ADG: Annotated Dependency Graphs for Software Understanding

Ahmed E. Hassan and Richard C. Holt
Software Architecture Group (SWAG)
School of Computer Science
University of Waterloo
Waterloo, Canada
{aeehassa,holt}@plg.uwaterloo.ca

ABSTRACT
Dependency graphs such as call and data usage graphs are often used to study software systems and perform impact analysis during maintenance activities. These graphs show the present structure of the software system ($e.g.$ In a compiler, an $Optimizer$ function calling a $Parser$ function). They fail to reveal details about the structure of the system that are needed to gain a better understanding. For example, traditional call graphs cannot give the rationale behind an $Optimizer$ function calling $Parser$ function.

In this position paper, we advocate a new view on dependency graphs – Annotated Dependency Graphs (ADG). ADG can assist maintainers understand better the current structure of large software systems. We show an example of using an ADG to study $Postgres$, a large DBMS open source software system.

1 INTRODUCTION

To aid in software understanding tasks, documentation is used to narrate different aspects in the life cycle of a software system. Unfortunately software developers are not interested in documenting their work. Documentation rarely exists. If it does it is usually incomplete, inaccurate, and out of date. Faced with the lack of sufficient documentation, developers choose alternative understanding strategies such as searching or browsing the source code. The source code in many cases represents the definitive source of accurate information about the system [11]. Developers search the code using tools such as `grep`. They browse the code using simple text editors or cross-reference code browsers such as `LXR`, which permit jumping between variables/functions usage and variables/functions declarations while browsing the source files.

Dependency graphs have been proposed and used in many studies and maintenance activities to assist developers in understanding large software systems before they embark on modifying them to meet new requirements or to repair faults. Call graphs and data usage graphs are the most commonly used dependency graphs.

The rationale behind the existence of dependencies between two nodes in a dependency graph are usually based on domain and system knowledge. For example, based on our knowledge of the reference architecture of a compiler, we can reason about the rationale behind the dependencies shown in the graphs [10]. For domains that are not well understood that may not be clear and may prove to be a challenging and daunting task. Moreover, for well understood architectures such as compilers, we may find unexpected dependencies that indicate, for example, that an $Optimizer$ function depends on a $Parser$ function. As a maintainer of such a system, the rationale behind such unexpected dependency is not clear - Are there valid reasons for such dependency? Or was it due to laziness or ignorance of the developer that introduced the dependency?

Much of the knowledge about the design of a system, its major changes over the years and its troublesome subsystems live only in the brains of its developers. Such live knowledge is sometimes called *wet-ware*. When new developers join a team, mentoring by senior members and informal interviews are used to give them a better understanding of the system. Leveraging this knowledge may not always be possible as the software may have been bought from another company, its maintenance outsourced, or its senior developers are no longer part of the company. Thus, answering questions about unexpected dependencies and other discoveries as developers study dependency graphs becomes a challenging and time consuming task. Traditional dependency graph are only capable of giving us a current view of the software system without details about the rationale, the history, or the individuals behind the dependency relations.

In this paper, we propose to extend dependency graphs – Annotated Dependency Graphs (ADG) – to attach more details, in an attempt to assist developers in understanding and studying software systems. In an ideal world, if each developer attached a sticky note to each added dependency to record their name, the rationale behind the addition or removal of the dependency then the job of the maintainer will be much easier. In the fast paced world of software development with tight schedules and short time to market, this is neither possible nor practical. Thus in addition to proposing these extended dependency graphs, we present a technique to build such graphs automatically without any input from the developers of the system.

Organization of Paper
The paper is organized as follows. Section 2 highlights sev-

eral problems associated with traditional dependency graphs and proposes Annotated Dependency Graphs (ADG) to address these shortcomings. Section 3 gives an overview of how to build an ADG. Section 4 presents a short case study of an ADG for *Postgres*, a large open source database management system (DBMS). Section 5 describes related work. Finally section 6 draws conclusions from our work and proposes future directions.

2 SHORTCOMINGS OF DEPENDENCY GRAPHS

As maintainers prepare to modify a software system to add features or repair bugs, they start off by examining any available documentation, and consulting senior developers. Then they browse the source code and use tools to generate dependency graphs such as call and data usage graphs. Following these steps in an iterative manner, maintainers start gaining a better understanding of the software system. Using their newly acquired understanding, they form an internal cognitive model of the software [12]. Dependency graphs assist maintainers in gradually piecing together the software puzzle. Unfortunately, dependency graphs fall short in the following areas:

1. Rationale: They do not indicate the reasons behind the introduction or removal of dependencies between source code entities.

2. Time: They do not indicate how long a dependency has existed for or how long ago has it been removed.

3. Inter-dependency patterns: They fail to show patterns of dependencies. For example, in a compiler system it is hard to deduce from a call graph that once a function depends on the $populate\_symbol\_table()$ function, it will also depend on $symbol\_table\_entry$ data type.

4. Creator: They fail to show the name of the developer that introduced the dependency.

In the following subsections, we elaborate on the benefits of each area for software understanding.

Rationale

In a compiler, when a dependency between an $Optimizer$ function and a $Parser$ function is discovered, the maintainer puzzled by the unexpected dependency can contact the senior developer to get a better understanding of the rationale behind the introduction of such a dependency. The senior developer may be too busy or may not recall the rationale. Furthermore the developer who introduced the dependency may no longer work on the software system. Then the maintainer has to examine the source code closer and spend hours trying to understand the rationale behind such unexpected dependency. In some cases the added dependency may be justified due to, for example, optimizations or code reuse; or not justified due to, developer ignorance, or pressure to market.

Another popular usage of dependency graphs is the discovery of dead code. Dead code is code which no other entities in the software system depends on. Again dependency graphs are able to locate the dead code such as unused functions and unused data types, but fail to indicate the reason for the death of the code. A maintainer would like to know if the dead code has been replaced by an optimized or better piece of code capable of performing the same functionality, replaced by a more general piece of code that encourages more reuse, or even decommissioned as the system no longer supports the functionality offered by the dead code.

These two aforementioned examples are some of the many situations where the $Rationale$ for the appearance or disappearance of dependencies is essential in aiding maintainers of large software systems. Unfortunately, attaching the $Rationale$ for each dependency would require many resources and is time consuming. Consequently an automated technique to annotate each dependency with its rationale would be very beneficial.

Time

Current dependency graphs only provide a single current view of the software system. As pointed out in the previous section this prevents the dependency graph from helping developers understand the evolution of dependencies in the software system. Determining the evolution of dependencies for a dead piece of code is a good example. For example, the developer may want to know whether the dead function was introduced in the previous release to go around a bug and the bug is now fixed yet the functions isn't removed or whether the dead function has been around for many releases.

Inter-dependency patterns

The ability to determine that adding a dependency to one entity entails adding dependencies to other entities is a very valuable information that can be used in building tools to assist developers in maintaining large complex software systems. This knowledge would help developers guide developers to other entities in the source code that may require modifications.

Creator

In some cases, the creator of a dependency may pose a concern. The creator is a good indicator of the validity and trustworthiness of unexpected dependencies. For example an unexpected dependency created by a developer that was junior when the dependency was created is a strong sign that the dependency may be an invalid one.

3 BUILDING ANNOTATED DEPENDENCY GRAPHS

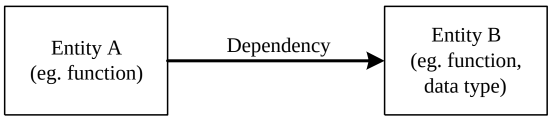

Figure 1: Schema for a Traditional Dependency Graph

To overcome the shortcomings highlighted in the previous sections, we propose Annotated Dependency graph (ADG). Whereas a traditional dependency graph would consist of entities and edges, as shown in Figure 1, an ADG would have several attributes attached to the nodes and the edges. Figure 2 shows the attributes that are attached. These attributes address shortcomings of traditional dependency graphs.

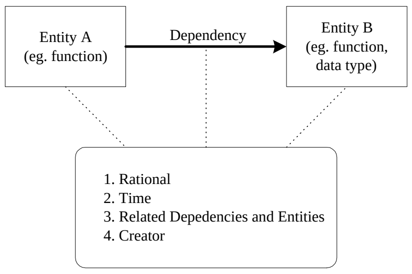

Figure 2: Schema for a Annotated Dependency Graph

Although, The ADG can be created manually by developers as they update the source code, this is not a practical solution for the following reasons:

1. It requires developers of large established software projects to go through a traditional dependency graph of the software system and try to populate the ADG. This is a time consuming and erroneous task. In many cases the developer may no longer recall the reasons for the dependencies and in most cases won't recall the details for the other attributes in an ADG.

2. Another alternative would be to use the ADG at the start of a new project and make sure developers always annotate any new or removed dependencies. Again this is extra work which most developers would not be interested in doing.

Instead of building the ADG manually, we chose to use the change records stored in the source control repository such as RCS [13] or CVS [3, 5].The repository contains details about the development history of each file in the software system. The repository stores the creation date of the file, its initial content and a record of each modification done to the file. A *modification record* stores the date of the modification, the name of the developer that performed the changes, the line numbers that were changed, the actual lines of code that were added or removed, and a detailed message entered by the developer explaining the rationale behind the change.

Source control systems store the details of the modification at the line level of a file which is not sufficient to build the ADG which has as functions and data types as nodes. Thus, we first need to preprocess the modification records to map the changes to the appropriate source code entities (*i.e.* functions or data types). Then we build the Annotated Dependency Graph and annotate it with details from the modification records such as time, and rationale. Due to size limitations, we will not discuss the details of the ADG building as it is detailed elsewhere [7].

4 CASE STUDY

To validate the usefulness of our approach we show a small case study on *Postgres*. *Postgres* is a sophisticated open-source Object-Relational DBMS supporting almost all SQL constructs. Its development started in 1986 at the University of California at Berkeley as a research prototype. Since then it has become an open source software with a globally distributed development team. It is being developed by a community of companies and people co-operating to drive the development of one the world's most advanced Open Source database software (DBMS).

In our case study we build an ADG using data beginning with 1996 when Postgres became an open source project. Building the ADG for Postgres requires 2 hours and 30 mins on a 1.8 Pentium 4 CPU. Luckily building the ADG needs to be done only once, then we can use it to generate graphs as required. All graphs shown in this section have been generated in under 5 seconds once the ADG was generated.

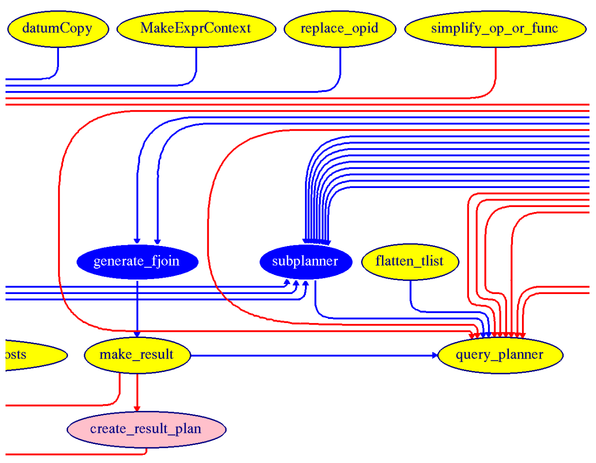

Figure 4: Zoomed-In Low-Level (*Function*) Call Graph for Postgres Optimizations

Using an ADG, we can generate a call graph similar to traditional call graphs generated by parsing the latest source code. Instead we chose to generate a more interesting call graph that showcases the benefits of using an ADG. Figure 3 shows an example of such graph. The displayed call graph shows changes to the call graph that

- occurred in the last month,
- and which were due to optimizations work done to speed up the database.

Each oval represents a function. Blue ovals indicate functions that have been removed from the source code, pink ovals indicate functions that have been added to the source code, and yellow ovals indicate functions that been modi-

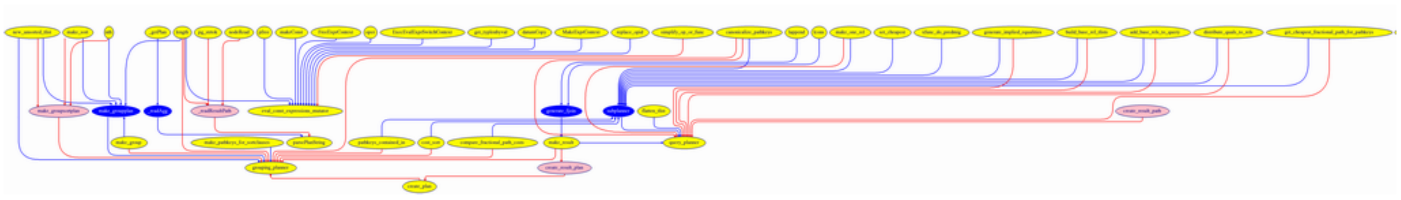

Figure 3: Low-Level ($Function$) Call Graph for Postgres Optimizations

fied in the last month. Also, blue arrows indicate function calls that have been removed and red arrows indicate function calls that have been added in the last month.

The graph shown in Figure 3 is too small to distinguish its various details. In Figure 4, we show a small zoomed-in section of the larger graph to give a better idea of the changes. A developer visualizing the call graph can focus on their area of interest using their visualization software.

Alternatively, we can lift the details of the graph from the function level to the subsystem level [6]. For each function in the call graph we locate the file that defines it, then the subdirectory in which that file resides. We use the subdirectory as the node in that lifted visualization instead of the function. The lifted visualization is shown in Figure 5 where each node represents a subdirectory. The new visualization is much clearer but does not have as much details. If more details are required then the function level visualization can be used.

5 RELATED WORK

The work presented in this paper addresses two main streams of research in particular, visualizing the evolution of source code and locating features in the source code.

Visualizating Evolution

In [8] and [14], two approaches are presented to tackle the issue of visualizing software structural changes. Both approaches are based on studying the changes between releases on a software system instead of basing their study on changes that occurred in the source control system, which are more granular and more detailed. They are only capable of comparing changes from one release to the next instead of being able to compare on a calendar basis or even comparing changes relative to other changes (such as examining changes that occurred before or after a particular change). Furthermore, they do not provide techniques to filter the changes and categorize them according to their rationale.

Locating Feature in Source Code

Chen $et\ al.$ have shown that comments associated with source code modifications provide a rich and accurate indexing for source code when developers need to locate source code lines associated with a particular feature [2]. We extend their approach and map changes at the source line level to changes in the source code entities, such as functions and data structures. Furthermore, we map the changes to dependencies between the source code entities.

Murphy $et\ al.$ argued the need to attach design rationale and concerns to the source code [1, 9]. They presented approaches and tools to assist developers in specifying and attaching rationale to the appropriate source code entities. The processes specified in their work is a manual and labor intensive process, whereas our approach uses the source code comments and source control modification comments to automatically build a similar structure to assist developers in maintaining large code bases. Moreover as our approach is automated, developers do not need to worry about maintaining the outcome of the process in addition to maintaining their source code, a challenge faced by the aforementioned processes.

Finally, Eisenbarth $et\ al.$ presented an approach to locate features in the source code based on the integration of static and dynamic dependencies graphs [4]. Their approach uses a set of test cases to exercise features in the source code, then the static and dynamic call graphs for the specific test are analyzed to locate areas in the code that implement the feature. Whereas their approach uses a combination of static and dynamic source code and a suite of test cases, we only use the source code and the source code repository to locate features. Our approach will only locate features if they were specified in some comment in the source code or the source control system throughout the development history of the project. Thus in some cases using a dynamic analysis such as proposed by Eisenbarth $et\ al.$ will be of great value and will complement our work.

6 CONCLUSIONS AND FUTURE WORK

In this paper we presented a new view on dependency graphs – Annotated Dependency Graphs (ADG). ADG represents a family of graphs which can assist maintainers as they work on gaining a better understanding of large software system. An ADG provides maintainers with the ability to study dependencies between the software entities and limit the dependencies to various criteria such as to a specific period in time, a specific change reasons, or even a specific developer.

In future work, we plan on using ADG to extract aspects from the source code and to build wizards that assist developers by suggesting source code entities that need to be modified once an entity has been modified based on the co-modification history stored in the ADG.

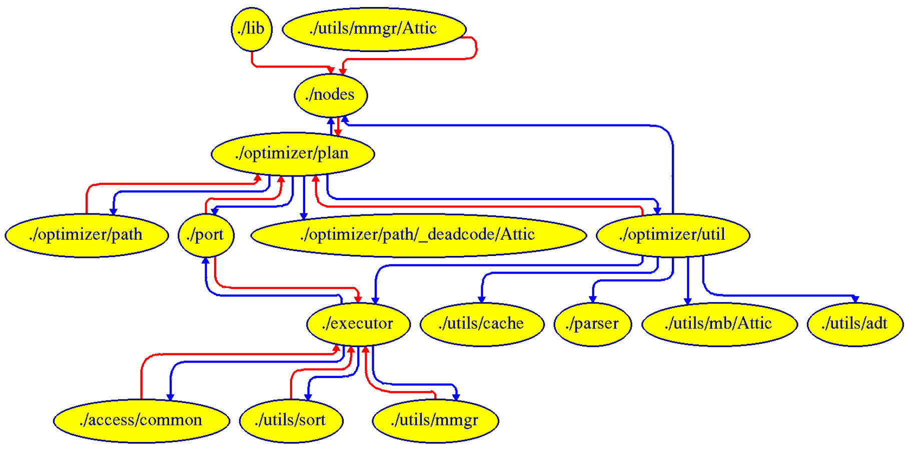

Figure 5: High-Level (*Subsystem*) Call Graph for Postgres Optimizations

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the significant contributions from the members of the open source community who have given freely of their time to produce large software systems with rich and detailed source code repositories; and who assisted us in understanding and acquiring these valuable repositories.

The figures shown in this paper were generated using the aiSee graph layout software.

REFERENCES

[1] E. L. Baniassad, G. C. Murphy, and C. Schwanninger. Design Pattern Rationale Graphs: Linking Design to Source. In *IEEE 25th International Conference on Software Engineering*, Portland, Oregon, USA, May 2003.

[2] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: Searching through Source Code Using CVS Comments. In *IEEE International Conference Software Maintenance (ICSM 2001)*, pages 364–374, Florence, Italy, 2001.

[3] CVS - Concurrent Versions System. Available online at `<http://www.cvshome.org>`

[4] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. *IEEE Transactions Software Engineering*, 29(3):195–209, Mar. 2003.

[5] K. Fogel. *Open Source Development with CVS*. Coriolos Open Press, Scottsdale, AZ, 1999.

[6] A. E. Hassan and R. C. Holt. Architecture Recovery of Web Applications. In *IEEE 24th International Conference on Software Engineering*, Orlando, Florida, USA, May 2002.

[7] A. E. Hassan and R. C. Holt. Understanding Change Propagation in Software Systems. In *Submitted for Publication*, 2003.

[8] R. C. Holt and J. Y. Pak. GASE: visualizing software evolution-in-the-large. In *Working Conference on Reverse Engineering*, pages 163–, 1996.

[9] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *IEEE 24th International Conference on Software Engineering*, Orlando, Florida, USA, May 2002.

[10] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ., USA, 1996.

[11] S. E. Sim. Supporting Multiple Program Comprehension Strategies During Software Maintenance. Master's thesis, University of Toronto, 1998. Available online at `<http://www.cs.utoronto.ca/˜simsuz/msc.html>`

[12] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. In *International Workshop on Program Comprehension*, pages 17–28, 1997.

[13] W. F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.

[14] Q. Tu and M. Godfrey. An Integrated Approach for Studying Software Architectural Evolution. In *Workshop on Program Comprehension (IWPC2002)*, Paris, France, June 2002.