Volume 3, Issue 7, July 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Source Code to Module Dependency Graph Using Class Dependency Analyzer

B.Rajani, Y. Mohana Roopa
Computer Science & Engineering
Annamacharya Institute of Technology & Science
Tirupathi, India

Abstract--- In this demonstration we will show how our tool source code analysis, can be used to recover the high-level structure of a software system directly from its source code. We accomplish this task by first using a source code analysis system (e.g., CIA, Acacia) to produce a module dependency graph that represents system modules and module-level inter-relationships.

Keywords---source code, software system, module dependency graph, system modules, module-level inter-relationships

I. Introduction

Understanding the intricate relationships that exist between the source code components of a software system can be an arduous task. Frequently, this problem is exacerbated because the design documentation is out of date and the original system architect is no longer available for consultation. While modules do much to improve software development and maintenance, they are insufficient for supporting the design and ongoing maintenance of large systems. Such systems often contain several hundreds of thousands of lines of code that are packaged into a large number of cooperating modules. Fortunately, we often find that these systems are organized into identifiable clusters of modules, called subsystems that collaborate to achieve a higher-level system behavior [1]. Unfortunately, the subsystem structure is not obvious from the source code structure. Our research therefore proposes an automatic technique that creates a hierarchical view of the system organization based solely on the components and relationships that exist in the source code. The first step in our technique is to represent the system modules and the module-level relationships as a module dependency graph. We then use our algorithms to partition the graph in a way that derives the high-level subsystem structure from the component-level relationships that are extracted from the source code. Fully automatic modularization techniques are useful to programmers who lack familiarity with a system. These techniques are also useful to system architects who want to compare documented modularizations with the automatically derived ones, and possibly improve the design by learning from the differences between the modularizations. Figure 1 shows the architecture of our automatic software modularization environment. The first step in the modularization process is to extract the module-level dependencies from the source code and store the resultant information in a database. We used AT&T's CIA tool [2] (for C), Acacia[3] (for C++) and Chava[5] (for JAVA) for this step. After all of the module-level dependencies have been stored in a database, we execute an AWK script to query the database, filter the query results, and produce, as output, a textual representation of the module dependency graph. Our clustering tool, called Bunch, applies our clustering algorithms to the module dependency graph and emits a text-based description of the high-level structure of the systems organization. We then use the AT&T dotty visualization tool[4] to read the output file from our clustering tool and produce a visualization of the results. The structure of the remainder of this paper is as follows: in Section 2, we provide an overview of related work, in Section 3, we describe the reverse engineering services, , in section 4,we explain the code querying services, in section 5, we conclude by summarizing the paper and outlining the contributions of our work.

II. Background

The project is closely related to work in the areas of code analysis. As many software engineering tools operate on graphs or produce graphs as output (i.e., Module Dependency Graph).

A. Code Analysis

There has been considerable progress in code analysis tools for C, C++, Java, and COBOL. Code analysis tools parse source or intermediate (e.g., byte code) code and produce a database of code entities (e.g., functions and variables) and relationships (e.g., method invocation and object instantiation). When trying to understand the code of a program, developers formulate queries against the code database such as:

(a) What is the inheritance hierarchy rooted at class C?
(b) (b) Is function f1 reachable from function f2 through a sequence of calls?
(c) What are the signatures of the constructors of class C?

The results of such queries are typically displayed as simple text or as a graph using visualization tools such as dot [6, 7]. Two code analyzers have been developed at AT&T: Acacia [10] for C/C++ and Chava [5] for Java (both tools are available on-line [8]). These tools are described in more detail in Section 3. Other examples of code analyzers are Reprise [9] for C++ and Cobol/SRE [10] for COBOL. In addition to code analysis tools, there has been work in meta-tool code analysis generators such as Aria [11] and Genoa [12].

Early research on code analysis has influenced the more current advances in commercial tools. Red Hat's Source-Navigator [13] is a code analysis tool that supports C, C++, Java, Tcl, FORTRAN and COBOL. Using this tool one can display relationships between classes, functions, and members, find every place in the code where a given function is called, find each file that includes a given header file, display call trees, and so on.

Netcomputing's AnyJ [14] is a Java IDE and code analysis tool. In addition to the typical tools offered by an IDE (e.g., editor, debugger), AnyJ includes code browsing and analysis tools that operate on .zip, .class, .jar, or .java files For example, AnyJ allows programmers to jump to the definition of methods/variables with respect to the type of the current expression and class hierarchy. Western Ware's CC-Rider [15] is a code analysis and browsing tool to visualize and document software written in C and C++. CC-Rider's visual browsing supports a variety of Tree Views such as: class hierarchy, class ancestry, class nesting, function call, data usage, and file usage. Similar to CC-Rider supports code navigation using Hypertext links.

III. Code Analysis Services

At the core of the reverse engineering services is a database that captures a software engineering view of a program or system. The schema for this database supports a representation of programs as a collection of software entities, such as files, functions, types and variables, and relations among pairs of entities, such as "function a" calls "function b", or "class D" is a subclass of "class C". At present, reverse engineering of a software system is focused on the analysis of C/C++ and Java code. It provides two analysis engines, one for each language family, that can analyze a program and generate a database with the appropriate information, and in the supported format.

A. C/C++ Code Analysis Services

By default, REportal uses Acacia for the analysis of C and C++ programs. Though more fully described elsewhere [3], we give a brief description of its features and structure. Acacia mimics the typical C compiler interface, in that one can build a database by replacing every use of the C or C++ compiler by a call to Acacia. In particular, it accepts the conventional compiler flags, and is typically used to process a collection of source files, followed by a final pass that links the individual databases into a single program database, analogous to the linking of a collection of object files into a single executable program.

The analysis accepts source code written in ISO C/C++. An initial preprocessor pass is performed, allowing Acacia to catch macro definitions and uses. This is important, as the database is meant to give a source level view of the program. Acacia then performs parsing and type checking, which results in a annotated parse tree. From the parse tree and related tables, it extracts the entities and relations.

One of the difficulties in the analysis of C or C++ is the non-portability inherent in any significant program, despite international standardization. This is in distinction to Java code, which, at present, is portable, at least at the level of compilation. The problems in the C family tend to arise from three causes.

1. There are many flavors of C and C++. Despite the existence of standards, some compiler providers have yet to implement them, sometimes due to the complexity of the standards themselves. In other cases, compiler providers have intentionally introduced new, non-standard C or C++ syntax. These additions to the syntax occur in such popular compilers as *gcc* and *Visual C++*. In either case, a programmer will often use the constructs without even realizing they are not standards-compliant. In a bow to the popularity of *gcc*, Acacia includes various fixes to finesse some of the more frequent non-standard constructs.
2. In the C family, much of the system level functionality is not standardized, especially as regards implementation and the use of include files. This means that it is easy to write code that compiles on one system but not on another. At best, even if the programmer makes the code portable by the disciplined use of compiler directives (e.g., #ifdef, etc.), the resulting databases will vary greatly from system to system, especially concerning the low-level, system entities.
3. The build process typically relies on some version of the Unix *make* tool, which in turn often relies on various other programs used to configure the build, or generate code and data. This further magnifies the portability problem, and can make the automatic generation of a software system database difficult.

We hope that, as reverse engineering of a software system develops, it will be able to support multiple platforms more cleanly, either by providing the necessary include files and Acacia databases of system libraries, or by including different platforms as part of the service.

B. Java Code Analysis Services

Chava [5] is used for the analysis of Java programs. Chava extracts information from Java applets or applications about classes, methods, fields and their relationships. This information is then stored in a CIAO relational database. Given our Java data model, we can query, visualize, and analyze the structural information with the language independent CIAO tools.

Chava is able to process either Java source files or compiled class files, making it possible to analyze remote applets whose source code is unavailable. Analysis using only class files is possible primarily due to properties of the Java language. Java's absence of a preprocessor means that we do not have to deal with constructs such as macros, include files, and templates, whose information would not be available in an object file. Also, Java is an architecture-neutral language, so it is possible to scan through object code in a machine-independent manner to discover relationships in a program. Performance numbers indicate that the tool scales well running times indicate that Chava runs an order of magnitude faster than javac (the Java compiler). In fact, Chava is also faster than javap, the Java program that dumps the contents of a class file. The generated database size is in the order of the size of the class files, which is quite manageable. Size could be reduced significantly if compression is used on the database. The number of entities and relationships is small enough that queries can be performed efficiently.

IV. Code Querying Services

Users to perform customized entity or relationship queries that explore various structural aspects of the C, C++, or Java program being analyzed. An entity query allows a user to select database records based on attribute values specified only on the left column in the query table. A relationship query examines relationships among entities by specifying attribute values of the source entity (left column) and the sink entity (right column) of a relationship. The result can be presented in database mode, graph mode (laying out the relationships as a graph), or text mode (show the source lines where those references occur). Presentation is accomplished by piping the intermediate database to the corresponding presentation tool.

V. Visualization Service

Software systems are typically modified in order to extend or alter their functionality, improve their performance, port them to different platforms, and so on. For developers, it is crucial to understand the high level design of a system before attempting to modify it. However, the high-level design may not be apparent to new developers, because the design documentation is non-existent or, worse, inconsistent with the implementation. This problem could be alleviated, to some extent, if developers were able to produce design descriptions from the low-level source code.

The first step in the design recovery process is to extract the module-level dependencies from the source code and store the resultant information in a database. Software system uses the Acacia[3] or Chava [5] tools for this step. After all of the module-level dependencies have been stored in a database, software system executes a script to query the database, filter the query results, and produce a textual representation of the module dependency graph (MDG).

For Example:

Various attributes can be applied to graphs, nodes and edges in DOT files. These attributes can control aspects such as color, shape, and line styles. For nodes and edges, one or more *attribute-value pairs* are placed in square brackets ([]) after a statement and before the semicolon. Graph attribute are specified as direct *attribute-value pairs* under graph element. Multiple attributes are separated by a comma and a space. Node attributes are placed after a statement containing only the name of the node, and no relations.

graph graphname

```
{    // This attributes applies to the graph itself
     Size="1.1";
     //The label attributes can be used to change the
       label of a node
     a[label=Foo];
     // Here, the nodes shape is changed
     b[shape=box];
     //These edges both have different lines properties
     a — b — c[color=blue];
     b – d[style=dotted];
}
```

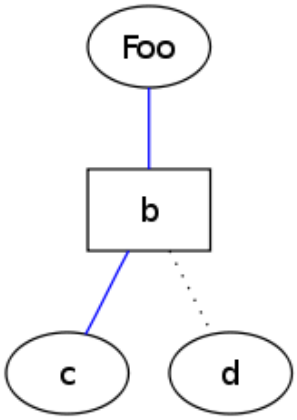

Fig 1: Graph with Attributes

In Figure 2 we show the results of applying our tools to the source code of dotty . The model we employed to create the MDG of dotty uses files as modules, and defines a directed edge between two files if the code in one file refers to a type, variable, function or macro defined in the other file. Subsystems are shown as rectangles containing the MDG nodes and edges.

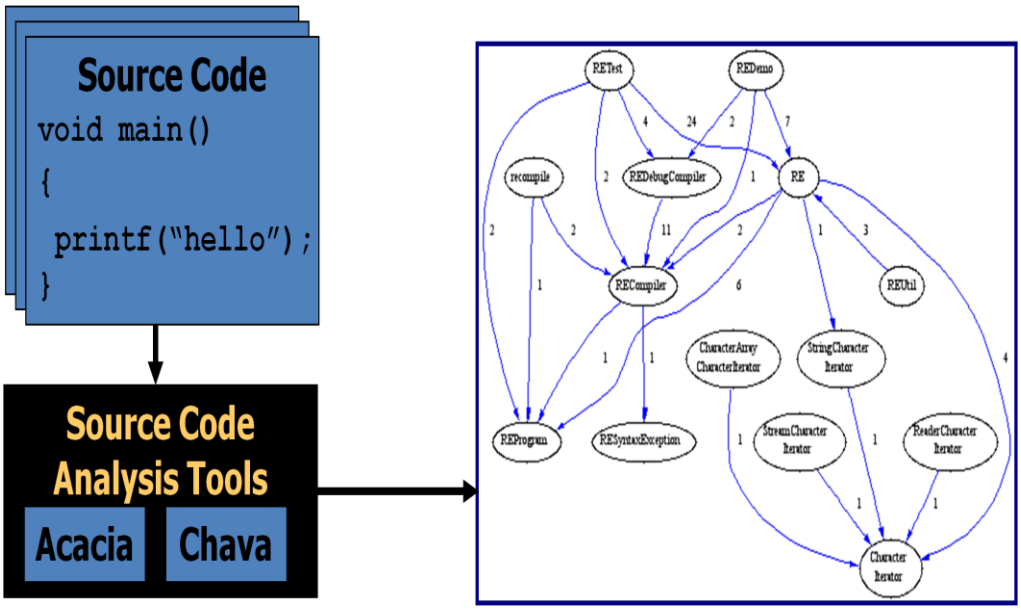

Fig 2: Source Code to MDG Using Code Analyzer

VI. Conclusion

In this paper, we conclude that reverse engineering of software system is possible with different types of source code analyzer tools or class dependency analyzer tools. In the future we extend the paper to find source code analyzer for different upcoming programming languages and produce a textual representation of the module dependency graph (MDG).

References

[1] F. DeRemer and H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. IEEE Transactions on Software Engineering, pages 80{86, June 1976.

[2] Y. Chen. Reverse engineering. In B. Krishnamurthy, editor, Practical Reusable UNIX Software, chapter 6, pages 177{208. John Wiley & Sons, New York, 1995.

[3] Y. Chen, E. Gansner, and E. Koutso os. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering, Sept. 1997.

[4] S. North and E. Koutso os. Applications of graph visualization. In Proc. Graphics Interface, pages 235{245, 1994.

[5] J. Korn, Y. Chen, and E. Koutsofios. Chava: Reverse Engineering and Tracking of Java Applets. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 314–325, 1999.

[6] E. Gansner, E. Koutsofios, S. North, and K. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, Mar. 1993.

[7] S. North and E. Koutsofios. Applications of graph visualization. In *Proc. Graphics Interface*, 1994.

[8] AT&T Labs - Research Tools http://www.research. att.com/sw/tools.

[9] The Western Wares CC-Rider Source Code Analyzer and Browser. http://www.westernwares.com/.

[10] Y. Chen, E. R. Gansner, and E. Koutsofios. A C++ data model supporting reachability analysis and dead code detection. In *Proceedings of the European Conference on Software Engineering/Foundations of Software Engineering*, 1997.

[11] P. Devanbu, D. Rosenblum, and A. Wolf. Generating Testing and Analysis Tools with Aria. *ACM Trans. Software Engineering and Methodology*, 5(1):42–62, 1996.

[12] P. Devanbu. Genoa—a language and front-end independent source code analyzer generator. In *Proceedings of the Fourteenth ICSE*, pages 307–317, 1992.

[13] The Red Hat Source-Navigator. http://sources. redhat.com/sourcenav/.

[14] The Netcomputing AnyJ Java IDE. http://www. netcomputing.de/html/main.html.

[15] The Western Wares CC-Rider Source Code Analyzer and Browser. http://www.westernwares.com/.

Authors Biography

Author 1: B. Rajani received the B.Tech Computer Science and Engineering from JNTUA, Anantapur, India in 2009 and pursuing her Master's degree in Computer science and Engineering from the JNTUA, Anantapur, India. She has published one international conference. Her research areas are Software Engineering, Data Warehousing and Data Mining, and Database Management Systems.

Author 2:Y. Mohana Roopa working as Associate Professor & HOD of Computer Science and Engineering in Annamacharya Institute of Technology and Sciences, Tirupati, A.P., India. She has received M.Tech Degree from Jawaharlal Nehru Technological University (JNTU), Kakinada, A.P., India. She has received B.Tech Degree from KSRM College of engineering, kadapa, which is affiliated to Sri Venkateswara University, Tirupati. She published papers in various journals. Her area of interest is Software Architecture, Component Based Development, and Swarm Intelligence.

