, R. C., A. Winter and A. Schürr, *GXL: Toward a Standard Exchange Format*, in: "7th Working Conference on Reverse Engineering", IEEE Comp. Soc., 2000, 162–171.

[19] Kamp, M., *Managing a Multi-File, Multi-Language Software Repository for Program Comprehension Tools – A Generic Approach*, in: "6th International Workshop on Program Comprehension", IEEE, Comp. Soc., 1998, 64–71.

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

[21] Kazman, R. and J. Carrière, *Playing Detective: Reconstructing Software Architecture from Available Evidence*, Automated Software Engineering, 6, 1999, 107–138.

[22] Kullbach, B., *Command Line GReQL (CLG), Benutzerhandbuch*, Projektbericht 3/2001, Universität Koblenz-Landau, Institut für Softwaretechnik, 2001.

[23] Kullbach, B. and V. Riediger, *Folding: An Approach to Enable Program Understanding of Preprocessed Languages*, in: "8th Working Conference on Reverse Engineering", IEEE Comp. Soc., 2001, 3–12.

[24] Kullbach, B. and G. Schmitz, *Dokumentation des Ada-Parsers für GUPRO*, Projektbericht 9/01, Universität Koblenz-Landau, Institut für Softwaretechnik, 2001.

[25] Kullbach, B. and A. Winter, *Querying as an Enabling Technology in Software Reengineering*, in: "3rd European Conference on Software Maintenance and Reengineering", IEEE Comp. Soc., 1999, 42–50.

[26] Kullbach, B., A. Winter, P. Dahm and J. Ebert, *Program Comprehension in Multi-Language Systems*, in: "5th Working Conference on Reverse Engineering," IEEE Comp. Soc., 1998, 135–143.

[27] Lange, C., H. Sneed and A. Winter, *Comparing Graph-based Program Comprehension Tools to Relational Database-based Tools*, in: "9th International Workshop on Program Comprehension", IEEE, Comp. Soc., 2001, 209–218.

[28] Lientz, B. P. and E. B. Swanson, "Software Maintenance Management, A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations", Addison-Wesley, 1980.

[29] Moad, J., *Maintaining the Competitive Edge*, Datamation 36 1990, 61–66.

[30] Pigoski, T. M., "Practical Software Maintenance, Best Practices for Managing Your Software Investments", Wiley, 1996.

[31] Riediger, V., *Analyzing XFIG with GUPRO*, in: "7th Working Conference on Reverse Engineering", IEEE Comp. Soc., 2000, 194–196.

[32] Riediger, V., *The GUPRO C Parser*, Projektbericht 5/01, Universität Koblenz-Landau, Institut für Softwaretechnik, 2001.

[33] *SWAG Software Toolkit*, http://www.swag.uwaterloo.ca/swagkit/.

[34] Widmann, F., *Entwicklung von Batch-Jobs bei der Debeka*, Projektbericht 6/01, Universität Koblenz-Landau, Institut für Softwaretechnik, 2001.

[35] Winter, A., *Exchanging Graphs with GXL*, in: Mutzel, P., M. Jünger, and S. Leipert (eds.), "Graph Drawing, 9th International Symposium", LNCS 2265, Springer, 2001, 485–500.

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