Analyzing XFIG with GUPRO

Volker Riediger
University of Koblenz–Landau
Institute for Computer Science
Rheinau 1, D–56075 Koblenz, Germany
riediger@uni–koblenz.de

Abstract

GUPRO (Generic Unit for Program Understanding, http://www.gupro.de) provides an adaptable and extesible workbench for program understanding and software reengineering. GUPRO is strongly based on graph technology, i. e. source code is parsed into graph structures which are accessible by graph algorithms and a general graph query language GReQL. In GUPRO, these base technologies are combined into reengineering tools like source code browsers, cross referencers or slicers. This paper describes the application of GUPRO on the 'XFIG Bake–Off', a reverse engineering problem for a tool demonstration at the Working Conference on Reverse Engineering WCRE 2000.

1. Introduction

GUPRO (Generic Unit for PROgram understanding) [3] is a generic toolset for understanding software systems. The GUPRO approach to program comprehension focuses on querying a graph repository [4]. Program comprehension is supported by an iterated three step process of repository queries, report generation and source code browsing. Query results are presented in various formats, where hyperlinks provide connectivity between the conceptual entities of the query result with relevant source code regions. Queries generate an initial set of bookmarks for concept driven source code navigation. Results from previous 'round trips' can be used as input for subsequent queries.

Software of different source and description languages is represented uniformly in a graph based repository. By adapting the conceptual model, the maintenance engineer can focus on arbitrary properties of a software system at any level of granularity. The relevant concepts are represented in UML–like class diagrams. GUPRO tools are parameterized by graph classes defined by the conceptual models. This generic mechanism allows tailoring problem oriented reengineering tools. Conceptual models (including extractors) exist for multi language systems on architectural level [5] and for various single language systems on ASG (Abstract Syntax Graph) level.

GUPRO is strongly based on graph technology. Source code is transformed into a graph structure by specialized parsers. The resulting graph is an instance of a graph class defined by the conceptual model. Software graphs are stored as *TGraph*s. A *TGraph* is a typed attributed directed ordered graph [2]. An efficient implementation of *TGraph* structures and algorithms is given by the *GraLab* library [1]. The software graph is accessible by graph algorithms and a general graph query language *GReQL* (Gupro REpository Query Language [4]). In GUPRO, these base technologies are combined into reengineering tools like source code browsers, inquirers, cross referencers, slicers etc.

GReQL allows the computation of arbitrary reports on software graphs. Graph properties can be used for easy and efficient querying even complex structural relationships between the instances of the modeled software concepts. *GReQL* reports are generated by the command line tool `greqleval` and can be exported in various formats, e.g. plain text tables, CSV files or HTML.

The GUPRO team consisted of the author who did the actual analysis and three students responsible for the necessary tool modifications.

The following sections will introduce the C and CSystem graph classes used as conceptual model throughout the analysis. We present solutions to some of the analysis tasks and give an overview over the principal analysis steps. Finally, we suggest some improvements to GUPRO.

2. Results

2.1. The C/CSystem graph class

In the XFIG analysis, we used a fine granular conceptual model of ANSI–C. The ASG represented by this model is closely related to the concrete syntax tree of the C program. GUPRO was previously used with this model in a

software evaluation project to generate evidence for formal proofs of algorithm properties in single file C programs. In this context, we extended the conceptual model and our C parser to accept Microsoft specific extensions. Unknown language constructs of other C dialects are transformed to opaque syntactic entities in the ASG. Opaque means that the corresponding source code of those constructs is not analyzed for syntactic structure but simply stored as one ASG node.

Static data flow analysis is supported by a SDG (System Dependency Graph) inference tool. The SDG is simply added to the software graph. A slicing tool can generate forward and backward slices of a statement. This is done by *inline GReQL* and native graph algorithms of the *GraLab* library. *inline GReQL* is a function library to embed *GReQL* queries into other programming languages.

2.2. The GOTO query

As an example for the solution of one of the tasks, we present a simple *GReQL* query. This query reports the GOTOs as well as the jump destinations.

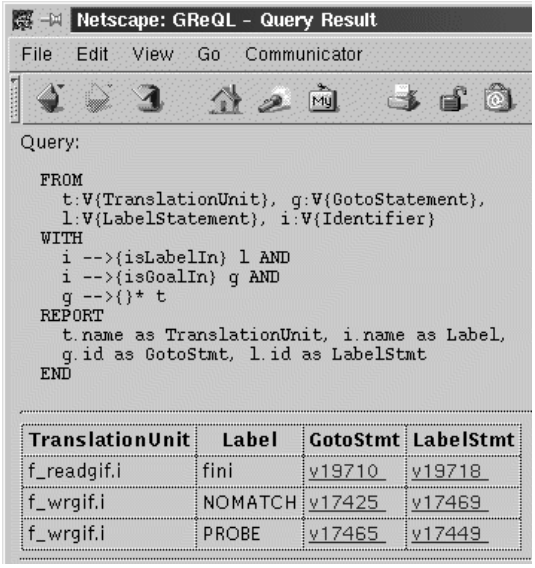

Figure 1. HTML view of the GOTO report

The *GReQL* expression in fig. 1 selects all `GotoStatements` g and uses two path expressions to yield the corresponding label $i \rightarrow \{isLabelIn\}l$ and jump destination $i \rightarrow \{isGoalIn\}g$.

Since the underlying *TGraph* is directed, the arrows in path expressions determine the direction of the edges. The argument in curly brackets can optionally be used to restrict the edge type. The `TranslationUnit` representing a source file is found by the regular expression property of path expressions, specifying an arbitrary path from the `GOTO` to it's source file: $g \rightarrow \{\} * t$.

The `GotoStmt` and `LabelStmt` columns identify the vertices in the ASG representing the respective entities of the underlying conceptual model (see fig. 2).

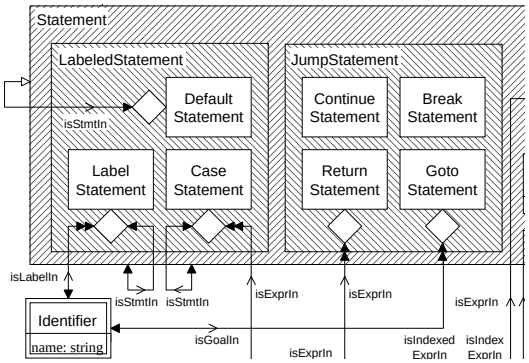

Figure 2. Extract from C/CSystem graph class

Different to other tools, parsing of C in GUPRO is based on *preprocessed* source files. Preprocessing of XFIG was done on a Sun Sparc architecture. This is the reason why the two more `GOTO`s of XFIG in `main.c` are not discovered by the query. The C preprocessor macro `USE_TAB` to include graphics tablet support was not defined in the extraction phase. Accordingly the entities depending on that macro are not represented in the software graph.

2.3. Architecture recovery

A possible solution to architecture recovery is to report function calls inside one module and between modules. This is an example for a *GReQL* query to generate a direct function call report:

```
     FROM caller, callee: V{Identifier}
     WITH
(1)    callee (-->{isFunctionNameIn}
(2)    -->{isExprIn}*
(3)    -->{isStmtIn}*
(4)    -->{isCompoundStatementIn}
(5)    <--{isDirectDeclaratorIn}+) caller
     REPORT
       caller.name, callee.name
     END
```

The path expression in this statement is slightly more complex. The path in the ASG leads (1) from the callee `Identifier` to a `FunctionCall` vertex, (2) to the (zero or more) nested `Expressions` applying the function, over zero or more (3) `Statements` to a `CompoundStatement`. This `CompoundStatement` is the body in a (4) `FunctionDefinition` which finally leads over one or more `isDirectDeclaratorIn` edges (5) to the `Identifier` vertex representing the caller's name.

As mentioned before, this query reports direct function calls. Note that all indirect function calls could be discovered by computing the transitive closure over the given path

from callee to caller. In *GReQL*, this can easily be specified by adding a '+' (after the closing round bracket) to the path between `callee` and `caller`.

To accomplish the architecture task, we had to extend our single file conceptual model to a CSystem graph class. The CSystem ASGs contain software graphs of all source files providing a repository for inter–module queries. This required to augment the conceptual entities representing translation units and source code position references. We did not complete the architecture task due to graph size problems.

3. Experience report

3.1. Parsing XFIG

In order to parse XFIG with GUPRO tools, we had to choose a destination platform and analyze the output of `make`. Preprocessing the source files (2MB) with the macro definitions for the Sun Sparc architecture took about 5 minutes yielding approximately 15MB preprocessor output.

Parsing the preprocessed files with the GUPRO C parser using the C graph class for single source files took another 30 minutes. This resulted in software graphs for each module. Graph sizes are about 21,000 nodes and 30,000 edges per module, adding up to a total of more than 1,440,000 nodes and 2,070,000 egdes (188MB).

3.2. Tool modifications

We had to modify the C parser in two ways: First, it refused to parse some GCC specific constructs. GCC's optimization hints (i. e. `__attribute__` declarations) had to be ignored by the scanner. The second modification addressed the single source file problem. To analyze XFIG, the C graph class had to be extended to the multi file capable CSystem class.

For a preview of first results of these modifications, we used the `g2html` tool to convert graph files into a human readable form. The attempt to browse the HTML version of the graphs proved that current main stream web browsers are incapable to handle large files: Netscape as well as Internet Explorer refused to display the 26MB file and crashed.

A re–link of some GUPRO tools with a newer version of the *GraLab* library resulted in a remarkable speedup of the `greqleval` tool, our command line report generator. Relevant processing time in `greqleval` is consumed by loading the graph files. The re–link reduced the load time for a 3MB graph from about 20 seconds to only 3 seconds.

3.3. Large graph problem

The fine granular conceptual model results in a very large software graph. GUPRO tools currently require the entire graph to be in memory. This results in long startup phase and huge main memory requirements. To address this problem, we are thinking about a *GReQL* server. This server could work on a powerful machine to answer multiple queries requiring only a single graph load phase.

4. Lessons learned

The application of the GUPRO tools requires a detailed knowledge of the underlying conceptual model. Beginners have to take this hurdle before any meaningful queries can be formulated. The C graph class was built with respect to a fine granular representation required by it's first application in a software certification project. A maintenance engineer familiar with the concepts of syntax analysis should be able to understand this model and make effective use of GUPRO in not more than two days.

To improve *GReQL*'s practical use, we would suggest two extensions to the existing function library: A function for regular expression matching on strings and some very simple functions for reporting vertex and edge types of the *TGraph* components.

In short time, we expect to finish a new integrated GUI with a folding source code browser integrated with a *GReQL* query interface.

Our first steps in the XFIG tasks lead to the following understanding: The granularity of the conceptual model MUST match the reverse engineering problem. To discover software architecture and to complete tasks at the level of the this tool demonstration, a far more abstract model is essential. Due to the genericity and scalability of the GUPRO tools, it is very easy to tailor the existing conceptual models or create new models for future challenges.

References

[1] P. Dahm and F. Widmann. Das Graphenlabor, Version 4.2. Fachberichte Informatik 11/98, Universität Koblenz-Landau, Institut für Informatik, Koblenz, 1998.

[2] 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*, volume 903 of *LNCS*, pages 38–50. Springer, Berlin, 1995.

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

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

[5] B. Kullbach, A. Winter, P. Dahm, and J. Ebert. Program Comprehension in Multi-Language Systems. In *Fifth Working Conference on Reverse Engineering*, pages 135–143. IEEE Computer Society, Los Alamitos, 1998.