Usage of Call Graph for Representing Software Component Interactions

Nor Laily Hashim[1], and Noraini Ismail[2]

[1]*School of Computing, Universiti Utara Malaysia, laily@uum.edu.my*
[1]*School of Computing, Universiti Utara Malaysia, mail.noraini@gmail.com*

ABSTRACT

A call graph is a ubiquitous representation in most aspect in software engineering. This paper presents an initial proposed technique to represent components relationships in the form of a call graph. To support this study, this paper will cover types of component, a technique used to extract information of component integration, and a process of constructing a call graph, in order to represents the relationship of the component in the software.

Keywords: component, call graph representation, static analysis technique.

I INTRODUCTION

A software component can be a single part of software that can be integrated with each other. Two components are integrated if they can potentially react to the same events (Fiege, 2005), which is by passing messages through their interfaces when the components provide or require for specific events (Inverardi & Tivoli, 2003). The communication between components is typically realized by procedure calls or any kind of messaging.

When new components are integrated, the newly added component has an impact to another component, and it can also be used by other components. Due to this situation, the program may crash or immediately stop the execution of the system. For this reason, a programmer must scan through the program and investigate which components are causing the errors.

To show the flow of the system, the programmers need to refer to the software program to check the program line of code. This task will become more complex and time consuming when it requires scanning through the line of codes as it requires knowledge of the developers to handle this problem.

To assist the programmer to overcome this difficulty, this research presents a call graph to display the information flow of the program without referring to the line of codes.

The organization of this paper is structured as follows. Section 2 presents a brief about software component which is applied in this study. Section 3 explain other technique that to represent software component. Section 4 explained the detail of call graph representation. Section 5 proposes the process of constructing call graph to represent component relation, and finally Section 6 contains the conclusion.

II SOFTWARE COMPONENT

This section covers software components which is related concepts of this work that includes the definition of component, component integration and the techniques to extract the component integration

There have been many discussions about component specification. The common definition for component has been stated by (Briand et al., 2006 and Wu & Woodside, 2004), is the most used today is as follows:

"A unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties".

The two components are related if they can potentially react to the same events (Fiege, 2005), which is by passing the message through its interface when component provides or requires specify events (Inverardi and Tivoli, 2003). The communication between components is typically realized by procedure calls or any kind of messaging.

Component can be a single part of software that can be integrated to each others. Reekie, & Lee (2002) defined software components as “binary units of independent production, acquisition, and deployment that interact to form a functioning system". He also clarifies that “binary" means any format that can be executed by a target machine. This may be serial coded for a specific processor, or virtual machine code, or in some cases even source code (as in some scripting languages).

A. **Component Integration**

Software component integration is the one of the main problems in the development of software system. It can be approached from different views including infrastructures and characteristics of individual components which might support integrations (Rader, 1997).

When designing integrated systems, components are required to refer to other components using simple object oriented techniques to create an interaction between components. To detect the most wanted behaviors, components will need to call up each other. When the interactions succeed in the dependence between components, it results in coupling which prevents separate compilation of integrated component (Rajan & Sullivan, 2005).

The main purpose of integration is to ensure the interactions between their environment and components are properly working. The integration of system must be assessed on the final platform, either when the system is modified or system is starting (Piel & Gonzalez-Sanchez, 2009). For this study, the integration between components is very important to ensure that call graph will be created correctly in order to identify the relationship of the components in the system.

B. **Technique to Extract Component Integration**

This section explains the techniques used in extracting software component. There are two types of extraction techniques: static analysis and dynamic analysis. Static analysis is a method of a computer program debugging that is conducted by examining the code without executing the program Dynamic analysis is conducted by examining the code when the program is executed. Both techniques provide an understanding of the code structure, and can assist to ensure that the code adhere to industry standards (Bergeron et al., 2001). In this study, static analysis is used to extract the component from the source code to construct the call graph, as time is not important factor to consider in this study.

By extracting component interaction information using static analysis technique, the call graph can be constructed from the source code of the program. However, discovering the static call graph from the source code would involve two steps: (1) finding the source code of the program (which may sometime not be available), (2) scanning and parsing of the code, which may be written in several languages. But in some condition where source code is available, to obtain the graph is still a challenging task, as it needs high understanding when observing system call trace, which requires time and expertise (Eick et el., 2002). The comparison between static analysis and dynamic analysis are as follows:

Table 1 Comparison of static and dynamic analysis (Bergeron et al., 2001)

Characteristics of static analysis	Characteristics of dynamic analysis
Allows complete analysis, because they are not bound to a specific execution of a program and can guarantee all executions of the program.	Allows examination of behaviors that correspond to selected test cases.
Judgment can be given before execution.	Judgment cannot be given before execution.
There is no run-time overhead.	Perform on execution programs.

III **REPRESENTING SOFTWARE COMPONENT**

This section explained the technique to representing software component. In order to show the component relation in a program, programmers have to understand the operations of the program. Understanding the operation is one of the most time-consuming activities especially when the programs are complex, all relevant information must be extracted from the system.

However, buy using different techniques, it will help work become easier, software representation allows the building of a system for critical code review, which can support process relatively easier

for formalization and understanding. Some of techniques to represent software are shown in the figure 1.

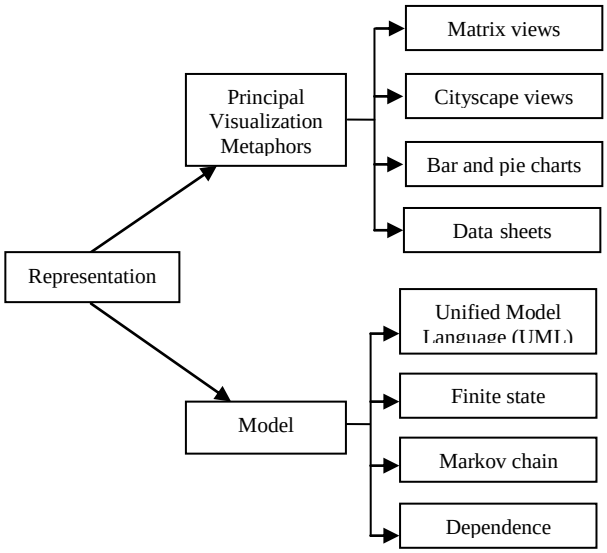

Figure 1. Different techniques in representing software

Base on Figure 1, to identify the software components representation, two different techniques have been studied which are through principal visualization metaphors, and model.

A. **Principal Visualization Metaphors**

Principal visualization metaphors are an effective visual representation to represent the software. The five primary forms of visualization are matrix views, cityscape views, bar and pie charts, data sheets and network views that are related to software structure (Eick et el., 2002, Lanza, 2001). It collects the data about the software routinely and shows it based on colours, different aspects of the data will use the different visual metaphor for each. The details of each form of representation software are base on the analysis by [12] on their studied on visualization.

B. **Model**

The model of software is necessary use for the development of complex and large systems, and it is very useful when dealing with firsthand. Beside, software models are abstractions from code. It can serve as input for program generators and provide documentation to developers as well (ClauÃŸ, 2001).

The main purpose of engineering models is to make possible for developer to understand the important aspects of a complex system before going actual constructing. A quality of the model can help developer on features of a system where there is uncertainty either about requirements or about the capability of a proposed solution. Base on figure 2.3, there are four models to be studied to identify the technique to choose for representing component in software which are Unified Model Language (UML), Finite state machine, Markov chain and dependence graph.

For this study, dependence graph is used as a model to representing component software. A dependence graph relates a variable at one program, point to a variable at another program. In the other words, the dependence graph is represented the dependencies between operations in a program.

The model introduces a node and edge to represent its dependencies. The nodes of the graph represent functions in the program, and edges connecting the nodes represent call paths in the program (Hashemi, 1997). Furthermore, the dependence graph also can be used to determine which functions are called by a particular function.

IV **CALL GRAPH REPRESENTATION**

Call graph is one types of dependence graph. This section describes call graph in representing component software. In Graph theory, a graph represents a collection of nodes that may or may not connect among each other by lines (Deo, 2004). It never considers the size of the nodes, how long the paths are, or whether the paths are straight, or curved. The study of graph properties can be helpful in understanding the characteristics of the software systems (Chatzigeorgiou, 2006), as well as representing any pair of relations between objects from a certain collection (Deo, 2004).

There are many graph representations that have been proposed in recent years to represent variety of features of a program. Basically, a representation of a program can capture characteristics of the program that are of interest in the area of studies (Mall & Samanta, 2009). Besides, this representation is also another way to display information; it helps to break the size and complexity of the software.

Call graph also represents the connectivity of interactions between the components in their relationships. Moreover, it provides binary relation over selected entities in a program, such as methods, classes, subsystem, modules or files. Call graph shows the relation that could be made from one to another entity in any possible execution of the program (Xie & Memon, 2008). Moreover, the call graph is suitable in analyzing tracks of the flow's values between various modules of a program.

By constructing a call graph, nodes of the graph represent functions in the program, and edges connecting the nodes represent call paths (Hashemi, 1997). When trying to understand a system, using the call graph is one of the techniques that are used in software engineering, to ensure that the functions of the system are correctly executed. Call graph is a basic program analysis result that can be used for human understanding of programs, or as a basis for future analysis [18]. To represent component, the call graph is directed, from a caller to a *callee*. Specifically, each node represents a procedure and each edge (*a*,*b*) indicates that procedure *a* calls procedure *b*. Thus, a cycle in the graph indicates recursive procedure calls.

V PROCESS CONSTRUCTING CALL GRAPH

This section contains a proposed process of constructing a call graph that represents components software. To construct a call graph, tools are necessary to use to extract component information and to constructing a call graph. The processes of the overall call graph creation a components level is shown as follow (see Figure 2):

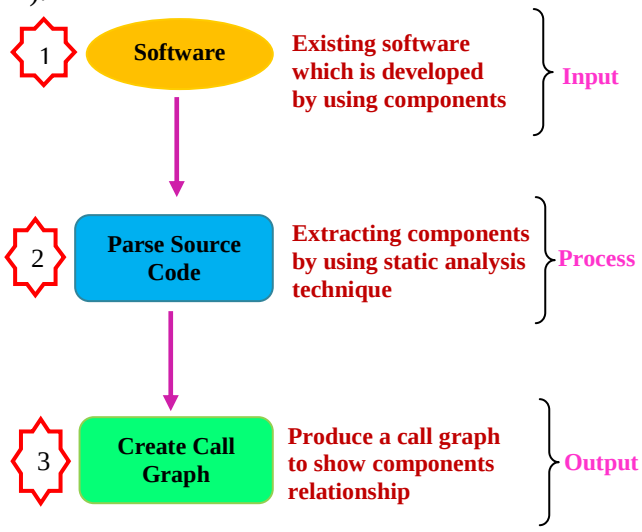

Figure 2. Process in creating a call graph

The first activity requires a software application which has different as an input. The applications of the components must be working properly.

Next, this process requires collecting component information from source code from the sample. This program is developed to extract the information of component's interactions which are selected from the program source code. The component interaction information is extracted using static analysis technique. This program will produced a text file in *dotty* format, which is graph text format.

This method uses filter that exclude Java APIs component library's method names. Therefore, any methods that are not listed as interface operations, such as execution of Java APIs methods, private methods, or any public methods that are not defined in the component interface are ignored.

The technique to extract traces of the software components interaction used in this research is a static analysis which examines code without performing the program execution.

Lastly, based on the information of component interaction in previous process, this phase will used tool name *Graphviz* (which can be freely download) to represent a call graph to show the interaction of component in software.

VI EXPECTED CONTRIBUTIONS

It is expected that this work will contribute to produce a call graph representation that will provide the information of components relating to their interactions, positions and the name of the component that are involved in the software. This information is useful to identify the flow of the system in software.

Furthermore, to produce a call graph also will provide an effective way for those who are unfamiliar with the location of software components by showing them in the form of a call graph which makes it easier for them to understand the flow of software systems when compared to code review. Code review requires careful examination of each line of code in order to find the component in software.

VIII CONCLUSION

As a conclusion, this paper highlights the definition of component, component integration and the static analysis use as a technique to extract the component which is applied in this study. This paper also covers other techniques in representing software component either by using principal visualization metaphor or by using a model. For this study, model is use to represent component software in form of call graph. The call graph is referred base on graph theory which is used by Mall & Samanta, (2009). In order to archive an objective to representing a call graph, process in creating a call graph also has been proposed.

REFERENCES

Ayewah, N., Hovemeyer, D., Morgenthaler, J. D., Penix, J., & Pugh, W. (2008). Using static analysis to find bugs. *Software, IEEE, 25*(5), 22-29.

Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M. M., Lavoie, Y., & Tawbi, N. (2001). Static detection of malicious code in executable programs. *Int. J. of Req. Eng, 2001*, 184-189.Briand, L. C., Labiche, Y., and SÃ³wka, M. M. (2006). *Automated, contract-based user testing of commercial-off-the-shelf components*.

Chatzigeorgiou, A., Tsantalis, N., & Stephanides, G. (2006). *Application of graph theory to OO software engineering*. Paper presented at Proceedings of the 2006 International Workshop on Workshop on Interdisciplinary Software Engineering Research.

ClauÃŸ, M. (2001). *Generic modeling using UML extensions for variability*. Paper presented at the Workshop on Domain Specific Visual Languages at OOPSLA.

Deo, N. (2004). *Graph theory with applications to engineering and computer science*: PHI Learning Pvt. Ltd.

Eick, S. G., Graves, T. L., Karr, A. F., Mockus, A., & Schuster, P. (2002). Visualizing software changes. *Software Engineering, IEEE Transactions on, 28*(4), 396-412.

Fiege, L. (2005). *Visibility in Event-Based Systems.* Unpublished Phd, Technische University at Darmstadt, Darmstadt, Germany.

Hashemi, A., Kaeli, D., & Calder, B. (1997). *Procedure mapping using static call graph estimation*. Paper presented at Proc. Workshop Interaction Between Compiler and Computer Architecture.

Inverardi, P and Tivoli M. (2003). Software architecture for correct components assembly. *Formal Methods for Software Architectures*, 92-121.

Lanza, M. (2001). *The evolution matrix: Recovering software evolution using software visualization techniques*. Paper presented at the Proceedings of the 4th international workshop on principles of software evolution.

Mall, R., & Samanta, D. (2009). A dependence graph-based representation for test coverage analysis of object-oriented programs. *ACM SIGSOFT Software Engineering Notes, 34*(2), 1-8.Xie, Q., & Memon, A. M. (2008). Using a pilot study to derive a GUI model for automated testing. *ACM Transactions on Software Engineering and Methodology (TOSEM), 18*(2), 7.

Piel, E and Gonzalez-Sanchez, A. (2009). *Data-flow integration testing adapted to runtime evolution in component-based systems*. Paper presented at the Proceedings of the ESEC/FSE workshop on Software integration and evolution@ runtime.

Rader, J. A. (1997). *Mechanisms for integration and enhancement of software components*. Paper presented at the Proceedings Fifth International Symposium on Assessment of Software Tools and Technologies.

Rajan, H and Sullivan, K. (2005). *Classpects: unifying aspect-and object-oriented language design*. Paper presented at the Proceedings of 27th International Conference on Software Engineering (ICSE).

Reekie, H. J., & Lee, E. A. (2002). Lightweight component models for embedded systems. Electronics Research Laboratory, College of Engineering, University of California.

Wu, X., & Woodside, M. (2004). Performance modeling from software components. *ACM SIGSOFT Software Engineering Notes, 29*(1), 290-301.H. Reekie, and E. Lee. *Lightweight Component Models for Embedded Systems*: Electronics Research Laboratory, College of Engineering, University of California, 2002.

Xie, Q., & Memon, A. M. (2008). Using a pilot study to derive a GUI model for automated testing. *ACM Transactions on Software Engineering and Methodology (TOSEM), 18*(2), 7.

Xue, J., Hu, C., Wang, K., Ma, R., & Leng, B. (2009). *Constructing a Knowledge Base for Software Security Detection Based on Similar Call Graph*. Paper presented at 2009 Second International Conference on Computer and Electrical Engineering.