JavaPDG: A New Platform for Program Dependence Analysis

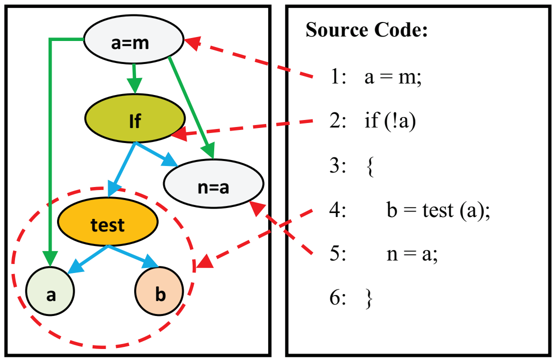
Gang Shu, Boya Sun, Tim A.D. Henderson, Andy Podgurski
The Dept. of Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, OH 44106
<gang.shu, boya.sun, tadh, podgurski>@case.edu

***Abstract*—Dependence analysis is a fundamental technique for program understanding and is widely used in software testing and debugging. However, there are a limited number of analysis tools available despite a wide range of research work in this field. In this paper, we present JavaPDG[1], a static analyzer for Java bytecode, which is capable of producing various graphical representations such as the system dependence graph, procedure dependence graph, control flow graph and call graph. As a program-dependence-graph based analyzer, JavaPDG performs both intra- and inter-procedural dependence analysis, and enables researchers to apply a wide range of program analysis techniques that rely on dependence analysis. JavaPDG provides a graphical viewer to browse and analyze the various graphs and a convenient JSON based serialization format.**

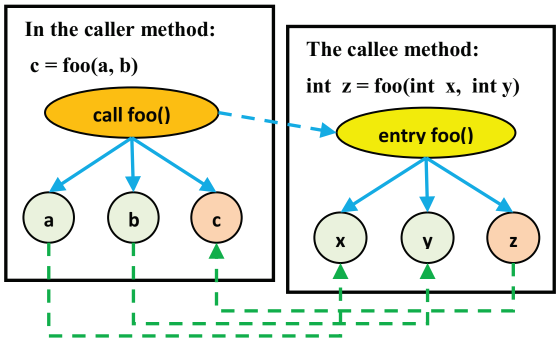
Keywords-program dependence graph; system dependence graph; procedure dependence graph; call graph; Java Virtual Machine; Java bytecode

I INTRODUCTION

Program dependence analysis is a fundamental technique for program understanding, and is widely used in software testing and debugging. A program element, *x*, is said to be dependent on another one, *y*, if *y* controls the execution of *x* or influences the data utilized by *x*. Techniques for dependence analysis can be categorized into two types [1]: static analyses [2, 3] and dynamic analyses [4, 5]. Static dependences are computed by taking all the possible executions into consideration without actually executing the program; in contrast the computation of dynamic dependences relies on execution profiles for a given test suite. This work takes the static approach and does not incorporate any run-time information. Compared with the large number of publications on dependence analysis, there have been only a few implementations. Moreover, due to the nature of the problem most tools target particular languages (often C or related languages), and only few exist for object-oriented languages like Java. CodeSurfer [6, 7] is a typical static analysis tool from *GrammaTech* for C/C++. Unlike some well-known frameworks such as Soot [8] and Wala [9], which create intermediate representations from Java bytecode, JavaPDG analyzes Java bytecode directly.



(a) Example PDG



(b) Example SDG

Figure 1

As illustrated in the Figure 1(a), a PDG [10] is defined as a labeled, directed graph that maps out control dependences (blue edges) and data dependences (green edges) between elements in a program. A *system dependence graph* (SDG) [11] is a generalization of PDG and contains one *procedure dependence graph* (pDG) for each method. In the Figure 1(b) for an example SDG, two pDGs are linked together by inter-procedural control dependence edges (blue dashed lines) and data dependence edges (green dashed lines). A significant body of recent research applies PDG in combination with graph mining techniques such as frequent subgraph mining and subgraph matching in order to discover implicit programming rules and rule violations in software (e.g., [12-16]); to conduct change impact analysis for evolving software systems (e.g., [17, 18]); and to detect semantically similar code from a code base (e.g., [19-21]). Empirical studies of the above research are currently limited to C/C++ programs because there is a lack of PDG-based tools for other languages such as Java. The goal of this work is to fill a gap by facilitating

[1] The JavaPDG tool and manual, as well as figures in this paper are publicly available at http://selserver.case.edu:8080/javapdg/.

the adaptation of proposed dependence-based analysis techniques to Java bytecode programs.

We adopt and implement the approaches proposed by Zhao [22] on dependence analysis for Java bytecode. JavaPDG implements static dependence analysis for *Java Virtual Machine* (JVM) bytecode. The tool parses the bytecode of a Java program, computes the SDG and related graphs, and stores the data for each program in a database. JavaPDG includes tools for visualizing the graphs it produces and for exporting the data in the JSON format. Additionally, users are able to query the output using SQL by utilizing Apache Derby [23].

The remainder of the paper is organized as follows. Section II presents background on dependence analysis for Java bytecode. Section III gives the overview of JavaPDG and introduces analysis approaches behind. Section IV gives a quick view of using JavaPDG. Section V introduces related work on (1) applying dependence analysis in software testing and debugging, (2) related Java bytecode analysis frameworks, and (3) various modifications to the traditional SDG for representing Java source code. Finally, we summarize the features of the JavaPDG and propose future improvements in the Section VI. We give a demo description in the Appendix.

II BACKGROUND

In Java, source code is compiled to bytecode, a binary format that contains loading information and execution instructions for the JVM [24]. There are several other languages which also target the JVM such as: Scala [25] and Clojure [26]. The JVM is a stack oriented virtual machine with a bytecode consisting of a mixture of high and low level instructions. The high level instructions deal with object manipulations such as getting, setting field members and invoking methods. The low level instructions do stack manipulations and basic arithmetic for a variety of data types.

A JVM instruction consists of a one-byte opcode that indicates a particular operation, followed by zero or more operands specifying the constants, references or local variables involved in the operations. Unlike human-readable source code, bytecode encodes the result of parsing and semantic analysis on the source code, and they therefore allow much better performance than direct interpretation of source code.

Traditional dependence analysis has been employed to a range of languages, however Zhao [22] pointed out that the existing techniques cannot be applied to Java bytecode directly due to the specific features of JVM. Zhao also gave guidance to analyze control flow in bytecodes [28], and introduced primary types of intra-procedural dependences specific to Java bytecode [22].

Figure 2 shows an example SDG for Java bytecode. This SDG is constructed from a Java class with two methods and hence consists of two pDGs. The left side shows source code statements compiled into Java bytecode instructions. The right side shows pDG vertices (corresponding to JVM instructions) linked together by control-dependence edges (blue) and data-dependence edges (green). Intra-procedural dependences are shown as solid lines while inter-procedural dependences are shown as dashed lines.

III OVERVIEW OF JAVAPDG

Dependence analysis for Java bytecode introduces some challenges, mainly due to: (1) its complex branching instructions including unconditional branches, simple conditional branches and compound conditional branches; (2) its stack-based architecture, in which stack cells store intermediate calculations and may lead to implicit data flow between instructions; and (3) Java specific features, such as instance method invocation which implicitly passes the reference *this* into the callee method, adding additional control and data dependence edges.

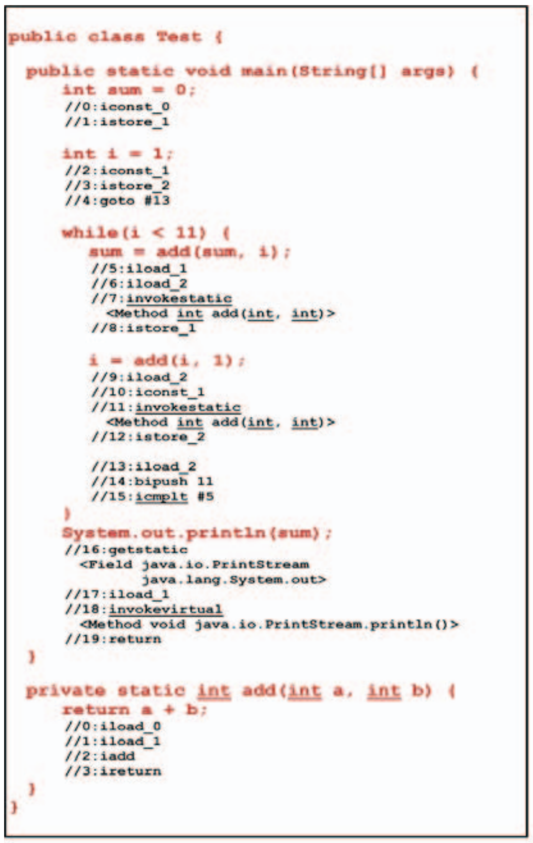
A. *Dependence Analysis*

To address the above difficulties, JavaPDG evaluates and implements the primary types of dependences in a bytecode program identified by Zhao [22] for dependence analysis. We ignore some sources of control dependences mentioned in Zhao's work such as unconditional branching instructions *goto, goto_w, jsr* and *jsr_w*, because they result in over-expansion of SDG produced. Though these instructions can change the flow of control for the instruction execution, they are usually used with conditional branching instructions and hence were left out of consideration.

1) Intra-Procedural Control Dependences

Intra-procedural control dependences represent interactions due to conditional control flow between instructions inside a method. An instruction, *x*, is control dependent on another instruction, *y*, if *y* controls whether or not *x* is executed. For example all the instructions in the body of the *if*-statement (and *else*-statement) are control dependent on the branching instruction of the *if*-statement. Thus, identifying control conditions that may affect the program execution is the first step.

In the JVM, a branching instruction can conditionally cause program execution to jump to an indicated instruction or continue to the next instruction. Such instructions include: (1) simple branching instructions: *ifeq, ifne, iflt, ifle, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmple, if_icmpgt if_icmpge, if_acmpeq* and *if_acmpne*, and (2) compound branching instructions: *tableswitch* and *lookupswitch*. Since these instructions can control which instructions the JVM executes, they are the direct source of control dependences.



```
public class Test {

  public static void main(String[] args) {
    int sum = 0;
    //0:iconst_0
    //1:istore_1

    int i = 1;
    //2:iconst_1
    //3:istore_2
    //4:goto #13

    while(i < 11) {
      sum = add(sum, i);
      //5:iload_1
      //6:iload_2
      //7:invokestatic
        <Method int add(int, int)>
      //8:istore_1

      i = add(i, 1);
      //9:iload_2
      //10:iconst_1
      //11:invokestatic
        <Method int add(int, int)>
      //12:istore_2

      //13:iload_2
      //14:bipush 11
      //15:icmplt #5
    }
    System.out.println(sum);
    //16:getstatic
      <Field java.io.PrintStream
            java.lang.System.out>
    //17:iload_1
    //18:invokevirtual
      <Method void java.io.PrintStream.println()>
    //19:return
  }

  private static int add(int a, int b) {
    return a + b;
    //0:iload_0
    //1:iload_1
    //2:iadd
    //3:ireturn
  }
}
```

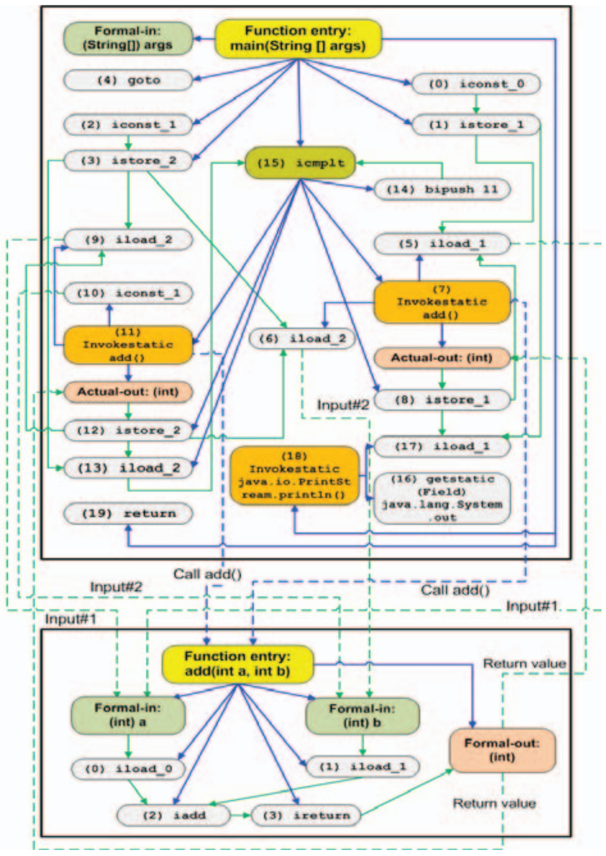


Figure 2 An example Java system dependence graph (right side) generated from Java bytecode (left side).

In addition to the explicit control dependences arising from branch instructions, method entries also control which set of instructions the machine executes. To facilitate inter-procedural dependence analysis and to collect caller and callee functions together, we create a special entry point for each method, and it controls all top-level instructions inside a method.

Another source of control dependences comes from call-sites, which manage the type and the number of parameters passed into and returned from a callee method. Every call-site directly controls its actual-input parameters and actual-output parameter (if any). Many programming patterns mined by Chang et al [12, 13] involving call-sites indicate that the surrounding context of call-sites is a highly probable spot to discover program defects.

Consider, for example, the function *main()* in Figure 2. Control dependence arises from branch instructions (colored blackish green). The branching instruction #15 *if_icmplt* (corresponding to the *while*-statement in the source code) pops the top two ints (say *value1* and *value2*) off the JVM stack and compares them. If *value2* is greater than or equal to *value1*, execution continues at the next instruction #16; otherwise program execution jumps to instruction #5. As a result, instructions #7, 8, 11, 12, 13 and 14 are control dependent on instruction #15.

Control dependence may also arise due to call-site instructions (colored orange). The instruction #7 calls *add*() which has two actual-input parameters and one return value, so instruction #7 controls instruction #5, 6 and its actual-out node. The call-site instruction #11 invokes the same function *add*() as instruction #5 does, thus it similarly controls instructions #9 and 10 and an actual-out node. The call-site instruction #18 is a little different because its callee method does not contain a return value, so it merely controls its two actual-input parameters #16 and #17.

Control dependence also arises from the method entry that controls every top-level instruction (without a common control-dependent predecessor) inside a method. In the example *main*() function, instructions #0, 1, 2, 3, 4, 15, 18, 19 and the formal input parameter are control dependent on the method-entry node.

2) Intra-Procedural Data Dependences

Intra-procedural data dependences identify the data flow between instructions within a single method. Using reaching definition and upward exposed uses analysis, use-to-definition chains can be constructed.[29] From there, the data dependence graph for the procedure is quite easy to produce. Unfortunately, the traditional definition-use chain does not work for implicit data-flow between instructions via the JVM stack. For example, instruction *istore* stores an integer from the stack to a local variable, while instruction *iload* retrieves an integer from a local variable. The two instructions can be data-dependent on each other in two independent situations: (1)

iload is data dependent on *istore* if they refer to the same local variable, and its value read by *iload* was written by *istore* via the variable; (2) *istore* is data dependent on *iload* if they refer to the same stack cell, and its value read by *istore* was written by *iload* via JVM stack. The data flow between stack and local variable for each situation is shown in Figure 3.

Situation#1: *iload* is data dependent on *istore* via local variable					
Instruction execution	*istore* <var#1> store integer from stack to variable#1 *iload* < var#1> retrieve integer from variable#1 to stack				
Before		**After *istore* <var#1>**		**After *iload* < var#1>**	
Stack	Var#1	Stack	Var#1	Stack	Var#1
value	-	-	value	value	-

Situation#2: *istore* is data dependent on *iload* via JVM stack					
Instruction execution	*iload* < var#1> retrieve integer from variable#1 to stack *istore* <var#2> store integer from stack to variable#2				
Before		**After *iload* <var#1>**		**After *istore* < var#2>**	
Stack	Var#1	Stack	Var#1	Stack	Var#2
-	value	value	-	-	value

Figure 3 The data flow analysis between JVM stack cells and local variables in the two example situations.

In the JVM, as each new thread comes into existence, it gets its own program counter and JVM stack frame. Method execution state is stored on the stack. Each stack frame contains local variables, method parameters, the return value (if any) and intermediate calculations. In order to track both data-dependence situations, JavaPDG monitors the data flow via an *abstract variable* regardless of local variables or JVM stack, and determines the define-set *D* and use-set *U* of each instruction in terms of the values it writes to and reads from the abstract variable, respectively. Informally, we define two instructions as data dependent if they might reference the same abstract variable and one of the references is an assignment to the variable.

Take function *add*() in Figure 2 for example. When the function is called, the JVM allocates a stack frame and pushes the values of method arguments on the top of the stack. The instruction #0 *iload_0* retrieves the value of the first argument *a* onto stack and this value is then transferred into instruction #2 *iadd*; the similar procedure makes the value of the second argument *b* flow into the instruction #2 via the stack as well. *iadd* pops two ints from the stack, adds them, and pushes the int result back onto stack. This computational result is popped up and is finally returned by instruction #3 *ireturn*.

JavaPDG ignores the container where data is read from or written to but tracks the sender and receiver that uses the data; in the running example, the return value is data dependent on *ireturn* and further dependent on *iadd*. The instruction *iadd* is data dependent on both *iload_0* and *iload_1* which is transitively data dependent on input nodes node *a* and *b*, respectively.

3) Inter-Procedural Dependences

There are three types of dependences between methods [30]: 1) Method-call dependences that represent call relationships between a caller and the callee method; 2) Parameter-in dependences that represent parameter passing between actual-input parameters and formal-input parameter; 3) Parameter-out dependences representing parameter passing between a formal-output parameter (return value) and actual-output parameters. Thus, in the SDG, pDGs are associated with the three types of inter-procedural control and data dependence edges.

Refer again to Figure 2. The inter-procedural control-dependence edges connect call-site instructions #7 and #11 in the function *main()* to the entry vertex of the callee function *add*(). And there are four inter-procedural data-dependence edges starting from actual-input vertices in the caller and ending at their corresponding formal-input vertices in the callee; and two inter-procedural data-dependence edges associated formal-output vertices with actual-output vertices.

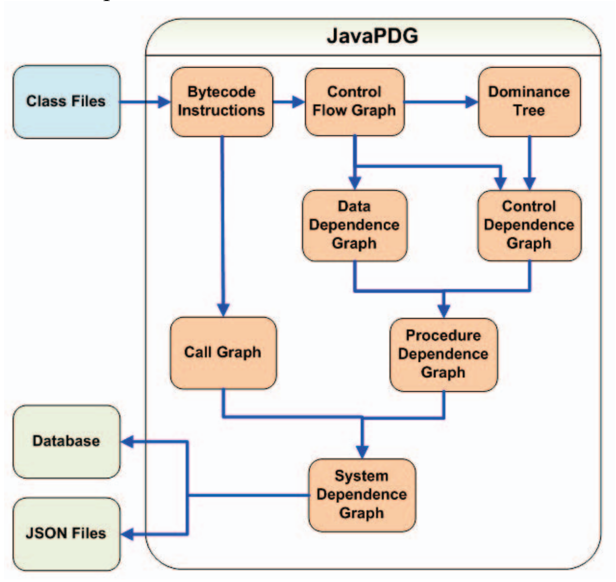
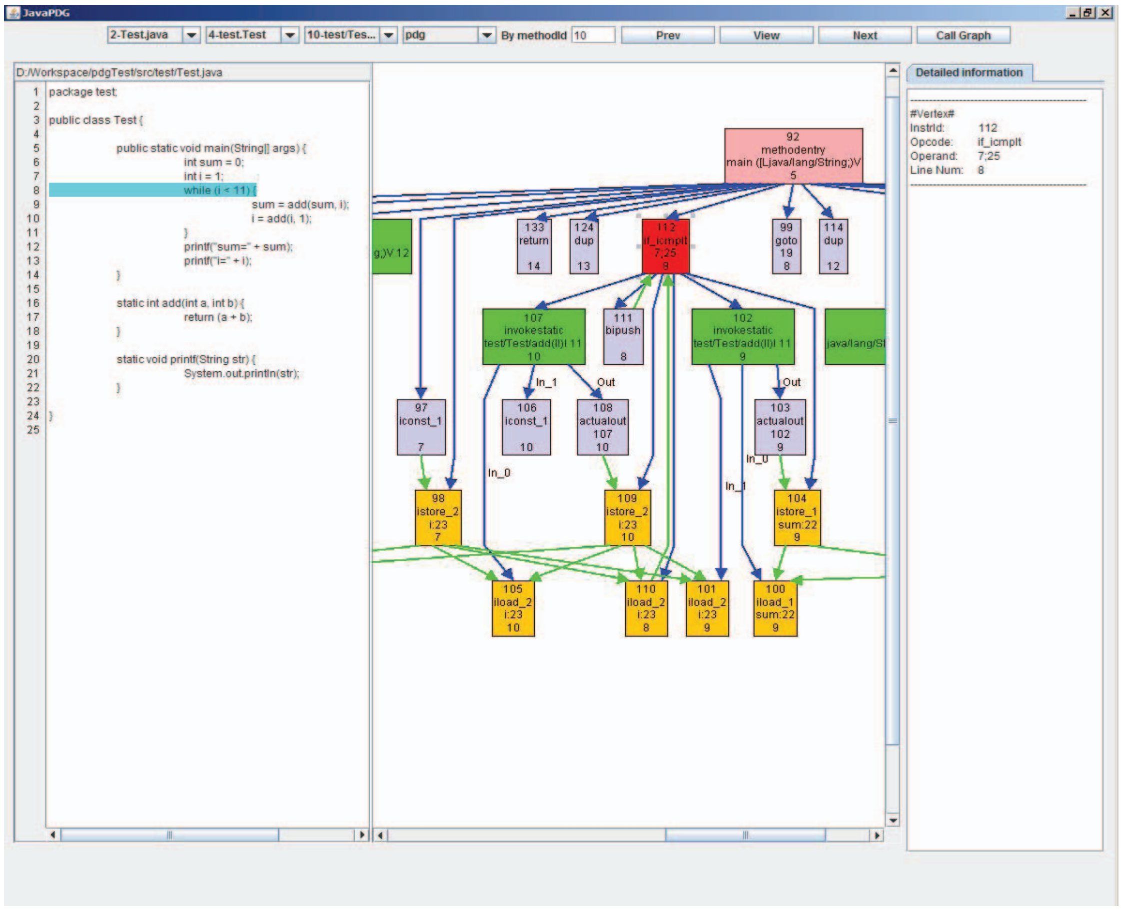


Figure 4 The JavaPDG Framework

B. *Framework*

The analysis process takes as input the compiled class files of a Java program, and yields a SDG and related graphs as the final output after performing the following steps as indicated in Figure 4:

(1) Preprocessing: In the SDG, one pDG vertex represents each instruction. Artificial entry and exit vertices for every method are added to the graph to represent the start and end of the method, respectively. A vertex is added for every call-site as its actual-output parameter if the callee method has any return value.



```
package test;

public class Test {

        public static void main(String[] args) {
                int sum = 0;
                int i = 1;
                while (i < 11) {
                        sum = add(sum, i);
                        i = add(i, 1);
                }
                printf("sum=" + sum);
                printf("i=" + i);
        }

        static int add(int a, int b) {
                return (a + b);
        }

        static void printf(String str) {
                System.out.println(str);
        }

}
```

```
----------------------------------------
#Vertex#
InstrId:     112
Opcode:      if_icmplt
Operand:     7;25
Line Num:    8
----------------------------------------
```

Figure 5 A Screenshot of JavaPDG Viewer

(2) Control Flow Analysis: The bytecode of each method is transformed into a *control flow graph* (CFG). This step makes the unstructured bytecode control flow explicit. The CFG is the basis for construction of a pDG. Dependence analysis involves two steps: control-dependence analysis and data-dependence analysis. Conventionally, both are based upon the CFG.

(3) Control Dependence Analysis: A *dominance tree* (DT) is computed for each CFG. Informally, control dependent nodes come after a decision and before a junction on a CFG. A DT characterizes the topological ordering in a flow-graph and is therefore used to identify control dependences on the CFG. Having CFG and DT, a *control dependence graph* (CDG) for every method is derived. In the CFG, if there is a path from vertex x to y that doesn't contain the immediate forward dominator of x, y is control dependent on x.

(4) Data Dependence Analysis: A *data dependence graph* (DDG) for every method is calculated by tracking data flows on its CFG. A data flow is usually represented by a definition-use chain, i.e., one instruction assigns a value to an abstract variable and the other instruction uses the value. Reaching-definition and upward-exposed-uses analyses are conducted following the steps: 1) analyze the effect of each instruction in terms of its variable definition and use sets; 2) iteratively propagate the information over the CFG; 3) during each iteration, inspect whether there is any unknown definer/assigner of the variable(s) used in each instruction, and update its information sets accordingly; 4) once the information propagation ends (no changes are found), the data dependences between instructions is calculated by the definition-use chain analysis.

(5) Inter-procedural Analysis: An SDG is a collection of interconnected pDGs, each of which is composed of the CDG and DDG for a method. The static call graph of a program is used to investigate communications between methods. Based on the call graph, three types of inter-procedural control and data dependences are computed.

(6) The Output: The output SDG is a labeled, directed graph consisting of multiple pDGs. Besides the SDG, JavaPDG outputs some additional information, including

- Static structure of a program that describes classes, fields, methods, and relationships among them.
- Variable information that contains the name, type and scope of every class field, object field and local variable (including formal input parameter).
- Control flow graphs and dominance trees that are constructed during dependence analysis and share the same vertices as in the SDG.

- A static call graph whose vertices correspond to Java methods and whose edges represent potential caller-callee relationships indicated in the program.

IV USING JAVAPDG

For a Java program, JavaPDG automates dependence analysis from bytecode to SDG. JavaPDG maintains an embedded Apache Derby (or Java DB) database, which stores the intermediate data and output. One database is created for a subject program. The results can be also exported to JSON format. JavaPDG thus enables researchers to extend and customize the produced SDG to meet their research-specific needs.

A graphical viewer is included in JavaPDG that allows users to browse the program dependence graphs (or call graph) and inspect the source code. In Figure 5, a screenshot of the viewer is shown. The tool can display not only pDG for single method but also its CFG, DT, CDG and DDG for separately reviewing. The static call graph for the whole program can be also visualized in the viewer. In Figure 5, the left side highlights a Java method in its source file and the right side shows pDG vertices associated by intra-procedural control-dependence edges (blue) and data-dependence edges (green).

V RELATED WORK

A. *PDG-based Research*

PDG, because of its usefulness, has become the focus of recent research on software testing and debugging. CodeSurfer, the commercial PDG-based analyzer, supports most of the following studies on C/C++ programs. Chang et al [12, 13] showed how to use PDG as a general representation of programming rules. They then employed a frequent subgraph mining algorithm on SDG to find programming rules, and used a graph matching algorithm to find rule violations. Sun et al [14, 16] explored how to propagate bug fixes using fast subgraph matching on PDGs transformed from a subject program. In a further extension, Sun et al [15] presented an approach to discover and transform project-specific rules from PDGs into checkers for Klocwork, a commercial static analysis tool. Klocwork can then be utilized to find rule violations. Acharya and Robinson [17] proposed a framework for change impact analysis using PDG-based slicing for large and evolving industrial software systems. Leitner et al [31] presented a test case minimization method based on static program slicing. It starts from the failure instruction, and proceeds backwards by following data dependences to identify a minimized, subset of the code consisting only of those instructions that might affect the failure instruction. Krinke [20] developed an approximation approach to identify similar code in programs by finding isomorphic subgraphs in PDGs. Nagy and Mancoridis [32] proposed an approach to locate faults that are related to security by conducting dataflow analysis on PDGs to identify parts of the source code that involve user input. Shu et al [33] developed a technique to build a causal graph based on a program's dynamic call graph and inter-method DDG for locating faulty methods in large and complex software using causal inference methodology. Baah et al [34] showed how PDG can be used to construct a probabilistic graphical model that captures the statistical dependences among program elements and enables the use of probabilistic reasoning to analyze program behaviors.

B. *Dependence Analysis Framework for Java Bytecode*

Soot [8] is a widely used optimization frameworks for the static analysis on Java bytecode, and provides four intermediate representations for analyzing and transforming Java bytecode. Soot has been evolving for more than a decade and it provides some useful APIs for program dependence analysis. However, one important feature that Soot has been lacking is the implementation of an inter-procedural program analysis framework.

Wala [9] is another well-known framework for static analysis on Java bytecode. Wala extracts *static single assignment* (SSA) representation from Java bytecode and is able to conduct iterative dataflow analysis, pointer analysis and call graph construction on it. A prototype SDG slicer has been provided in its recent versions. However, its basic PDG vertex represents the SSA instruction rather than Java bytecode instruction as used in JavaPDG.

Indus [35] is a non-PDG based program slicer in which Java programs are represented in Jimple [36] via Soot. Instead of maintaining inter-procedural dependence edges in a SDG, the logic to handle them (including unconditional jumps, procedure calls, etc.) is encoded in the Indus's slicing algorithm.

C. *Java System Dependence Graph*

There have been a number of modifications proposed to traditional SDG for the representation of Java programs. However, Java is a growing language and grammar changes may affect the generality of grammar-dependent program slicers, therefore the following publications actually have not resulted in general tools. Kovács et al.'s approach [37] is able to represent some Java-specific features such as inheritance, packages, interfaces and polymorphic calls. Zhao [30] used a group of dependence graphs to represent Java methods, classes, interfaces, programs, and packages, respectively. Chambers et al. [38] proposed an approach that can accurately analyze data dependences in Java programs on the occurrences of exceptions, synchronization and memory consistency. Grove et al. [39] gave an approach to draw out call graph for object-oriented programs. Liang and Harrold [40] improved precision in inter-procedural slicing for Java programs by making the dependence analysis partially object-sensitive. Walkinshaw et al. [41] combined and adapted the earlier approaches for multiple dependence representations, and provided a guidance for Java system dependence graph constructions. These methods provide a theoretical basis for dependence representations of Java programs.

VI CONCLUSION

JavaPDG is a static analysis platform that constructs system dependence graph from Java bytecode. It aims at facilitating PDG-based research as there does not exist a SDG analysis tool directly built on Java bytecode. It stores intermediate data and graph-based output in a built-in database. Its portable, project-specific database design and user-friendly graphical viewer provides users with a convenient way to explore the secrets behind the PDGs.

Future versions of JavaPDG will consider additional sources of dependences such as exceptions. When an exception is thrown, execution of the method terminates immediately and the control transfers to another location, possibly non-local to the method where the exception was thrown. Exceptions may be thrown explicitly or implicitly by instructions and introduce new dependences into the program.

Another area of future work is representation. Currently, the resulting SDG has nodes which represent individual bytecode instructions. Similar graphs can be built at more abstract levels such as basic blocks allowing the analyst to get a better overview of the dependence structures in the application.

VII ACKNOWLEDGMENTS

We would like to thank ABB Corporate Research for supporting this work. We also thank the National Science Foundation for supporting this work with awards CCF-0820217 and CNS-1035602.

VIII REFERENCES

1. F. Tip, "A Survey of Program Slicing Techniques," Journal of Programming Languages, vol. 3(3), pp.121-189, 1995.
2. M. Weiser, "Program slicing," the 5th *Intl. Conf. on Software Eng.(ICSE),* San Diego, California, 1981.
3. D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. "Dependence Graphs and Compiler Optimizations," the 8th *ACM Symp. on Principles of Programming Languages*, pp. 207–218, 1981.
4. H. Agrawal and J.R. Horgan. "Dynamic Program Slicing," *the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation*, pp. 246–256, 1990. SIGPLAN Notices 25(6).
5. W. Masri and A. Podgurski. "Algorithms and Tool Support for Dynamic Information Flow Analysis," *Information and Software Technology*, vol. 51(2), pp.385–404, 2009.
6. P. Anderson and T. Teitelbaum. "Software Inspection Using Codesurfer," the 1st *Workshop on Inspection in Software Engineering*, 2001.
7. L. Andersen. "Program Analysis and Specialization for the C Programming Language," *PhD thesis, Department of Computer Science, University of Copenhagen*, May 1994.
8. R. Vall´ee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. "Soot-A Java Optimization Framework," *the 1999 Conf. of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*, pp. 125–135, 1999.
9. *WALA: T.J. Watson Libraries for Analysis*. Available from: http://wala.sourceforge.net/wiki/index.php/Main_Page.
10. J. Ferrante, K.J. Ottenstein, and J.D. Warren. "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9(3), pp.319–349, 1987.
11. S. Horwitz, T. Reps, and D. Binkley. "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12(1), pp.26–61, 1990.
12. R. Y. Chang, A. Podgurski, and J. Yang. "Finding What's not There: A New Approach to Revealing Neglected Conditions in Software," *the Intl. Symp.on Software Testing and Analysis (ISSTA)*, London, United Kingdom, 2007.
13. R. Y. Chang, A. Podgurski, and J. Yang. "Discovering Neglected Conditions in Software by Mining Dependence Graphs," *IEEE Transactions on Software Engineering*, vol. 34(5), pp.579–596, 2008.
14. B. Sun, G. Shu, A. Podgurski, S. Li, S. Zhang, and J. Yang, "Propagating Bug Fixes with Fast Subgraph Matching," the 21st *Intl. Symp. on Software Reliability Engineering (ISSRE)*, San Jose, CA, 2010.
15. B. Sun, G. Shu, A. Podgurski, and B. Robinson. "Extending Static Analysis by Mining Project-Specific Rules," the 34th *Intl. Conf. on Software Eng. (ICSE)*, Switzerland, 2012.
16. B. Sun, R. Y. Chang, X. Chen, and A. Podgurski, "Automated Support for Propagating Bug Fixes," the 19th *Intl. Symp. on Software Reliability Eng. (ISSRE)*, Seattle, Washington, 2008.
17. M. Acharya and B. Robinson, "Practical Change Impact Analysis based on Static Program Slicing for Industrial Software Systems," the 33rd *ACM SIGSOFT Intl. Conf. on Software Eng. (ICSE)*, Waikiki, Honolulu, Hawaii, 2011.
18. X. Qu, M. Acharya, and B. Robinson. "Impact Analysis of Configuration Changes for Test Case Selection," the 22nd *Intl. Symp. on Software Reliability Engineering (ISSRE)*, Hiroshima, Japan, 2011.
19. R. Komondoor and S. Horwitz. "Using Slicing to Identify Duplication in Source Code," the 8th *Intl. Static Analysis Symp. (SAS)*, 2001.
20. J. Krinke. "Identifying Similar Code with Program Dependence Graphs," the 8th *Working Conf. on Reverse Eng.*, 2001.
21. C. Liu, C. Chen, J. Han and P. S. Yu. "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis," the 12th *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06)*, Philadelphia, 2006.
22. J. Zhao. "Dependence Analysis of Java Bytecode," the 24th *IEEE Annual Intl. Computer Software and Applications Conference*, pp. 486-491, 2000.
23. *Apache Derby version v10.9.1.0*. 2012; Available from: http://db.apache.org/derby/.
24. F. Yellin, and T. Lindholm. "The Java Virtual Machine Specification," *Addison-W esley*, 1996.
25. M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. "An Overview of the Scala Programming Language," *Technical Report, EPFL*, Lausanne, Switzerland, 2004.
26. R. Hickey. "The Clojure Programming Language," the 2008 *Symp. on Dynamic Languages, (DLS '08)*, New York, NY, 2008.
28. J. Zhao. "Analyzing Control Flow in Java Bytecode," the 16th *Conf. of Japan Society for Software Science and Technology*, pp. 313–316, 1999.

29. S. S. Muchnick. "Advanced Compiler Design and Implementation," *Morgan Kaufmann*, 1997.
30. J. Zhao. "Applying Program Dependence Analysis to Java Software," *Workshop on Software Eng. and Database Systems, 1998 International Computer Symp.*, 1998.
31. A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. "Efficient Unit Test Case Minimization," the 22nd *Intl. Conf. on Automated Software Eng.*, Atlanta, Georgia, 2007.
32. C. Nagy, and S. Mancoridis, "Static Security Analysis based on Inputrelated Software Faults", the 13th *European Conference on Software Maintenance and Reengineering (CSMR'09)*, Kaiserslautern, Germany, March, 2009.
33. G. Shu, B. Sun, A. Podgurski, and F. Cao. "MFL: Method-Level Fault Localization with Causal Inference," the 6th *Intl. Conf. on Software Testing, Verification and Validation (ICST)*, Luxembourg, 2013.
34. G. K. Baah, A. Podgurski, and M. J. Harrold. "The Probabilistic Program Dependence Graph and its Application to Fault Diagnosis," *the Intl. Symp.on Software Testing and Analysis (ISSTA)*, Seattle, WA, 2008.
35. G. Jayaraman, V. P. Ranganath, and J. Hatcliff. "Kaveri: Delivering the Indus Java Program Slicer to Eclipse," the *Fundamental Approaches to Software Eng. (FASE'05)*, LNCS, vol. 3442, pp. 269–272. Springer, 2005.
36. R. Vallee-Rai and L. J. Hendren. "Jimple: Simplifying Java Bytecode for Analyses and Transformations," *Technical report, McGill University*, Montreal, Canada, 1998.
37. G. Kovács, F. Magyar, and T. Gyimóthy. "Static Slicing of Java Programs," *Technical Report TR-96-108, Jozsef Attila University*, Hungary, 1996.
38. C. Chambers, I. Pechtchanski, V. Sarkar, M. J. Serrano, and H. Srinivasan, "Dependence Analysis for Java," the 12nd *Intl. Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA, 1999.
39. D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call Graph Construction in Object-Oriented Languages," *ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, Atlanta, GA, 1997.
40. D. Liang and M. J. Harrold. "Slicing Objects using System Dependence Graphs," *the Intl. Conf. on Software Maintenance*, pp. 358–367. IEEE Computer Society, 1998.
41. N. Walkinshaw, M. Roper, and M. Wood. "The Java System Dependence Graph," the 3rd *IEEE Intl. Workshop on Source Code Analysis and Manipulation*, pp. 55, 2003.

APPENDIX: DEMO DESCRIPTION

The purpose of this demo is to show the main features of our static program dependence analysis tool JavaPDG. We have implemented JavaPDG in Java, which can be run in standalone mode within Java Runtime Environment. The user interacts through the command-line interface of JavaPDG that has three main modules and we plan to run the demo by letting the audience go through each of them to experience the easiness and usefulness of the tool. Next we describe how we plan to run the demo.

A. *Module#1: Program Dependence Analyzer*

User can run the program dependence analyzer from the command line by setting the value 1 to option '-mt' which indicates module type. Other necessary options to this module includes (1) the option '-pc': full path of the base folder where class files are located, and (2) the option '-da': database connection URL which means the name and port number of the server hosting database and the name of the database connect to. The user can also change the default user name (using option '-du') and password (using option '-dp') of the database as well as assign name (with option '-pn') and version (with option '-pv') of the subject program for its own customization.

Having these options specified, the user can activate the dependence analysis so that the module ingests Java bytecode and proceeds with the whole-program analysis as explained in the Section III.

B. *Module#2: Graph Viewer*

Once the analysis finishes and the results are stored into a database, the user is able to browse and analyze the various graphs using a built-in graph viewer by setting the value 2 to the option '-mt'. Options '-da' and '-pj' are required to this module as well. The option '-da' stands for the same meaning as in the dependence analyzer and the option '-pj' specifies the full path of the base folder where Java source files are located. Additional options may be needed if the default values of database username and password were changed.

As an interactive graph viewer starts up, user can choose to see to any of the graphs produced. First, the user can click on a button named 'call graph' on the upper right corner of Figure 5 to visualize static call graph for the program. Second, user may select an interesting Java method from the cascading drop-down menus like the one on the top of Figure 5. This menu list contains three menus: 'file', 'class' and 'method'. One source file may contain multiple Java classes, and a class possibly has several methods. Each time the selection of the one in a menu changes, its cascading menu is updated. Note that the 'method' menu may not contain any implemented method if the item selected from the 'class' menu is an interface. Once chosen method changes, a field labeled 'By MethodId' that displays its method id changes accordingly. The user can also specify a method id in the field instead. In the drop-down menu 'graph type', various types are available like pDG, CFG, DT, CDG and DDG. Having method id and graph type selected, user can click on the button 'View' to visualize the graph on the right-side graph panel. The buttons 'Prev' and 'Next' are provided in case anyone needs to browse neighboring methods with the same graph type. The corresponding source file is loaded up on the left-side panel at the same time. When user click on a vertex or an edge on the graph panel, its detail is displayed on 'Detailed Information' panel and its line in the source code can be highlighted.

C. *Module#3: JSON Format Exporter*

JSON is so much more lightweight and less verbose preferably by some applications. With specifying option '-mt' to be 3 and appropriately setting the option '-da' and option '-js' (the full path of output folder), the module enables to export data from database into JSON files.