Data Dependence for Object-Oriented Programs

Carlos Galindo
cargaji@vrain.upv.es
VRAIN
Universitat Politècnica de València
Valencia, Spain

Sergio Pérez
serperu@dsic.upv.es
VRAIN
Universitat Politècnica de València
Valencia, Spain

Josep Silva
jsilva@dsic.upv.es
VRAIN
Universitat Politècnica de València
Valencia, Spain

Abstract

The *System Dependence Graph* (SDG) is a program representation used in several static analyses. In particular, it is the basis of program slicing, a technique that extracts the part of the program that may directly or indirectly affect the values computed at a given program point (known as the slicing criterion). Several approaches have enhanced the SDG representation to deal with object-oriented situations like inheritance, polymorphism, or dynamic bindings. Currently, the most advanced approach is the *Java System Dependence Graph* (JSysDG), which subsumes previous approaches and that is able to represent all those situations. In this paper, we show that even the JSysDG does not produce complete slices in all cases when some object variables are selected as the slicing criterion. To solve this limitation, we first identify the source of the problem: the representation of dependences between partial definitions of objects is insufficient in the JSysDG, leading to a loss of completeness in many cases. Then, we extend the JSysDG with the addition of a specific flow dependence for object type variables called *object-flow dependence*. This extension provides a more accurate flow representation between object variables and its data members and allows us to obtain complete slices when an object variable is considered as slicing criterion.

Keywords: Program slicing, JSysDG, flow dependence, object-flow dependence

1 Introduction

Program representation is an important field in the program analysis research area, and it is at the base of most program analysis and transformation techniques. An accurate representation of the internal program dependences is strongly associated with the quality, precision, and performance of many program analysis techniques. In this paper, we introduce a program representation that enhances the so-called *System Dependence Graph* (SDG) [7] with a specific treatment for object-oriented programs.

The SDG is in the core of *program slicing* [15, 16], a technique for program analysis and transformation whose main objective is to extract from a program the set of statements, the *program slice* [18], that affect the value of a variable v at a program point p ($\langle p, v\rangle$), which is known as the *slicing criterion* [14]. Program slicing is applied in many disciplines such as software maintenance [5], debugging [3], and program specialization [12], among others. The SDG allows us to compute slices in linear time as a graph reachability problem. Example 1.1 shows a slice of a simple Java program.

Example 1.1. Consider the code snippet in Figure 1, which contains a program with two classes: class `A` and class `Main` with a `main` method that instantiates a type `A` object.

The code inside the boxes is the slice with respect to ⟨14, `i`⟩ (variable `i` in line 14, represented with bold underlined code). Method `f` of class `A` internally uses its data member `y` to compute its result, but not the data member `x`. Hence, as the slicing criterion is the result of a call to method `f`, the slice contains the definition of method `f` and tracks down where the value of `y` comes from. In this example, the value of `y` comes from the call to the `A`'s constructor in line 13. Although object `a` is included in the slice, we can exclude data member `x` from it, since its value does not affect the slicing criterion ⟨`14`, `i`⟩.

1.1 A little bit of history

The first program representation used to define program dependences for program slicing was the *Program Dependence Graph* (PDG), defined by Ferrante et al. in [4]. The PDG defines each program method as an individual graph, where nodes represent statements, and edges connect two statements that are related by control or flow dependences. This model, was augmented by Horwitz et al. in [7], creating the *System Dependence Graph* (SDG), that incorporates

```
1  class A{
2    public int x, y;
3    public A (int a, int b) {
4      x = a;
5      y = b;
6    }
7    public void setX(int a) { x = a; }
8    public int f(int a) { return a * y; }
9  }
10
11 class Main{
12   public static void main(String[] args){
13     A a = new A(1, 2);
14     int i = a.f(10);
15     a.setX(3);
16     A a2 = a;
17   }
18 }
```

Figure 1. Java code and its slice w.r.t. variable `i` in line 14.

an interprocedural representation of programs by linking method calls with method definitions. The resulting graph, the SDG, connects the PDGs of all methods.

The SDG has been enhanced with many approaches to represent object-oriented (OO) programs (see the survey [11]) by adding an accurate representation for their main features: polymorphism, dynamic binding, and inheritance. Larsen and Harrold extended the SDG generating a new graph, the *Class Dependence Graph* (ClDG), with a representation for all the aforementioned features in the C++ programming language [9]. Unfortunately, the representation was not accurate enough and presented some limitations. For example, it was not possible to differentiate data members of different objects in method calls. Liang and Harrold [10] improved this representation allowing them to distinguish data members in parameter objects and upgrading the accuracy of graph-based operations as a result. As some features differ between OO languages, the ClDG was not able to represent some features of the Java programming language. Hence, other approaches focused on Java were proposed. The approach proposed by Kovács et al. in [8] or the one proposed by Zhao in [19] enabled the representation of Java particular features such as interfaces, packages, and single inheritance.

Some years later, Walkinshaw et al. [17] proposed the JSysDG, a Java-based graph that encapsulates the benefits offered by the two mentioned Java approaches. The JSysDG is the composition of different graphs for methods, classes, interfaces, and packages, obtaining a more accurate program representation. Moreover, the JSysDG allows the representation of abstract classes which are not necessarily interfaces, and it distinguishes data members in parameter objects. To the best of our knowledge, the JSysDG is the latest and most accurate representation for Java programs.

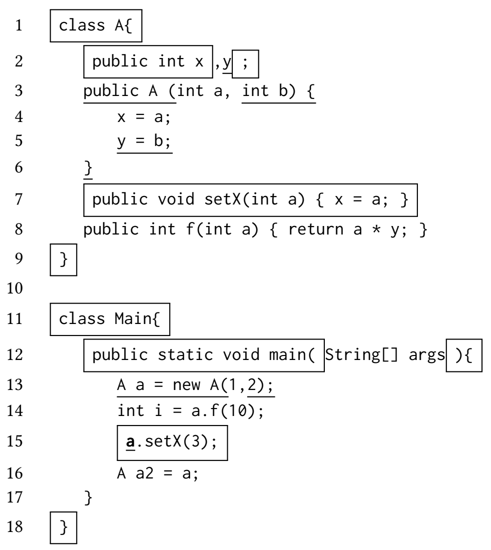
1.2 The problem

Although the JSysDG provides accurate slices and allows us to differentiate whether data members of different objects are required or not in a slice, we show in this paper that some scenarios exist where the produced slices are not complete. Therefore, it is not a problem related to precision (the slices contain more code than they need), but a problem related to completeness (the slices contain less code than they need), which means that some code that can affect the slicing criterion is not included in the slice. In particular, when an object variable is selected as the slicing criterion, under certain circumstances, only some of its required data members (not all of them) are included as part of the slice. The cause of this lack of completeness is the current definition of flow dependence. The classic flow dependence was designed for variables that are atomically defined or used in a single statement, but has never been reconsidered to deal with object variables, which can be partially defined or used (by defining or using one of its data members) in a statement.

Example 1.2 (JSysDG counterexample)**.** In Figure 2, if we define variable `a` in line 15 as the slicing criterion $\langle 15, \texttt{a}\rangle$, then we are interested in the whole object `a` after executing statement 15. This means that the slice should include all the data members of `a`, and the code needed to define them. The code inside boxes in Figure 2 is the result obtained by slicing the JSysDG. The JSysDG includes the definition of `a.x` in the slice because function call `a.setX(3)` is also included as part of the slice, but it ignores all the data members not being defined there. As a result, data member `a.y` in line 2 and its definition in line 5 are not included in the slice. The result is an incomplete slice for $\langle 15, a\rangle$, which provides no value for data member `a.y`. The complete slice is the union of the code inside the boxes and the underlined code.

This paper presents an approach to solve the problem described in Example 1.2 by extending the JSysDG. We augment the JSysDG by replacing the current definition of flow dependence with two more accurate definitions: a definition of flow dependence for primitive type variables, and another special definition for object variables that we call object-flow dependence. To allow for this specialization, the definition and use sets of each statement are also reconsidered when data members are used or defined in method calls.

The rest of the paper is structured as follows. Section 2 recalls some key concepts about the construction of the JSysDG. Section 3 redefines definition and use sets when object variables or its data members are involved. Section 4 formally introduces the object-flow dependence and justifies its necessity in OO programs. Section 5 presents some restrictions



```
1   class A{
2       public int x ,y ;
3       public A (int a, int b) {
4           x = a;
5           y = b;
6       }
7       public void setX(int a) { x = a; }
8       public int f(int a) { return a * y; }
9   }
10
11  class Main{
12      public static void main( String[] args ){
13          A a = new A(1,2);
14          int i = a.f(10);
15          a.setX(3);
16          A a2 = a;
17      }
18  }
```

Figure 2. JSysDG and expected slices of the code in Figure 1 w.r.t. ⟨15, a⟩.

that need to be added to the slicing algorithm due to object-flow dependences. Section 6 presents the related work and, Section 7 concludes.

2 Background: The Java System Dependence Graph (JSysDG)

In this section, we explain the JSysDG enhancements introduced to the original SDG in order to correctly represent the OO features of inheritance, dynamic binding, and polymorphism. We explain it through its incremental evolution: CFG $\rightarrow$ PDG $\rightarrow$ SDG $\rightarrow$ ClDG $\rightarrow$ JSysDG.

CFG. The starting graph to build a JSysDG is the *Control Flow Graph* (CFG) [1]. It is a graph that represents all possible execution paths of a method. In the CFG, each statement is represented with a node, and two nodes are connected if they may be executed sequentially . Additionally, two nodes, *Enter* and *Exit*, are added as the initial and final nodes of the method execution respectively.

PDG. From the CFG we can calculate two different dependences that are used to construct a Program Dependence Graph (PDG) [4]. These dependences are the control dependence and the flow dependence, defined hereunder.

Definition 2.1 (Control dependence)**.** Let G be a CFG. Let n and m be nodes in G. A node m post-dominates a node n in G if every directed path from n to the *Exit* node passes through m. Node m is *control dependent* on node n if and only if m post-dominates one but not all of n's CFG successors.

Definition 2.2 (Flow Dependence)**.** A node m is *flow dependent* on a previous CFG node n if:

(i) n defines a variable v,
(ii) m uses v, and
(iii) there exists a control-flow path from n to m where v is not defined.

The PDG of a procedure is a graph $G = (N, A)$ where N is the set of nodes of the CFG minus the *Exit* node, and A is a set of arcs that represent control and flow dependences.

SDG. A program usually contains a set of procedures connected by procedure calls. For this reason, in order to connect the graphs of all the procedures of a program (PDGs) and simulate parameter passing between calls and definitions, Horwitz et al. defined the *System Dependence Graph* (SDG) [7]. A SDG represents each parameter of a procedure with a formal-in node, and a formal-out node represents a parameter that may be modified inside the procedure. Analogously, each procedure call is augmented with an actual-in node for each argument of the call, and an actual-out node for each argument that may be modified by the procedure. The SDG connects procedure calls with their definitions representing parameter passing with parameter arcs. Additionally, a call arc is generated to connect the call node to the procedure *Enter* node.

Finally, a new kind of arc called summary arc is added to the SDG to describe the relation between defined and used arguments in method calls. A summary edge connects an actual-in node and an actual-out node if the value related to the actual-in node is needed to calculate the value defined in the actual-out node.

ClDG. With an SDG as its base, the *Class Dependence Graph* (ClDG) [9] augments its representation to consider OO programs. The ClDG defines a *class entry node* for each class, connected to the procedure *Enter* nodes of all its procedures by *class membership edges*, and to all its data members by *data membership edges*. In the ClDG graph, inheritance is represented with a *class dependence edge* from the base class to the derived classes.

JSysDG. The ClDG is augmented by the *Java System Dependence Graph* (JSysDG) with a process to represent polymorphic calls and dynamic binding. In this enhancement, there are two specific scenarios that are worth mentioning:

1. **A polymorphic object is the caller of a method, and the called method needs to be selected at runtime**. In this scenario, the JSysDG represents the caller and its defined and used data members as a tree. There is a node for each possible dynamic type connected to the corresponding method definition.
2. **A method call contains a polymorphic object as a parameter**. In this scenario the JSysDG representation follows the proposal introduced by Liang and Harrold in [10], where the object parameter is represented in the graph as a tree structure, with a subtree for each possible dynamic class, unfolding all its data members in both method call and definition.

Apart from the ClDG, the JSysDG also uses two other graphs for interfaces and packages. The first graph is the *Interface Dependence Graph* (InDG), which represents each interface and its defined abstract methods. Each abstract method contains in turn a set of parameter nodes representing its input parameters. Then, every abstract method and its parameters are connected to every instance of the method and the interface nodes are connected to all class nodes that implements the interface. The second graph is the *Package Dependence Graph* (PaDG). In this graph, a new node is defined for each package of the program. This node is connected to every class and interface of the package. This graph represents the program as a set of connected packages.

3 Definitions and uses of object variables

In this section we provide a more accurate description for definition and use sets of object variables. This is necessary in order to extend the notion of flow dependence (see Definition 2.2) for objects.

In Java, program variables can be of two different types. On the one hand, we have primitive type variables, which are atomic (e.g., int i = 42). These variables are always defined and used atomically, i.e., every time a primitive variable is defined in the program, the new value of the variable replaces the previous one, and the previous value cannot be further accessed. On the other hand, there are object type variables, which are compositionally formed by a collection of data members. Each data member, in turn, can be a primitive type variable, or another object type variable. Unlike primitive variables, object variables are not completely replaced every time they are defined. Since they are formed from a collection of data members, a statement may modify just some of them. As a result, the definition of all the data members of an object variable may be split into different statements. Hence, object variables can be defined in two different ways:

Definition 3.1 (Total definition)**.** An object variable v that points to a memory location $m1$ is *totally defined* in a program statement s if the execution of s makes v to point to $m2$, and $m1 \neq m2$ (v points to a different object).

Definition 3.2 (Partial definition)**.** An object variable v is *partially defined* in a program statement s if v points to the same object o before and after the execution of s, and s defines at least one data member of o.

The flow dependence definition is strictly related to the so-called definition and use sets (DEF and USE) of each statement in the program. To obtain accurate DEF and USE sets we need to reconsider the situations where object variables and its data members are defined and used considering the different kinds of possible definitions. After splitting object definitions into total and partial definitions, the possible definition situations can be classified into three different scenarios:

1. **Total definition by assignment of object variables and constructor calls (e.g., `a = new A(42)`)**. Constructor methods always define[1] all the data members of the variable and never use them. Hence, the DEF set includes the object variable itself and all its data members, and the USE set is the empty set.
2. **Total definition by alias assignment between two object variables that point to different objects (e.g., `a = b`, being `a` and `b` objects)**. Although alias assignments are not composed of a set of explicit data member definitions, they are also total definitions, so the DEF set contains a definition for the object variable itself and for all its data members. In contrast to total definition (i), their USE set contains the object variable of the right-hand side of the assignment.
3. **Partial definition. A method call that defines a data member of the caller (e.g., `a.setX(42)`)**. Method calls may partially define object variables. They can either define or use some or all the data members of the object itself. In order to differentiate the value of an object variable before and after a method call, we need to identify whether the caller object is defined or not during the method call. We consider that an object variable is being defined inside a method call (it is in the DEF set) if any of its data members may be defined during its execution. This allows us to establish the corresponding flow relationship between the previous object value and the later object value. In this case, the DEF set includes all the data members defined inside the method together with the object variable itself. For the DEF set, the order of the definitions is relevant. All data members are defined before the object variable itself. On the other hand, the USE set includes all the data members used inside the method together with the caller object itself, whose reference is always used in method calls.

Table 1 provides an example of DEF and USE sets for the `main` method of the program in Figure 1. In this table there are examples of all the described scenarios.

Scn.	Statement	DEF	USE
1	`A a = new A(1,2);`	`a.x, a.y, a`	$\emptyset$
3	`int i = a.f(10);`	`i`	`a, a.y`
3	`a.setX(3);`	`a.x, a`	`a`
2	`A a2 = a;`	`a2.x, a2.y, a2`	`a`

Table 1. DEF and USE sets for the `main` method of Figure 1.

This new method to annotate definitions and uses of object variables in program statements is of great importance to the

[1]This definition can be explicit or implicit, because Java initializes all data members by default.

redefinition of flow dependence. It allows us to differentiate the specific moment in which a data member and an object is defined and allows us to establish a relationship between previous data member definitions and later partial definitions and uses of object variables.

4 Object-flow dependence

In Java, as it happens in most programming languages, a variable cannot be used without being previously defined. The rationale why Definition 2.2 does not work with object type variables is because they can be *partially* defined, while primitive variables are always totally defined. This is also the problem of the JSysDG: it uses the standard definition of flow dependence (Definition 2.2) with object type variables. The solution is to extend the definition of flow dependence to account for objects.

We redefine this definition for object variables (it does not apply to primitive type variables, that continue using the classic definition (Definition 2.2)). We call this new dependence *object-flow dependence*. Formally,

Definition 4.1 (Object-Flow Dependence)**.** Let G be a CFG. Let n and m be nodes in G. m is *object-flow dependent* on the preceding node n if:

#1 (i) n defines an object o,
(ii) m uses the object o, and
(iii) there exists a control-flow path from n to m where object o is not defined.

or

#2 (i) n defines a data member x of an object o,
(ii) m defines o, and
(iii) there exists a control-flow path from n to m where data member x of o is not defined.

The first situation corresponds to the classic definition of flow dependence, the use-definition dependency, which also applies to object variables, even if the definition is partial. The second one considers a definition-definition dependency, produced by partial definitions. This flow dependence considers the case when the slicing criterion is an object variable, and it depends on all the complementary partial definitions that, together, produce the complete value of that variable.

When an object variable is used as a caller, we may be interested in the value before or the value after the call execution. To disambiguate this selection, we can select different nodes depending on the value we are interested in. The slicing criterion for the value before the call is the call node itself, while the one for the object after the call is the root node of the object tree representation.

Example 4.2. Consider again the code in Figure 1. We have augmented the JSysDG in Figure 3 with the object-flow dependences labeled with #1 and #2. Additionally we have augmented the information in the graph to easily identify which nodes define and use object variables and their data members with DEF and USE label sets inside some nodes. These object-flow dependences are the ones generated by the first and the second sets of conditions in Definition 4.1, respectively. The figure shows how object-flow dependences are also defined over the tree structure of method calls. Although object-flow dependences add arcs between object variables, the original definition of flow dependence is still applied to primitive type variables. This happens in the call `a.f(10)`, where data member `y` of the constructor call is linked to the argument-in node `y_in`. After the addition of object-flow dependences, the obtained slice with ⟨15, `a`⟩ as the slicing criterion in Figure 1 would be the expected slice, i.e., the code inside the boxes together with the underlined fragments of code in Figure 2.

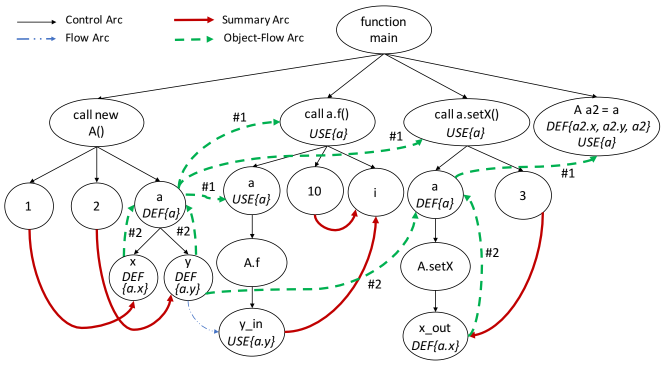


Figure 3. Object-flow dependences for the code in Figure 1.

5 The slicing algorithm

When Horwitz et al. ([7]) proposed the SDG, it was accompanied by a new slicing algorithm. This algorithm is the current method used by the program slicing community due to its accuracy and linear-time complexity. If we use Horwitz et al.'s algorithm with our graph, we would achieve the same precision. However, we would face these two problems:

a) We would include in the slice all the data members of an object variable even if we are only interested in one of them. For instance, in Figure 3, if we consider the node y inside the method call *new* $A(1,2)$, the algorithm would unnecessarily include data member x and its value 1 in the slice.
b) We would include in the slice the value of a caller object variable before a method call when we are only interested in the value after performing the call. For example, in Figure 3, if we consider the value of the object variable a after the call $a.setX(3)$, Horwitz et al.'s algorithm would include the value of the object variable a before the call in the slice.

These problems can be solved by limiting the traversal of the object-flow edges in certain cases. When the traversal reaches a node n, an incoming object-flow edge can only be traversed if it fulfils one of the following three conditions:

1. n is the slicing criterion
2. n has been reached via an object-flow edge
3. n is a predicate

Condition 2 enables the transitive traversal of object-flow edges and condition 3 prevents situations a and b, which arise of the sub-structures associated to method calls (caller object and arguments) and object variables (tree representation). The restriction imposed to the traversal of object-flow edges increases the precision of the algorithm in the presence of object variables, while keeping its linear-time complexity.

6 Related Work

Although many papers present approaches to represent features related to OO programs, there are few of them addressing the problem of data dependences explicitly. This is the case of Chen and Xu [2], who augmented the PDG of each method with tags, used to annotate data dependence arcs with the program variables involved in the data dependency. Hammer and Snelting [6] provided a specific definition about data dependence in statements due to field dependences. Their definition considered fields in an object variable through an alias variable if the name and the type of the field was the same. These two different perspectives are interesting, but their purpose was different to ours. Unlike their approaches, ours focuses on relating all the data members definitions to any object variable that appear in the program. The work by Orso et al. [13] exhaustively analyses data dependency in the presence of pointers. Their work differentiates 24 kinds of data dependency and enables slicing with respect to only some of them. Unfortunately, their data dependency is not related to the OO paradigm.

Our approach may seem similar to object slicing [10]but there are some differences between their approaches and ours. In object slicing, the slicing criterion is defined with a tuple $\langle v, p \rangle$ and an object variable O, where v is a variable in a program statement p and O is an object of the program. Its objective is essentially different: it determines which statements of O's class affect the slicing criterion through object O, while we are directly interested in considering an object variable O as the slicing criterion.

7 Conclusions

We have presented a counterexample showing that the JSysDG can be incomplete. Further, we have identified the sources of imprecision and incompleteness, and we have explained the rationale of these problems, which happen when an object variable is selected as the slicing criterion. In order to solve the problem, we have extended the definition of flow dependence with a specific treatment for object variables (object-flow dependence). This extension allows us to express the relationship between object variables and their data members in a more accurate way. Moreover, we have specified how DEF and USE variable sets must be for each statement that involves objects, especially for caller variables in method calls. Finally, the new changes in the JSysDG are accompanied by a new slicing algorithm that solves the described problems of that graph.

Acknowledgments

This work has been partially supported by the EU (FEDER) and the Spanish MCI/AEI under grants TIN2016-76843-C4-1-R and PID2019-104735RB-C41, by the *Generalitat Valenciana* under grant Prometeo/2019/098 (DeepTrust), and by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215. Sergio Pérez was partially supported by Universitat Politècnica de València under FPI grant PAID-01-18.

References

[1] Frances E. Allen. 1970. Control Flow Analysis. *SIGPLAN Not.* 5, 7 (1970), 1–19.

[2] Zhenqiang Chen and Baowen Xu. 2001. Slicing Concurrent Java Programs. *SIGPLAN Not.* 36, 4 (April 2001), 41–47. https://doi.org/10.1145/375431.375420

[3] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. 1996. Critical Slicing for Software Fault Localization. *SIGSOFT Softw. Eng. Notes* 21, 3 (May 1996), 121–134. https://doi.org/10.1145/226295.226310

[4] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (1987), 319–349.

[5] Ákos Hajnal and István Forgács. 2012. A Demand-Driven Approach to Slicing Legacy COBOL Systems. *Journal of Software Maintenance* 24, 1 (2012), 67–82. http://dblp.uni-trier.de/db/journals/smr/smr24.html#HajnalF12

[6] Christian Hammer and Gregor Snelting. 2004. An improved slicer for Java. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 17–22.

[7] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions Programming Languages and Systems* 12, 1 (1990), 26–60.

[8] Gyula Kovács, Ferenc Magyar, and Tibor Gyimóthy. 1996. *Static Slicing of JAVA Programs*. Technical Report 96-108. RGAI, Hungarian Academy of Sciences, Joesf Attila University, Hungary.

[9] Loren Larsen and Mary Jean Harrold. 1996. Slicing Object-Oriented Software. In *Proceedings of the 18th international conference on Software engineering* (Berlin, Germany) *(ICSE '96)*. IEEE Computer Society, Washington, DC, USA, 495–505. http://dl.acm.org/citation.cfm?id=227726.227837

[10] D. Liang and M. J. Harrold. 1998. Slicing Objects Using System Dependence Graphs. In *Proceedings of the International Conference on Software Maintenance (ICSM '98)*. IEEE Computer Society, Washington, DC, USA, 358–. http://dl.acm.org/citation.cfm?id=850947.853342

[11] Durga Prasad Mohapatra, Rajib Mall, and Rajeev Kumar. 2006. An Overview of Slicing Techniques for Object-Oriented Programs. *Informatica* 30, 2 (2006), 253–277.

[12] Claudio Ochoa, Josep Silva, and Germán Vidal. 2005. Lightweight Program Specialization via Dynamic Slicing. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming* (Tallinn, Estonia) *(WCFLP '05)*. ACM, New York, NY, USA, 1–7. https://doi.org/10.1145/1085099.1085101

[13] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. 2001. Effects of pointers on data dependences. In *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*. IEEE, 39–49.

[14] Karl J. Ottenstein and Linda M. Ottenstein. 1984. The Program Dependence Graph in a Software Development Environment. *SIGSOFT Software Engineering Notes* 9, 3 (1984), 177–184. https://doi.org/10.1145/390010.808263

[15] Josep Silva. 2012. A Vocabulary of Program Slicing-Based Techniques. *Comput. Surveys* 44, 3 (June 2012).

[16] Frank Tip. 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages* 3, 3 (1995), 121–189.

[17] N. Walkinshaw, M. Roper, and M. Wood. 2003. The Java system dependence graph. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*. 55–64.

[18] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th international conference on Software engineering (ICSE '81)* (San Diego, California, United States). IEEE Press, Piscataway, NJ, USA, 439–449.

[19] Jianjun Zhao. 1998. Applying Program Dependence Analysis To Java Software. In *Proceedings of Workshop on Software Engineering and Database Systems*. 162–169.