APPLYING PROGRAM DEPENDENCE ANALYSIS TO JAVA SOFTWARE

Jianjun Zhao

Department of Computer Science and Engineering

Fukuoka Institute of Technology

3-10-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-0214, Japan

zhao@cs.fit.ac.jp

ABSTRACT

Understanding program dependences is essential for many software engineering activities including program slicing, testing, debugging, reverse engineering, maintenance, and complexity measurement. This paper presents the software dependence graph for Java (JSDG), which extends previous dependence-based representations, to represent various types of program dependences in Java software. The JSDG consists of a group of dependence graphs which can be used to represent Java methods, classes and their extensions and interactions, interfaces and their extensions, complete programs, and packages respectively. The JSDG can be used as an underlying representation to develop software engineering tools for Java software.

1 INTRODUCTION

Java is a new object-oriented programming language and has achieved widespread acceptance because it emphasizes portability. As Java commercial applications are going to be accumulated, the development of tools to support understanding, testing, maintenance, complexity measurement of Java software will become an important issue.

Program dependences are dependence relationships holding between program elements in a program that are determined by the control flows and data flows in the program. Intuitively, if the computation of a statement directly or indirectly affects the computation of another statement in a program, there might exist some program dependence between the statements.

Many compiler optimizations and program testing and analysis techniques rely on program dependence information, which is topically represented in the form of a *program dependence graph* (PDG) [8, 14]. The PDG, although originally proposed for compiler optimizations, has been applied to various software engineering activities including program slicing, debugging, testing, maintenance, and complexity measurements for procedural programs [1, 7, 12, 17, 18]. Recently, researchers have applied program dependence analysis to object-oriented software [15, 16, 5, 6, 23] (for detailed discussions, see related work section) as well as software architectures [24]. However, although a number of dependence-based representations have been proposed for modeling various object-oriented features such as classes and objects, class inheritance, polymorphism and dynamic binding in object-oriented software, until recently, no dependence-based representation has been proposed which can be used to represent some specific features such as Java interfaces and their extensions, packages, and exception handling in Java software.

One of the best feature of Java is that it has elevated interfaces to first class status. An *interface* consists only of abstract methods and constants that define some functionality. An interface is a type and you can define variables to have such a type. Interfaces are implemented with classes. And an interface can be implemented several times with different classes. A variable of interface type can hold a reference to an object of any of the classes that implement that interface. Like classes, interfaces may also be extended.

In Java, code is collected into packages. A packages can be used to create a grouping for related interfaces and classes. Interfaces and classes in a package can use popular public names that make sense in one connect but might conflict with the same name in another package. Packages can have types and members that are available only within the package. Such identifiers are available to the package code, but inaccessible to outside code.

In order to represent the full range of Java software, an efficient dependence-based representation must be able to facilitate the analysis of these features such as interfaces and their extensions, and packages in Java software.

In this paper we present the *software dependence graph for Java* (JSDG), which extends previous dependence-based representations [15], to represent various types of program dependence relationships in Java software. The JSDG of Java software consists of a group of dependence graphs which can be used to represent Java methods, classes and their extensions and interactions, interfaces and their extensions, complete programs, and packages respectively. The JSDG can be used as an underlying representation to develop software engineering tools for Java software.

The rest of the paper is organized as follows. Section 2 introduces various types of program dependences that may exist in Java software. Section 3 presents the software dependence graph for Java. Section 4 discusses some applications of the JSDG. Section 5 discusses some related work. Concluding remarks are given in Section 6.

2 PROGRAM DEPENDENCES IN JAVA SOFTWARE

To perform program dependence analysis on Java software, it is necessary to identify all primary dependence relationships existing in Java software. In this section, we present various types of primary program dependence relationships which may exist in Java software.

2.1 Control and Data Dependences

There are two types of primary program dependences between statements in a Java method, i.e., control dependence and data dependence.

- *Control dependences* represent control conditions on which the execution of a statement or expression depends in a single method. Informally, a statement u is directly control-dependent on the control predicate v of a conditional branch statement (e.g., an if statement, switch statement, while statement, for statement, or do-while statement) if whether u is executed or not is directly determined by the evaluation result of v.

- *Data dependences* represent the data flow between statements in a single method. Informally a statement u is directly data-dependent on a statement v if the value of a variable computed at v has a direct influence on the value of a variable computed at u.

2.2 Call and Parameter Dependences

There are three types of primary program dependences between a call and a called method in Java classes.

- *Method-call dependences* represent call relationships between a call method and the called method. Informally a method u is method-call dependent on another method v if v invokes u.

Parameter dependence model the parameter passing between a call and a called method. There are two types of parameter dependences, i.e., parameter-in dependence and parameter-out dependence.

- *Parameter-in dependences* represent parameter passing between actual parameters and formal input parameter (only if the formal parameter is at all used by the called procedure).

- *Parameter-out dependences* represent parameter passing between formal output parameters and actual parameters (only if the formal parameter is at all defined by the called procedure). In addition, for methods, parameter-out dependences represent the data flow of the return value between the method exit and the call site.

2.3 Membership Dependences

There are four types of membership dependences in Java software.

- *Class-membership dependences* capture the membership relationships between a class and its member methods. Informally, a method u is class-membership dependent on class v if u is a member method of v.

- *Interface-membership dependences* capture the membership relationships between an interface and its member method declarations. Informally, a method declaration u is interface-membership dependent on an interface v if u is a member method declaration of v.

- *Package-membership dependences* capture the membership relationships between a package and its member classes, interfaces, and subpackages. Informally, a class/interface/subpackage u is package-membership dependent on a package v if u is a member class/interface/subpackage of v.

2.4 Inheritance Dependences

There are two types of inheritance dependences in Java software to represent Java class extensions and interface extensions. Note that unlike C++, Java does not support *friend* relationships among classes.

- *Class-inheritance dependences* capture the inheritance relationships between Java classes. Informally, a class u is class-inheritance dependent on class v if u is an extending class of v.

- *Interface-inheritance dependences* capture the inheritance relationships between Java interfaces. Informally, an interface u is interface-inheritance dependent on an interface v if u is an extending interface of v.

3 THE SOFTWARE DEPENDENCE GRAPH FOR JAVA

In this section, we show how to construct the *software dependence graph for Java* (JSDG for short). The JSDG of Java software consists of a collection of dependence graphs which can be used to represent Java methods, classes and their extensions and interactions, interfaces and their extensions, complete programs, and packages respectively. It basically takes advantage of constructing techniques of previous dependence-based representations [10, 15].

3.1 Dependence Graphs for Java Methods

A method of a Java class is similar to a procedure in conventional procedural languages. Thus it is reasonable to use the usual procedure dependence graph introduced in [10] to represent a single method in a Java class. In contrast to the procedure dependence graph, we call such a graph the *method dependence graph* (MDG for short). The MDG of a method is an arc-classified digraph whose vertices are connected by several types of dependence arcs. The vertices of the
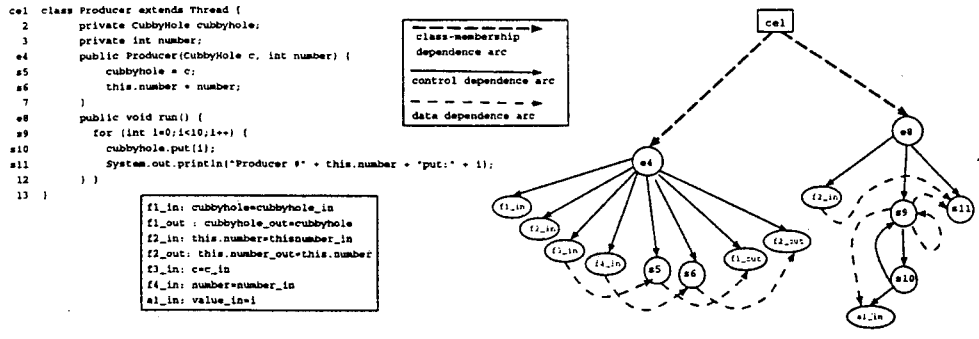
Figure 1: A Java class and its CDG methods

MDG represent statements or control predicates of conditional branch statements in the method. There is an unique vertex called *method start vertex* to represent the entry of the method. In order to model parameter passing between methods, an MDG also includes formal parameter vertices and actual parameter vertices. At the method entry there is a *formal-in vertex* for each formal parameter of the method and a *formal-out vertex* for each formal parameter that may be modified by the method. At each call site there is an *actual-in vertex* for each actual parameter at call site and an *actual-out vertex* for each actual parameter that may be modified by the called method. In addition, at each call site of the method, a *call vertex* is created for connecting the called method.

The arcs of the MDG represent two types of dependence relationships in a method, i.e., *control dependences*, and *data dependences*. There is a control dependence arc between two vertices u and v if u is control dependent on v, and there is a data dependence arc between two vertices u and v if u is data dependent on v. In addition, each formal parameter is control dependent on the method start vertex, and each actual parameter is control dependent on the call statement.

Example. Figure 1 shows a Java method run and its method dependence graph.

3.2 Dependence Graphs for Java Classes

This section describes how to construct dependence graphs for Java single classes, class extensions and interactions. The section also discusses how to represent polymorphism.

Single Classes

we use the *class dependence graph* (CDG for short) to represent a single Java class. The CDG of a Java class is an arc-classified digraph which consists of a collection of method dependence graphs each representing a single method in the class, and some additional vertices and arcs to model parameter passing between different methods in a class. There is an unique *class start vertex* for the class to represent the entry of the class, and the class start vertex is connected to the method start vertex of each method in the class by

class-membership dependence arcs. If a method invokes another method in the class, the method dependence graphs of two methods are connected at call site. In such a case, a call dependence arc is added between a call vertex of a method and the method start vertex of the method dependence graph of the called method, and *parameter dependence arcs* are added to connect actual-in and formal-in vertices, and formal-out and actual-out vertices to model parameter passing between the methods in the class. Unlike C++, Java does not support global variables. However, the instance variables of a Java class are accessible to all methods in the class, and therefore we can regard them as "global variables" to every method in the class, and create formal-in and formal-out vertices for all instance variables that are referenced in the methods.

In [10], interprocedural slices are computed by solving a graph reachability problem on an SDG. To obtain precise slices, the computation of a slice must preserve the calling context of called procedures, and ensure that only paths corresponding to legal call/return sequences are considered. To facilitate the computation of interprocedural slicing that considers the call context, an SDG represents the flow dependences across call sites. A *transitive flow of dependence* occurs between an actual-in vertex and an actual-out vertex if the value associated with the actual-in vertex affects the value associated with the actual-out vertex. The transitive flow of dependence can be caused by data dependences, control dependences, or both. A *summary* arc models the transitive flow of dependence across a procedure call. Similar to [10], we use *summary dependence arcs* to represent this kind of transitive flow of dependences in the class dependence graph.

Example. Figure 1 shows the CDG for a Java class Producer. In the figure, a rectangle represents the class start vertex and is labeled by the statement label related to the class entry. Circles represent statements in the class, including method start, and are labeled with the corresponding statement number in the class. Ellipses in dashed line represent actual parameter vertices and Ellipses in solid line represent formal parameter vertices. For example, ce1 is the class start vertex, and e4 and e8 are the start vertices of methods Producer and run. Bold dashed arcs represent

class-membership dependence arcs that connect the class start vertex to each start vertex of the methods. Therefore, $(ce1, e4)$ and $(ce1, e8)$ are class-membership dependence arcs. Each start vertex of the methods is the root of a subgraph which is itself a method dependence graph corresponding to the method. Hence each subgraph may contain control, data and parameter dependence arcs. Finally, constructor method Producer has no formal-in vertices for the two instance variables cubbyhole and number, since these variables cannot be referenced before they are allocated by the class constructor.

Class Interaction

In Java, a class may create an object of another class through a declaration or by using an operator such as new. When a class $c1$ creates an object of class $c2$, there is an implicit call to $c2$'s constructor. To represent this implicit constructor call, we adds a call vertex in $c1$ at the location of object creation. A call dependence arc connects this call vertex to $c2$'s constructor method. We also adds actual-in and actual-out vertices at the call vertex to match the formal-in and formal-out vertices in $c2$'s constructor. When there is a call site in method m_1 in c_1 to method m_2 in the public interface of c_2, we connect the call vertex in c_1 to the method start vertex of m_2 to form a call dependence arc, and also connect actual-in and formal-in vertices to form parameter-in dependence arcs and actual-out and formal-out vertices to form parameter-out dependence arcs. As a result, we can get a new CDG which represents a partial Java program by connecting these two CDGs.

Example. Figure 3 shows the representation of interaction classes. In the main class of the program, there is a statement $s51$ which instantiates an object of type Producer. The construction includes adding actual-in and actual-out vertices at call vertex for $s51$ to match the formal-in and formal-out vertices associated with $e4$ which is the method start vertex of constructor Producer, and connecting the call vertex for $s51$ to the method start vertex for $e4$ to form a call dependence arc, actual-in and formal-in vertices to form parameter-in dependence arcs and actual-out and formal-out vertices to form parameter-out dependence arcs.

Class Extending

One of the major benefits of object orientation is the ability to *extend* the behavior of an existing class and continue to use code written for the original class. When you extend a class to create a new class, the new extended class *inherits* all the fields and methods of the class that was extended. Unlike C++ that supports multiple inheritance, Java only supports single inheritance, i.e., a new class can extend exactly one superclass. This, among other things, can certainly simplify the analysis of Java class extension. However, Java provide a novel mechanism called *interface* to support multiple inheritance (we will introduce it in the next section). Just as class inheritance permits code reuse in Java software, a dependence-based representation for extended classes should also reuse the analysis

information.

To construct a CDG for an extended class, we first construct the representation for each method defined by the extended class, and then reuse the representations of all methods that are inherited from superclasses. There is a class entry vertex for the extended class, and the class-membership dependence arcs are used to connect this class entry vertex to the method entry vertex of each method in the definition of the extended class. The class-membership dependence arcs are also used to connect the class entry vertex to the method entry vertices of any methods defined in the superclass that are inherited by the extended class. Formal-in vertices for a method represent the method's formal parameters and instance variables in the extended or superclass that may be referenced by a call to this method. Similarly, formal-out vertices for a method represent the method's formal parameters and instance variables in super or extended classes that may be modified by a call to this method.

Polymorphism

Another important feature of object-oriented languages is *polymorphism*. In Java software, a polymorphic reference can, over time, refer to instances of more than one class. As a result, the static representation should represent this dynamic feature in Java software.

In this paper, a CDG represent such polymorphic method calls by using a way that all possible destinations of a method call are included in the representation, unless the type can be determined statically. We use a *polymorphic choice* vertex similar to that in [15] to represent the possible destinations of the polymorphic call in the CDG. A polymorphic choice vertex represents the selection of a particular call given a set of possible destinations.

3.3 Dependence Graphs for Java Interfaces

This section describes how to construct dependence graphs for Java single interfaces and interface extensions.

Single Interfaces

We use the *interface dependence graph* (IDG for short) to represent a Java interface and its corresponding classes that implement it. The IDG of a Java interface is an arc-classified digraph which consists of a collection of method dependence graphs each representing a single method in a class that may implement a method declaration declared in the interface, and some additional vertices and arcs to model parameter passing between different methods in a class.

There is an unique vertex called *interface start vertex* for the entry of the interface. Each method declaration in the interface can be regraded as a call to its corresponding method in a class that implement it, and therefore a call vertex is created for each method declaration in the interface. The interface start vertex is connected to each call vertex of the method declaration by *interface-membership dependence arcs*. If there are more than one classes that implement the interface, we connect a method call in the interface to every corresponding method that implement it in the classes.
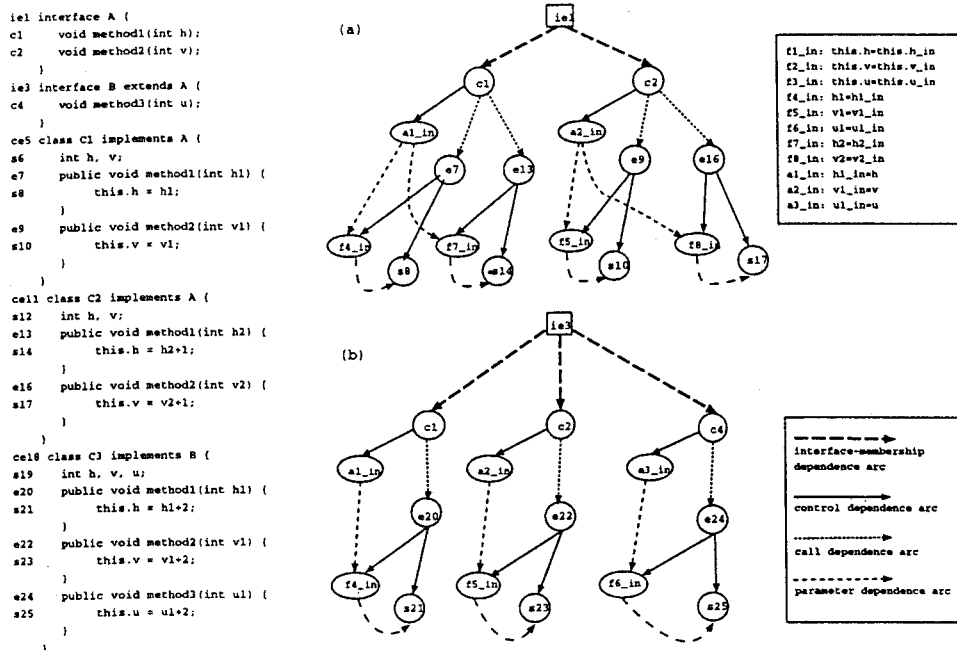
Figure 2: Two Java interfaces and Their IDGs.

Example. Figure 2 (a) shows the IDG for the interface A and the classes C1 and C2 that implement it.

Interface Extending

Similar to represent java class extending, we construct an IDG for an extended interface by constructing a representation for each method defined by the extended interface, and reusing the representations of all methods that are inherited from superinterfaces. We create an interface entry vertex for the extended interface, and add interface-membership dependence arcs connecting this interface entry vertex to the method call vertex of each method declaration in the definition of the extended interface. We also create interface-membership dependence arcs to connect the interface entry vertex to the method call vertices of any method declarations declared in the superinterface that are inherited by the extended interface.

Example. Figure 2 (b) shows the IDG for the interface B and the class C3 that implements it. The interface B is extended from the interface A, and therefore, class C3 should contain three methods, two for interface A, and one for interface B.

3.4 Dependence Graphs for Complete Java Programs

Generally, a complete Java program consists of classes and interfaces. In order to execute the program, the program must include a special class called main class. The program first starts the main class, and then transfers the execution to other classes.

We use the *system dependence graph* (SDG for short)

proposed in [15] to represent a complete Java program. The SDG of a complete Java program is an arc-classified digraph which consists of a collection of dependence graphs each representing a single method in the class, and some additional vertices and arcs to model parameter passing between different methods in a class.

To construct the dependence graph for a complete Java program, we first construct the class dependence graph for the main class, then connect the class dependence graph of the main class and other methods in other Java classes at call sites. A call dependence arc is added between a method call vertex and the start vertex of the method dependence graph of the called method. Actual and formal parameter vertices are connected by parameter dependence arcs,

Example. Figure 3 shows a complete Java program and its SDG.

3.5 Dependence Graphs for Java Packages

Java code is collected into packages. We use the *package dependence graph* (PADG for short) to represent a Java package. The PADG of a Java package is an arc-classified digraph which consists of a collection of dependence graphs each representing a Java class, interface, and subpackage, and some additional vertices and arcs to model the relationships among the package and its member classes, interfaces, and subpackages.

There is an unique vertex called *package start vertex* for the entry of the package. The package start vertex is connected to each class, interface, or subpackage start vertex of each class, interface, or subpackage by
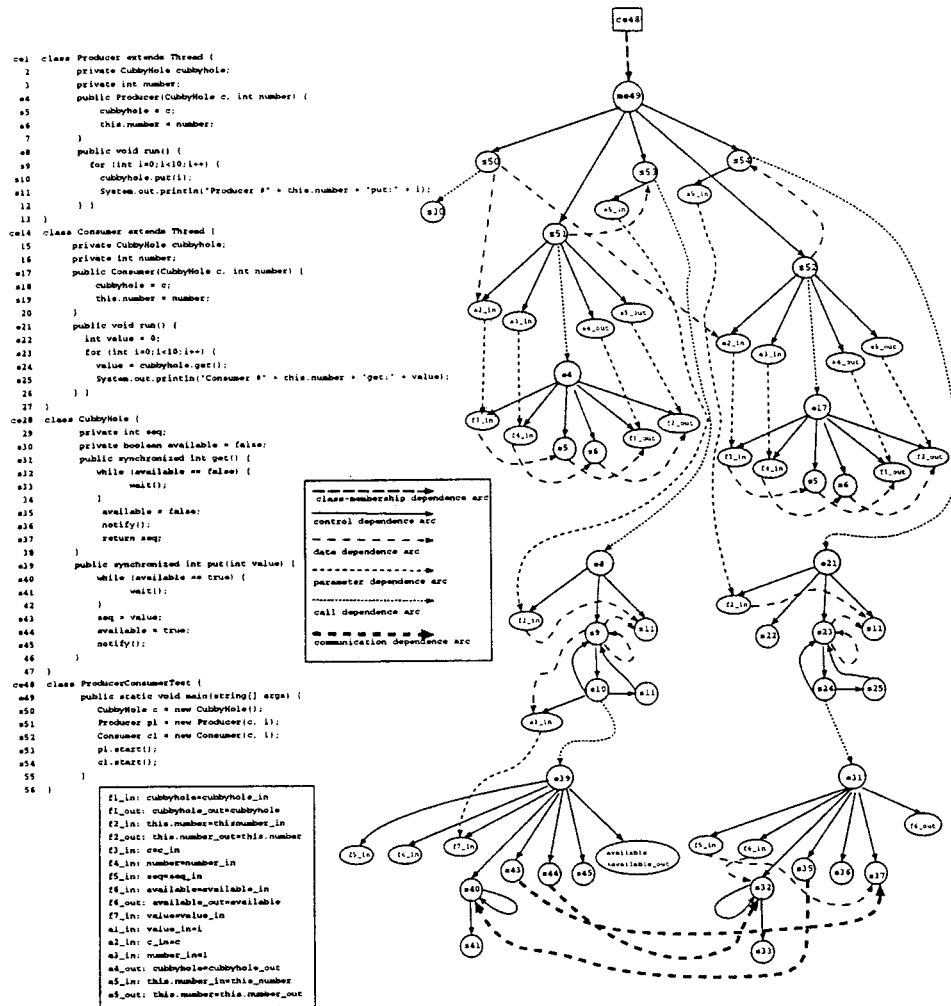
Figure 3: A complete Java program and its SDG.

package-membership dependence arcs.

Example. Figure 4 shows a simple package collections and its corresponding PADG. The package contains one interface A and two classes C1 and C2. p1 is the package start vertex and i2 is the interface start vertex of interface A. c3 and c4 are class start vertices of classes C1 and C2. Therefore, (p1,i2), (p1,c3), and (p1,c4) are package-membership dependence arcs.

4 APPLICATIONS OF THE JSDG

Having JSDG as an unified dependence-based representation for Java software, we discuss some important applications of the JSDG which include program slicing and software maintenance.

4.1 Program Slicing

The most direct application of JSDG is to slice Java software since the explicit representation of various program dependences in Java software makes the JSDG ideal for computing slices of a Java program [23].

Program slicing, originally introduced by Weiser [22], is a decomposition technique which extracts from program statements related to a particular computation. A *program slice* consists of those parts of a program that may directly or indirectly affect the values computed at some program point of interest, referred to as a *slicing criterion*. Program slicing has many applications in software engineering activities such as program understanding, debugging, testing, maintenance, reverse engineering, and complexity measurement. For more information, see Tip's survey on program slicing techniques [21].

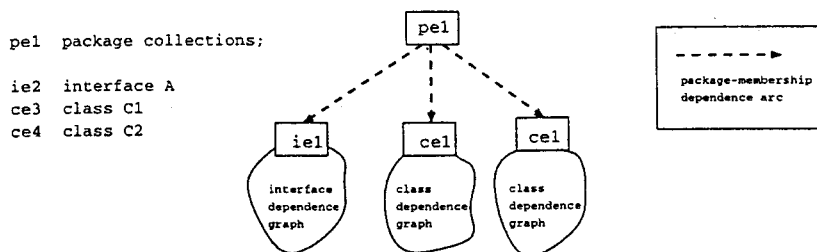In the following, we introduce some notions about

Figure 4: A simple package and its PADG

statically slicing of a complete Java program.

A *static slicing criterion* for a Java program is a tuple (s, v), where s is a statement in the program and v is a variable used at s, or a method call called at s. A *static slice* $SS(s, v)$ of a Java program on a given static slicing criterion (s, v) consists of all statements in the program that possibly affect the value of the variable v at s or the value returned by the method call v at s.

Since the SDG proposed for a complete Java program can be regarded as an extension of the SDGs for sequential object-oriented programs [15] and procedural programs [10], we can use the two-pass slicing algorithm proposed in [10, 15] to compute static slices of a Java program based on the JSDG. In the first step, the algorithm traverses backward along all arcs except parameter-out arcs, and set marks to those vertices reached in the SDG, and then in the second step, the algorithm traverses backward from all vertices having marks during the first step along all arcs except call and parameter-in arcs, and sets marks to reached vertices in the SDG. The slice is the union of the vertices of the SDG have marks during the first and second steps. Similar to the backward slicing described above, we can also apply the forward slicing algorithm [10] to the SDG to compute forward slices of Java programs.

In addition to slicing a complete Java program, we can also perform slicing on Java classes, interfaces, and packages based on the class dependence graphs, interface dependence graphs, and package dependence graphs.

4.2 Software Understanding and Maintenance

When we attempt to understand the behavior of a Java program, we usually want to know which variables in which statements might affect a variable of interest, and which variables in which statements might be affected by the execution of a variable of interest in the program. As discussed above, the slicing and forward-slicing based on the JSDG can satisfy these requirements. On the other hand, one of the problems in software maintenance is that of the ripple effect, i.e., whether a code change in a program will affect the behavior of other codes of the program. To maintain a Java program, it is necessary to know which variables in which statements will be affected by a modified variable, and which variables in which statements will affect a modified variable. The needs can be satisfied

by slicing and forward-slicing the program being maintained.

5 RELATED WORK

It is the first time, to our knowledge, to extend previous dependence-based representations to represent the full range of Java software.

Ferrante *et al.* [8] presented a dependence-based representation called the *program dependence graph* (PDG) to explicitly represent control and data dependences in a sequential procedural program with single procedure. Horwitz *et al.* [10] extended the PDG to introduce an interprocedural dependence-based representation called the *system dependence graph* (SDG) to represent a sequential procedural program with multiple procedures. Although these representations can be used to represent many features of a procedural program, they lack the ability to represent object-oriented features in Java software.

Larsen and Harrold [15] and Chan and Yang [5] extended the SDG for sequential procedural programs [10] to the case of sequential object-oriented programs. The SDGs they compute for sequential object-oriented programs can be regarded as a class of SDGs in [10]. Malloy *et al.* [16] proposed a new dependence-based representation called the *object-oriented program dependency graph* (OPDG) for sequential object-oriented programs. Chen *et al.* [6] also extended the program dependence graph to the *object-oriented dependency graph* (ODG) for modeling sequential object-oriented programs. Although these representations can be used to represent many features of sequential object-oriented programs, they lack the ability to represent some specific features such as interfaces and packages in Java software.

Zhao *et al.* [23, 25] presented a dependence-based representation called the *system dependence net* (SDN) to represent concurrent object-oriented programs. Although the SDN can be used to represent many object-oriented features as well as concurrency issues of concurrent object-oriented programs, it lacks the ability to represent some specific features such as interfaces and packages in Java software.

6 CONCLUDING REMARKS

We have presented, the *software dependence graph for Java* (JSDG), which is an extension of previous de-

pendence representations, to represent Java software. The JSDG consists of a group of dependence graphs which can be used to represent Java methods, classes and their extensions and interactions, interfaces and their extensions, complete Java programs, and Java packages respectively. The JSDG can be used to represent either object-oriented features or some specific features in Java software. We also discussed some applications of the JSDG which include program slicing and software maintenance. Now we are developing a maintenance environment for Java software in which the JSDG has been used as an underlying representation for developing software engineering tools including a program dependence analyzer and a program slicer.

References

[1] H. Agrawal, R. Demillo, and E. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software-Practice and Experience*, Vol.23, No.6, pp.589-616, 1993.

[2] S. Bates, S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs," *Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, pp.384-396, Charleston, South California, ACM Press, 1993.

[3] J. Beck, D. Eichmann, "Program and Interface Slicing for Reverse Engineering," *Proceeding of the 15th International Conference on Software Engineering*, pp.509-518, Baltimore, Maryland, IEEE Computer Society Press, 1993.

[4] J. M. Bieman, L. M. Ott, "Measuring Functional Cohesion," *IEEE Transaction on Software Engineering*, Vol.20, No.8, pp.644-657, 1994.

[5] J.T. Chan and W. Yang, "A program slicing system for object-oriented programs," *Proceedings of the 1996 International Computer Symposium*, Taiwan, December 19-21, 1996.

[6] J. L. Chen, F. J. Wang, and Y. L. Chen, "Slicing Object-Oriented Programs," *Proceedings of the APSEC'97*, pp.395-404, Hongkong, China, December 1997.

[7] J. Cheng, "Process Dependence Net of Distributed Programs and Its Applications in Development of Distributed Systems," *Proceedings of the IEEE-CS 17th Annual COMPSAC*, pp.231-240, U.S.A., 1993.

[8] J.Ferrante, K.J.Ottenstein, J.D.Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transaction on Programming Language and System*, Vol.9, No.3, pp.319-349, 1987.

[9] K. B. Gallagher and J. R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Transaction on Software Engineering*, Vol.17, No.8, pp.751-761, 1991.

[10] S. Horwitz, T. Reps and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transaction on Programming Language and System*, Vol.12, No.1, pp.26-60, 1990.

[11] M. Kamkar, N. Shahmehri, P. Fritzson, "Bug Localization by Algorithmic Debugging and Program Slicing," *Proceedings of International Workshop on Programming Language Implementation and Logic Programming, Lecture Notes in Computer Science*, Vol.456, pp.60-74, Springer-Verlag, 1990.

[12] B. Korel, "Program Dependence Graph in Static Program Testing," *Information Processing Letters*, Vol.24, pp.103-108, 1987.

[13] A. Krishnaswamy, "Program Slicing: An Application of Object-oriented Program Dependency Graphs," Technical Report TR94-108, Department of Computer Science, Clemson University, 1994.

[14] D. Kuck, R.Kuhn, B. Leasure, D. Padua, and M. Wolfe, "Dependence Graphs and Compiler and Optimizations," *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pp.207-208, 1981.

[15] L. D. Larsen and M. J. Harrold, "Slicing Object-Oriented Software," *Proceeding of the 18th International Conference on Software Engineering*, German, March, 1996.

[16] B. A. Malloy and J. D. McGregor, A. Krishnaswamy, and M. Medikonda, "An Extensible Program Representation for Object-Oriented Software," *ACM Sigplan Notices*, Vol.29, No.12, pp.38-47, 1994.

[17] K. J. Ottenstein and L. M. Ottenstein, "The Program Dependence Graph in a software Development Environment," *ACM Software Engineering Notes*, Vol.9, No.3, pp.177-184, 1984.

[18] A. Podgurski and L. A. Clarke, "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance," *IEEE Transaction on Software Engineering*, Vol.16, No.9, pp.965-979, 1990.

[19] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay, "Speeding Up slicing," *Proceeding of Second ACM Conference on Foundations of Software Engineering*, pp.11-20, December 1994.

[20] V. Sarkar, "A Concurrent Execution Semantics for Parallel Program Graphs and Program Dependence Graphs," *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, Vol.16, No.9, pp.965-979, 1990.

[21] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, Vol.3, No.3, pp.121-189, September, 1995.

[22] M. Weiser, "Program Slicing," *IEEE Transaction on Software Engineering*, Vol.10, No.4, pp.352-357, 1984.

[23] J. Zhao, J. Cheng, and K. Ushijima, "Static Slicing of Concurrent Object-Oriented Programs," *Proceedings of the 20th IEEE Annual International Computer Software and Applications Conference*, pp.312-320, August 1996, IEEE Computer Society Press.

[24] J. Zhao, "Using Dependence Analysis to Support Software Architecture Understanding," in M. Li (Ed.), "New Technologies on Computer Software," pp.135-142, International Academic Publishers, September 1997.

[25] J. Zhao, J. Cheng, and K. Ushijima, "A Dependence-Based Representation for Concurrent Object-Oriented Software Maintenance," *Proceedings of the 2nd Euromicro Working Conference on Software Maintenance and Reengineering*, pp.60-66, Florence, Italy, March 1998.