Journal of Engineering and Applied Sciences 14 (2): 648-657, 2019
ISSN: 1816-949X
© Medwell Journals, 2019

A Program Visualization Method for Large Scale Software

[1]Joonseon Ahn, [2]Seungcheol Shin, [1]Hyung Joon Lim and [1]Young Sub Lee
[1]School of Electronics and Information Engineering, Korea Aerospace University, Goyang-Si, 10540 Gyeonggi-do, Republic of Korea
[2]Codemind Corporation, 08381 Seoul, Republic of Korea

Abstract: Program visualization supports program comprehension by providing software properties in visualized forms. We introduce and analyze current representative visualization tools from the viewpoint of effective visualization of semantic properties for large-scale software. Then, we present the design and implementation of our visualization tool that helps users understand the relationship among modules of a program at various levels from program statements to packages. The tool has several features to overcome visual complexities of large software such as limitation of displayed nodes and auxiliary table. The tool stores underlying data structures in a graph database and handles concurrency and scalability effectively. We also explain the implementation of our tool and provide experimental results.

Key words: Program visualization, static analysis, program browsing, vulnerability analysis, data flow analysis, control flow analysis

INTRODUCTION

It is very common in software development and maintenance to utilize open source software and update existing program to fix bugs or add new functions. In addition because the size of software is increasing and it is very common that people maintain software that they have not developed, the necessity of tools for understanding software is increasing.

Program visualization supports program comprehension and maintenance by presenting software architecture and runtime property in a visualized form by Roman and Cox (1993), Rech and Schafer (2007) and Gallagher *et al*. (2008). Program visualization tools guides users in understanding software structure and finding parts to be updated for maintenance. There were numerous researches for analysing program properties in more efficient and precise way and many reports compare static analysis tools on their accuracy and speed by Srinivasan and Thambidurai (2007), Emanuelsson and Nilsson (2008), Mantere *et al*. (2009), Li and Cui (2010), Charest *et al*. (2016), Ramos (2016). However, fewer researches deal with effective visualization of analysis results and it is also very important to present program properties in a user-friendly and effective way.

To support program understanding using program visualization effectively, we must handle the limit of display size and human ability of visual comprehension. If visualization becomes too complex, it is not displayable in a limited area and too difficult for users to find information that they want check. Therefore, we must design visualization at various abstraction level appropriate for the kind of program properties and provide methods how users can navigate program points and find information of interest.

In this study, we introduce and analyze the characteristics of representative visualization tools from the viewpoint of effective visualization of semantic properties of large-scale software. Then, we present the implementation of our visualization tool for program understanding. Our tool helps users understand the relationship between modules of a program at various levels from program statements to packages and has several features to handle visual complexity. Using graph database, our implementation handles concurrent access to program information effectively and scales up to millions of LOC.

MATERIALS AND METHODS

A comparative study of program visualization tools: In this study, we present a comparative survey and analysis of program visualization methods of current program analysis and comprehension tools. We select three famous tools those have competitive visualization

Corresponding Author: Joonseon Ahn, School of Electronics and Information Engineering, Korea Aerospace University, Goyang-Si, 10540 Gyeonggi-do, Republic of Korea

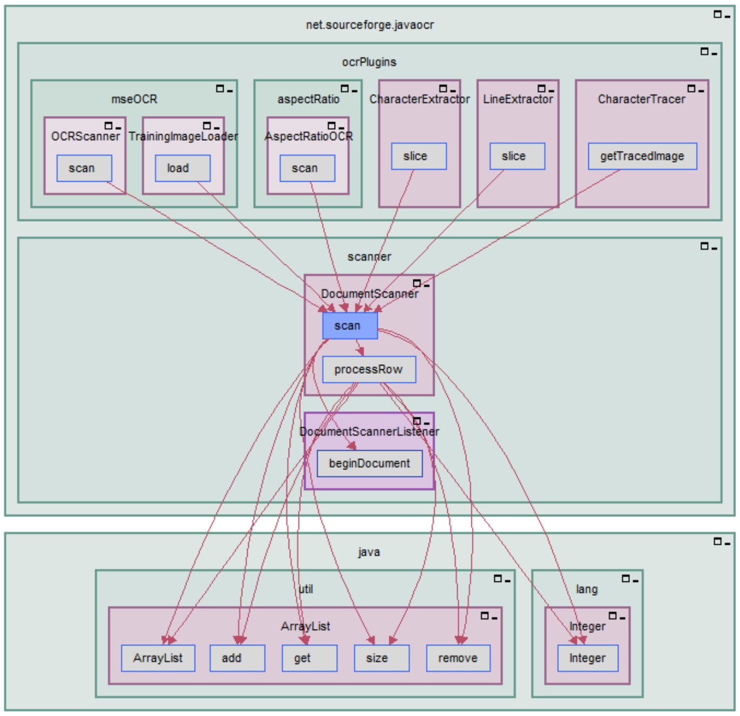


Fig. 1: Subsystem architecture diagram of Imagix 4D [Imagix]

representations which are Imagix 4D®, Understand® and Codesonar®. Imagix 4D® is a source code analysis tool for C, C++ and Java developed by Imagix Corporation. It provides various program properties such as hierarchical subsystem architecture, interprocedural and intraprocedural control flow and data flow in visualized forms and use-def relation among variable usages [Imagix] (Anonymous, 2018a-b).

The most distinctive point of Imagix 4D visualization is the unification of subsystem architecture and program analysis results. Subsystem architecture diagram shows hierarchical structure of modules of multiple levels and various program properties such as data flow, control flow and task interaction can be added to the visualization of hierarchical structure. In Fig. 1, rectangular structure shows hierarchical structure of multi-level modules from methods to packages and arrows express control flow.

Imagix 4D® also has traditional visualizations such as call graph and intra-procedural data flow graph which is shown in Fig. 2 and 3, respectively. Call graph of Imagix 4D® is distinctive in that it displays statements related to each function call in each function node. Intra-procedural data flow graph shows data flow information in a function and users can highlight data flow related to a specific assignment.

Although, Imagix 4D® subsystem architecture diagram helps users check the hierarchical module structure and other properties simultaneously, it becomes too complex for large software and Imagix 4D® has little consideration for visual complexity. Understand® by scientific tool works is a code visualization tool for promoting program comprehension developed [SciTools]. Understand® presents UML Class diagrams, treemaps for program metrics, interprocedural, intraprocedural control flow graph and butterfly graph that is customizable in callee and caller depth.

TreeMapping is an information visualization form for hierarchical data using nested rectangles. Figure 4 shows a TreeMap of Understand® that provides an overview of a program where module structures and size of files are represented.

Like Imagix 4D®, cluster graphs of understand® provides simultaneous presentation of hierarchical view and relation among modules. Users can access control flow relation from architecture, class and function level and the graphs can be customized with caller oriented, callee oriented and butterfly variations. In Fig. 5, a cluster graph shows the control flow of a C program where a file node is expanded to provide hierarchy information.

Codesonar® is a static source and binary analysis tool developed by Gramatech [Codesonar]. Codesonar® identifies bugs in a program and provides an advanced user experience in its visualization using its advanced zoom-in and out features and vulnerability analysis

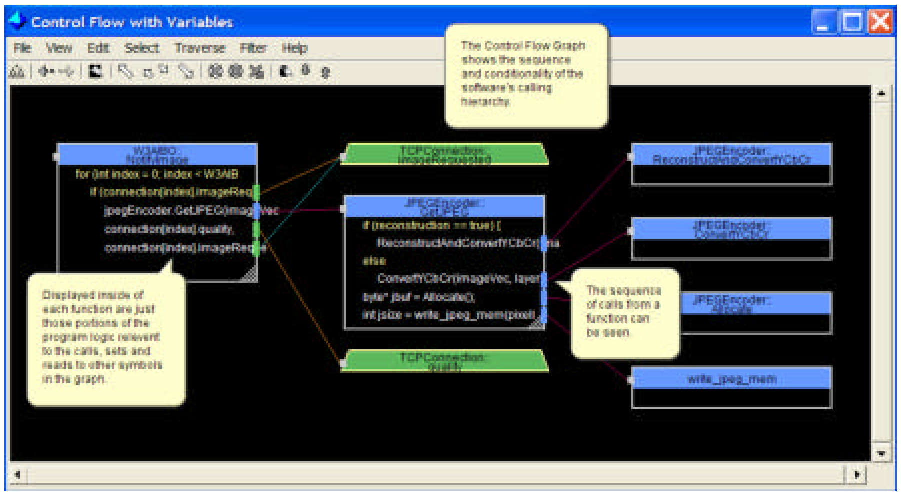

Control Flow with Variables
File View Edit Select Traverse Filter Help
The Control Flow Graph shows the sequence and conditionality of the software's calling hierarchy.
W3ABO: NotifyImage
for (int index = 0; index < W3AIB
if (connection[index].imageReq
jpegEncoder.GetJPEG(imageVec
connection[index].quality,
connection[index].imageReq
TCPConnection: imageRequested
JPEGEncoder: GetJPEG
if (reconstruction == true) {
ReconstructAndConvertYCbCr(ima
else
ConvertYCbCr(imageVec, layer
byte* jbuf = Allocate();
int jsize = write_jpeg_mem(pixel
TCPConnection: quality
JPEGEncoder: ReconstructAndConvertYCbCr
JPEGEncoder: ConvertYCbCr
JPEGEncoder: Allocate
write_jpeg_mem
Displayed inside of each function are just those portions of the program logic relevant to the calls, sets and reads to other symbols in the graph.
The sequence of calls from a function can be seen

Fig. 2: Call graph of Imagix 4D [Imagix]

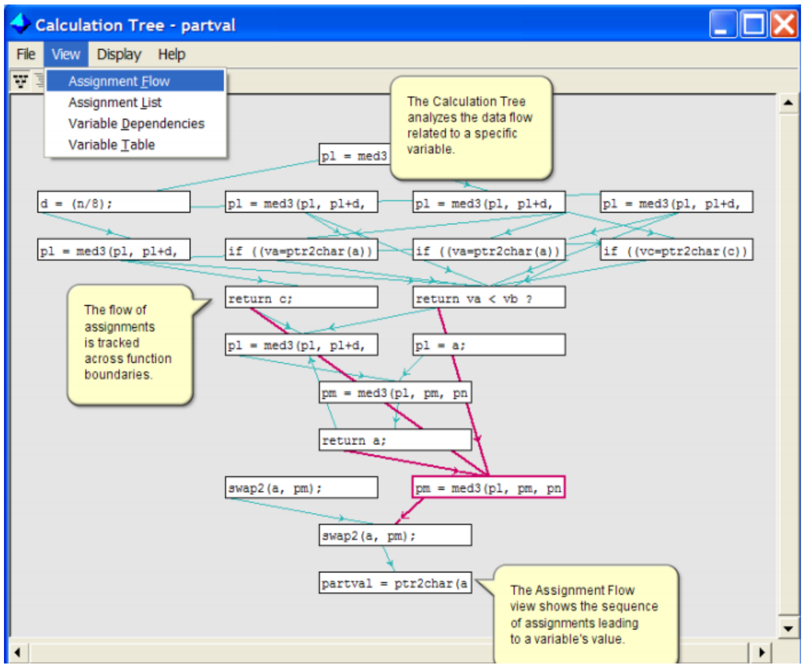


Fig. 3: Intra-Procedural data flow information of Imagix 4D [Imagix]

support based on tainted information visualization. Codesonar® provides a smooth zoom in and out visualization providing a map-like user experience such as Google Map as shown in Fig. 6 and 7. This, feature enables the seamless browsing of program modules from the highest level of the entire program structure to the lowest level of intra-procedural control flow. Each node can be a package, a class, a file, a function and so, on and they appears and disappears with the zoom-in and out action. This map-like visualization provides high-quality user experience and users can easily grasp the relation among program parts at the desired level. In addition, Codesonar® provides various visualization alternatives such as TreeMap, Flat, Map, circuit for architectural structure and program control and data flow. Codesonar® finds various vulnerabilities in source code and binary file using static analysis. Given analysis results, users must check whether the vulnerabilities found are false alarm or actually lead to exploit. Codesonar® provides information about vulnerabilities using visual tainted data tracking. By tracing the highlighted nodes in visualizations, users can traverse along the intermediate and source points of

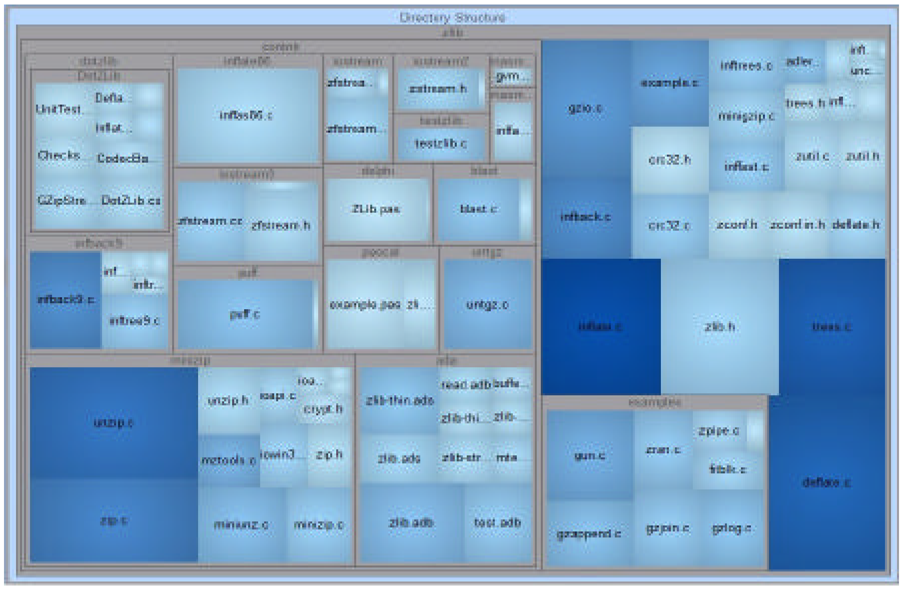


Fig. 4: Treemapping of Understand® [SciTools]

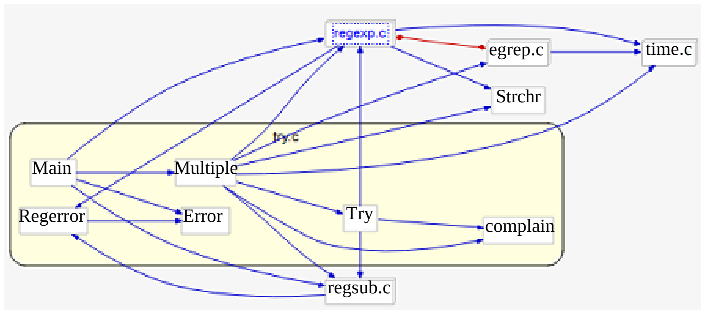


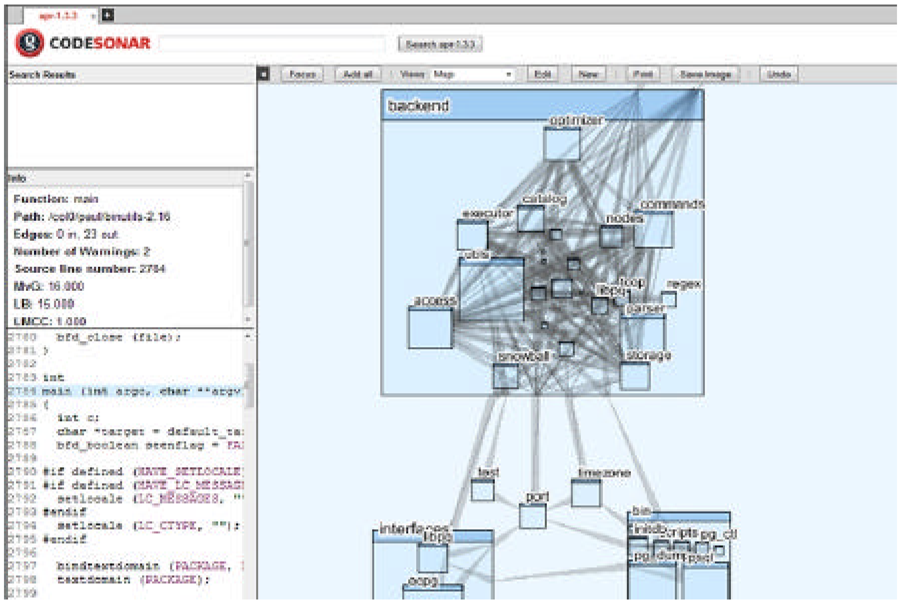
Fig. 5: Cluster graph [SciTools]

tainted dataflow. Although, Map-like visualization of Codesonar® is very attractive user experience there is a room for improvement related to visual complexity. In zoom-in and out visualization, the relative location of nodes is fixed in advance for map-like user experience. Therefore, when users zoom-in toward an interested node, nodes directly related to the node can disappear in the window with large programs. Therefore, map-like positioning of nodes for graph representation can be inadequate for identifying various program properties for large programs. In addition for practical programs, simultaneous visualization of multiple levels usually become too complex.

Design of a program visualization tool: In this study, we explain our program visualization system, Codemind Browser® which assists program understanding and vulnerability examination. Our program visualization tool has the following features:

- Consistent graph representation for all levels that are directory, package, classes, methods and statements
- Dealing with visual complexity using context-based node limitation, heavy mode and class-package assist
- Co-browsing of program codes and visual representations
- Visualization of intra-procedural data flow and control flow information to help check and eliminate vulnerabilities
- Package class table which help users browse related classes or methods of complex programs

Figure 8 shows the overall structure of Codemind Browser®. The integrated development environment (IDE) is composed of file browser, editor, visualization tabs and class/function list table. The Integrated Development Environment (IDE) is composed of file browser, editor, visualization tabs and class/function list



Function: main
Path: /col0/paul/binutils-2.16
Edges: 0 in, 23 out
Number of Warnings: 2
Source line number: 2784
MvG: 16.000
LB: 15.000
LMCC: 1.000

```
  bfd_close (file);
}

int
main (int argc, char **argv
{
  int c;
  char *target = default_ta
  bfd_boolean seenflag = FA

#if defined (HAVE_SETLOCALE
#if defined (HAVE_LC_MESSAGE
  setlocale (LC_MESSAGES, "
#endif
  setlocale (LC_CTYPE, "");
#endif

  bindtextdomain (PACKAGE,
  textdomain (PACKAGE);
```

Fig. 6: Visualization of software architecture using zoom-in and zoom-out [Codesonar]

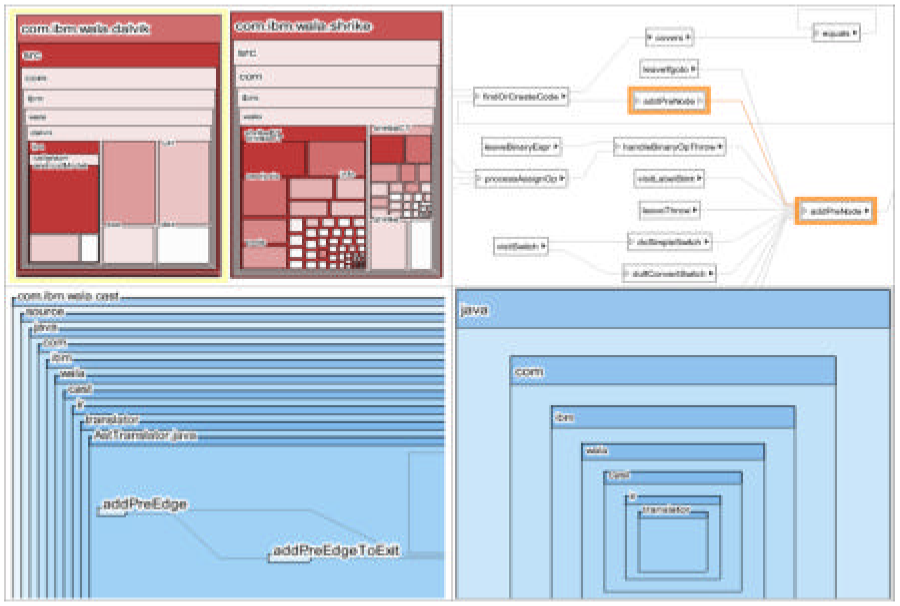
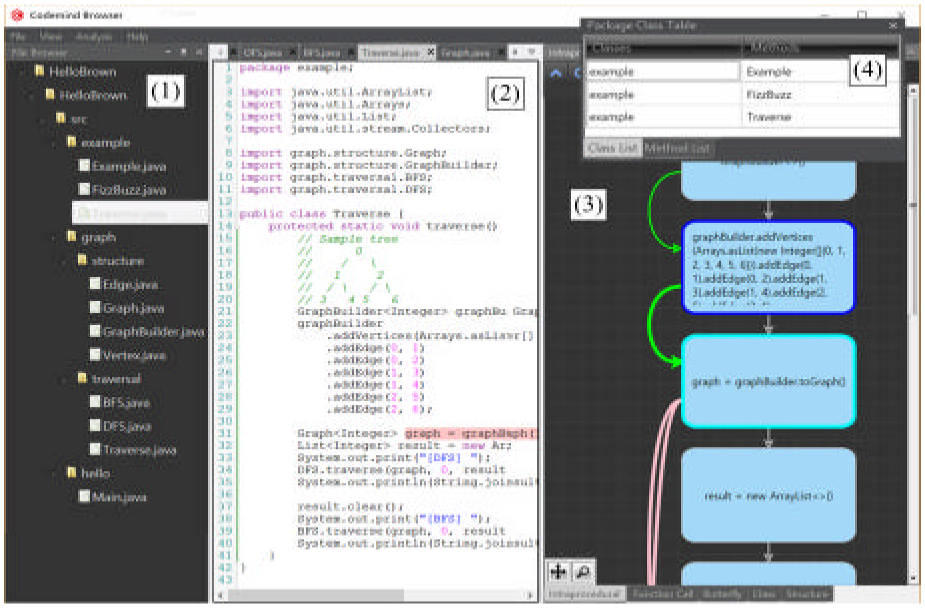


Fig. 7: Codesonar visualizations: TreeMap, Flat, Map, Circuit (clockwise from top left)[Codesonar]

table. File Browser (1) shows the tree of files and directories of given project which help users to find files of interest. The file can be a starting point of bottom-up browsing to understand the program. Editor Window (2) is used to not only check and update program text but also access visualized information related to each program construct such as procedures, classes and packages. Editor Window and Visualization Window works in a cooperative way according to user action in each Window. Visualization Window (3) presents visualized properties of a program. It is composed of tabbed panels that provide multi-level visualizations which are intraprocedural graph, function call graph, butterfly graph, ClassGraph and program structure graph. They behaves in cooperation with each other and users can reorganize the tabs into new inner windows, so that, they can examine multiple visualizations simultaneously. In Fig. 6-8, intraprocedural graph is displayed.



```
package example;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

import graph.structure.Graph;
import graph.structure.GraphBuilder;
import graph.traversal.BFS;
import graph.traversal.DFS;

public class Traverse {
    protected static void traverse()
        // Sample tree
        //       0
        //     /   \
        //    1     2
        //   / \   / \
        // 3   4 5   6
        GraphBuilder<Integer> graphBu Grap
        graphBuilder
            .addVertices(Arrays.asList(new[]
            .addEdge(0, 1)
            .addEdge(0, 2)
            .addEdge(1, 3)
            .addEdge(1, 4)
            .addEdge(2, 5)
            .addEdge(2, 6);

        Graph<Integer> graph = graphGraph(
        List<Integer> result = new Ar;
        System.out.print("[DFS] ");
        DFS.traverse(graph, 0, result
        System.out.println(String.joinsul

        result.clear();
        System.out.print("[BFS] ");
        BFS.traverse(graph, 0, result
        System.out.println(String.joinsul
    }
}
```

Classes	Methods
example	Example
example	FizzBuzz
example	Traverse

Fig. 8: Components of Codemind visualizer

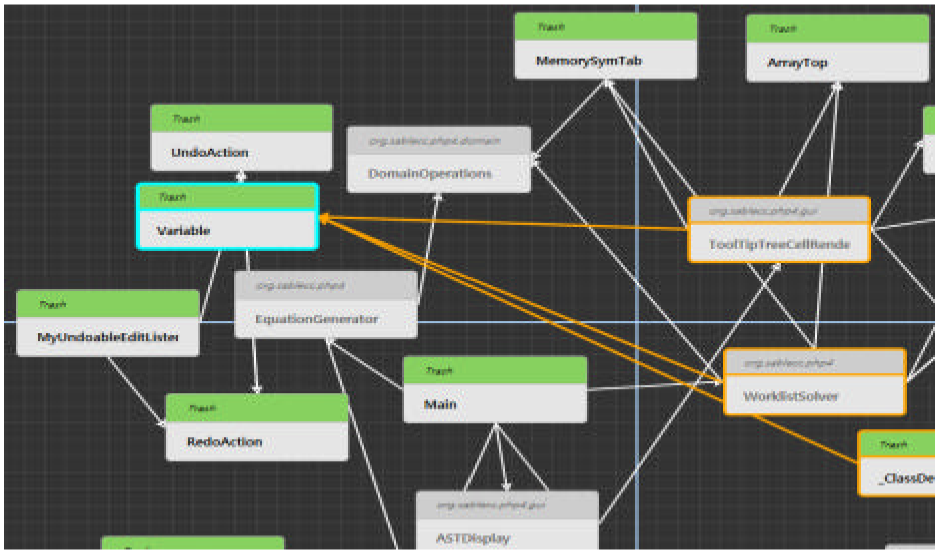


Fig. 9: ClassGraph

Package class table lists classes and methods in packages. It shows those components in accordance of user selection in editor window and visualization Window. Using this table, users can get information about classes or functions without moving to visualization of package level. The tabs in visualization window present visualizations of program constructs in a hierarchical way. Program structure graph shows overall structure of a program using a graph whose node represents packages or directories and edges represents control flow between nodes. If we select a node, class and procedure table shows classes and functions in the package. Users can move to ClassGraph tab by double-clicking a package node. ClassGraph is composed of class nodes and edges represents control flow among classes. Because it is not visually too complex to include all the classes in a program, ClassGraph include classes of the package that is selected in package graph or in Editor Window. In addition to the class nodes of a selected package, other classes that are directly related to the classes in the selected package is also included in the graph. Using this classes, users can navigate to other packages that are directly related. The ClassGraph of Fig. 9 shows control flow relation among classes in Trash package. In addition, it shows other classes which are directly related to the classes in the package. Class Nodes such as worklistSolver, domainOperations, ASTDisplay, EuationGenerator, DocumentSizeFilter and ToolTipTreeCellRende, whose names are in grey colour are such examples. Using these outside nodes, users can navigate to other related packages. If we click the ASTDisplay class node, the ClassGraph shows the relation among classes in the org.sablecc.php4.gui package. In graph representations, if we move a mouse over a node, nodes that are directed related are high lighted. In addition, if we double-click a node, the visualization window changes to the lower level visualization tab for the node. That is, if we double click a class node in a ClassGraph, the visualization windows shows function call graph for the class.

Function call graph presents control flow among procedures or methods in a class. Like ClassGraph, function call graph also includes outside-class nodes that are directly related. The control flow relation among procedures is also presented using the butterfly graph tab. Butterfly graph shows the control flow from and to a procedure which are selected in the editor window or call graph. We can also expand the graph in both directions by clicking a node at end. Using this visualization, we can easily understand procedure call relation from the viewpoint of a particular procedure.

If we double-click a node in call graph or butterfly graph tab, the visualization window shows the intraprocedural graph which is shown in (3) of Fig. 8. Intraprocedural graph shows the control and data flow in a procedure. We can get dataflow information by clicking check boxes for use-def and def-use information and selecting a node of interest.

When we visualize properties of large programs, we must balance the complexity of the graphical representation and quantity of information in a visualization. Because naive visualization of practical software usually become too complex for users, we prepared several measures to handle the problem.

For program call graph and ClassGraph, those nodes that belongs to a designated class or package and outer nodes that are directly related are shown in the display. Therefore, to view other nodes, users can navigate to other groupings by clicking the directly related nodes or by selecting the other package or class in the upper level representation. Other visualization tools does not support this kind of consistent limitation within a selected package or class.

If the number of nodes in a graphical representation increase, the relationship among nodes increases in the order of n^2 and the visualization becomes incomprehensible. To handle this difficulty, when the number of nodes is larger than a predefined number, the visualization changes to the heavy mode. In heavy mode, our visualization displays only those edges that are adjacent to a selected node.

Although, visualized representation supports user comprehension of program properties, users often get difficulty in finding specific nodes in a graph with many nodes. As an auxiliary mean to find interested nodes, we provide a pop-up table of package structure table. It shows classes in a package and methods in a class where the package and class selection is decided in accordance with the Visualization Window and Editor Window. Using this table, we can easily find a specific node using the sorted list. In addition, the table support multilevel navigation. For example, while examining a function call graph, we can easily move other function call graphs that belongs to the same class. Although, such program browsing is easily supported using multilevel visualization which is provides by other tools such as Codesonar®, multilevel visualization becomes too complex for large programs which reduces effectiveness of visualization and consumes much computation.

RESULTS AND DISCUSSION

Program visualizers execute in conjunction with program analyzers and they share various data structures that represent information about a program. Codemind Browser® uses our static analysis platform and Fig. 10 shows the overall structure of our analysis and visualization framework.

Static analysis platform composed of a frontend for Java and C++ and various static analysis modules such as data-flow analysis, control-flow analysis and semantic-based analysis such as abstract interpretation. Program visualizer constructs visual representation using results from the static analysis platform and displays them via. IDE. To support consistent and extendable sharing of program properties among static analysis modules and the visualizer, our tool models program source code, intermediate code and all the analysis results with graphs and stores them in an off-the-shelf graph database such as Neo4J. Program visualizer and analysis modules access program information using graph database query and we can get the following advantages (Table 1 and 2):

- Stable support for large-scale program analysis up to millions of LOC
- Effective handling of race condition caused by concurrent access of multiple modules such as program analysis modules, visualization modules and

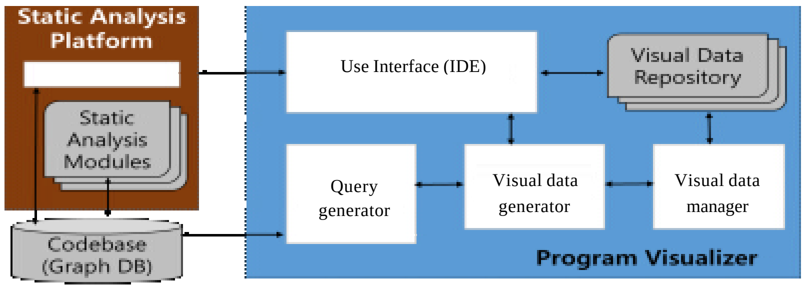


Fig. 10: Overall structure of our program visualization framework

Table 1: Time consumption for visualization

Visualized information/Software	Module	Time (sec)
Program structure		
Hibernate	org.hibernate.search.test.configuration	1.44
Hibernate	org.hibernate.search.spi	0.87
MYSQL-connector-Java	testsuite	0.70
MYSQL-connector-Java	testsuite.regression	0.32
Polyglot	polygot.types	0.56
Polyglot	polygot.ast	0.39
Control Flow (Inter procedural)		
Hibernate	SearchintegratorBuild	0.35
Hibernate	BuildContext	0.13
MYSQL-connector-Java	HangingInputStream	0.53
MYSQL-connector-Java	ServerPreparedStation	0.27
Polyglot	TypeNode_c	0.97
Polyglot	Node_c	0.55
Control Flow (Intra Procedural)		
Hibernate	buildNewSearchFacto	1.42
Hibernate	FindClass	0.18
MYSQL-connector-Java	getExceptionIntercept	0.12
MYSQL-connector-Java	proceedHandshakeWi	0.52
Polyglot	visit	0.37
Polyglot	exit	0.58
Use-def analysis (Intra-procedural)		
Hibernate	entityindexBinding	1.16
Hibernate	MetadataProvider	1.88
MYSQL-connector-Java	rowPacket	1.05
Polyglot	n	0.54
Polyglot	v	1.17
Def-Use Analysis (Intra-profcedural)		
Hibernate	rootFactory	1.38
Hibernate	factoryState	1.32
MYSQL-connector-Java	rowPacket	0.87
Polyglot	m	0.53
Polyglot	n	1.10

Table 2: Effectiveness of visualization support for vulnerability detection

Visualization support/metric	Averageresult
No visualization support (A)	
Exact analysis ratio	77.4
Exact correction ratio	52.5
Analysis time	1:40
Correction time	1:30
With visualization support (B)	
Exact analysis ratio	89.6
Exact correction ratio	58.6
Analysis time	1:11
Correction time	1:06
B-A	
Exact analysis ratio	12.3
Exact correction ratio	6.0
Analysis time	-0:29
Correction time	-0:24

program editors. Handling race condition is important for the usability of our system because it enables concurrent and incremental execution of program modification, program analysis and visualization. Therefore, we can update visualization in accordance with the progress of program update and analysis.

We tested our program visualization framework using practical Java programs such as Hibernate (111,419 lines), MYSQL-connector-Java (139,680 lines) and Polyglot (128,881 lines). Summaries the time consumptions for displaying various information given an user request. Our tool completed visualization between 0.13 and 1.88 sec which is acceptable for human users. This shows that using graph DB satisfies the required speed. Because usefulness of a program visualization depends on personal experience and usage, quantitative measurement of effectiveness is not straightforward. To measure the effectiveness of our visualization in an indirect way, we applied our visualizer to program vulnerability analysis Diaz and Bermejo (2013). Our vulnerability detection module for our experiment finds important weaknesses such as SQL injection (CWE-89), path traversal (CWE-22), OS command injection (CWE-78), XPath injection (CWE-643), LDAP injection information exposure through an error message (CWE-329), information exposure through comments (CWE-396), exceptional iteration (CWE-835), use of hard-coded password (CWE- 209), detection of error condition without action (CWE-390) and improper handling of exceptional conditions (CWE-396). For the purpose of fairness, we divided 11 people into three groups and collected three sets of 50 programs. Two groups analyzed each test set and each group analyzed two test sets. Each group used visualization support for one test set and did not used visualization support for the other set. Because there can be false-positive vulnerability reports or missed vulnerabilities, testers must examine analysis results.

Table 2 shows our experiment results. The metrics are the exact analysis ratio, the exact correction ratio and the average false-positive checking time and correction time for each program. In the table, testers got more productivity with visualization support which shows that our tool was useful for understand program.

CONCLUSION

In this study, we presented a survey of visualization support of existing program analysis and understanding tools and described a design and implementation of our visualization tool. Our tool provides a consistent browsing from the intraprocedural level to the whole program structure level. Our tool maintains an appropriate degree of visual complexity by restricting the number of nodes in a window. To supplement the restriction, our package class table enables users to access related methods or classes for the browsing context. In our implementation, a static program analyzer executes concurrently with visualization tool and we have solved the problem of scalability and race condition by using a graph DB instead of the heap allocation. Our experiment shows that our tool can handle large programs and is effective for program understanding.

RECOMMENDATIONS

Our further work includes finding more effective visualization method of data dependency information at upper levels such as class and package levels. In addition, to design effective visualization support for checking various vulnerability forms will be very useful.

ACKNOWLEDGEMENTS

This work was supported by 2017 Korea Aerospace University faculty research grant.

REFERENCES

Anonymous, 2018a. CWE/SANS TOP 25 most dangerous software errors. SANS Institute, San Francisco, California, USA. https://www.sans. org/ top 25-software-errors

Anonymous, 2018b. GrammaTech CodeSonar delivering resiliency for today's IoT devices. GrammaTech, Ithaca, New York, USA. https://www.grammatech. com/products/codesonar

Anonymous, 2018c. Reverse engineering and code analysis of C, C++ and Java. Imagix Corporation, California.

Anonymous, 2018d. The Neo4J graph platform. Neo4j's Inc, San Mateo, California, USA.

Charest, T., N. Rodgers and Y. Wu, 2016. Comparison of static analysis tools for java using the Juliet test suite. Proceedings of the 11th International Conference on Cyber Warfare and Security, March 17-18, 2016, Boston University, Boston, USA., pp: 431-438.

Diaz, G. and J.R. Bermejo, 2013. Static analysis of source code security: Assessment of tools against SAMATE tests. Inf. Software Technol., 55: 1462-1476.

Emanuelsson, P. and U. Nilsson, 2008. A comparative study of industrial static analysis tools. Electron. Notes Theor. Comput. Sci., 217: 5-21.

Gallagher, K., A. Hatch and M. Munro, 2008. Software architecture visualization: An evaluation framework and its application. IEEE. Trans. Software Eng., 34: 260-270.

Li, P. and B. Cui, 2010. A comparative study on software vulnerability static analysis techniques and tools. Proceedings of the IEEE International Conference on Information Theory and Information Security (ICITIS), December 17-19, 2010, IEEE, Beijing, China, ISBN:978-1-4244-6942-0, pp: 521-524.

Mantere, M., I. Uusitalo and J. Roning, 2009. Comparison of static code analysis tools. Proceedings of the 3rd International Conference on Emerging Security Information, Systems and Technologies SECURWARE'09, June 18-23, 2009, IEEE, Athens, Glyfada, ISBN:978-0-7695-3668-2, pp: 15-22.

Ramos, A., 2016. Evaluating the ability of static code analysis tools to detect injection vulnerabilities. Ph.D Thesis, UMEA University, Umea, Sweden.

Rech, J. and W. Schafer, 2007. Visual support of software engineers during development and maintenance. ACM. SIGSOFT. Software Eng. Notes, 32: 1-3.

Roman, G.C. and K.C. Cox, 1993. A taxonomy of program visualization systems. Comput., 26: 11-24.

Srinivasan, N. and P. Thambidurai, 2007. On the problems and solutions of static analysis for software testing. Asia J. Inform. Technol., 6: 258-262.