Journal of Logical and Algebraic Methods in Programming 130 (2023) 100826

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Program slicing of Java programs [☆]

Carlos Galindo, Sergio Pérez, Josep Silva [*]

Valencian Research Institute for Artificial Intelligence (VRAIN), Universitat Politècnica de València, Camino de Vera s/n, E-46022 Valencia, Spain

ARTICLE INFO

Article history:
Received 12 January 2022
Received in revised form 23 September 2022
Accepted 29 September 2022
Available online 3 October 2022

Keywords:
Program slicing
JSysDG
Flow dependence
Object-flow dependence

ABSTRACT

Program slicing is a technique to extract the part of the program that can affect the values computed at a given program point (known as the slicing criterion). To represent programs, program slicing uses the *System Dependence Graph* (SDG), for which several extensions like the Java System Dependence Graph (JSysDG) or the Sub-Statement Linear Dependence Graph (SSLDG) exist to deal with Java object-oriented programs. In this paper, we present an incompleteness result proving that these graphs do not produce complete slices in all cases, and specifically when some object variables are selected as the slicing criterion. We first identify the source of the problem: the representation of dependences between partial definitions of objects is ill-defined in these approaches, leading to a loss of completeness in many cases. To solve this limitation, we extend these representations 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 it allows us to obtain complete slices when an object variable is selected as the slicing criterion.

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 for the analysis of object-oriented programs. Specifically, we propose a program slicing technique for object-oriented programs based on our new program representation.

Program slicing [1,2] is a technique to extract from a program the set of statements, the *program slice* [3], that affects the value of a variable v at a given program point p ($\langle p, v\rangle$), which is known as the *slicing criterion* [4]. Program slicing is applied in many disciplines such as software maintenance [5], debugging [6], and program specialization [7], among others.

Example 1.1. Consider the Java code snippet in Fig. 1a, which contains a program with two classes: `A` and `Main`. The latter contains a `main` method that creates an instance of `A`.

☆ This work has been partially supported by grant PID2019-104735RB-C41 funded by Spanish MCI/AEI/10.13039/501100011033, 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. Carlos Galindo was partially supported by the Spanish *Ministerio de Universidades* under grant FPU20/03861. Sergio Pérez was partially supported by *Universitat Politècnica de València* under FPI grant PAID-01-18.

* Corresponding author.
E-mail addresses: cargaji@vrain.upv.es (C. Galindo), serperu@dsic.upv.es (S. Pérez), jsilva@dsic.upv.es (J. Silva).

https://doi.org/10.1016/j.jlamp.2022.100826
2352-2208/© 2022 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

```java
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 a1 = new A(1,2);
14     int i = a1.f(10);
15     a1.setX(3);
16     A a2 = a1;
17   }
18 }
```

(a) Java code

```java
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 a1 = new A(1,2);
14     int i = a1.f(10);
15     a1.setX(3);
16     A a2 = a1;
17   }
18 }
```

(b) Slice w.r.t. ⟨14,i⟩

Fig. 1. Java code fragment and slice with respect to variable i at line 14.

If we observe that variable i at line 14 produces a wrong value, we can use program slicing to compute the part of the program that may have influenced this value, and thus may contain the bug. Fig. 1b shows the slice (the grey code is excluded from the slice) with respect to $\langle 14, i \rangle$, represented with underlined code. Method f of class A internally uses its data member y to compute its result, but not 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 a1 is included in the slice, we can exclude its data member x and its initialization, since its value does not affect the value of the slicing criterion ⟨14,i⟩.

1.1. The problem

Currently, one of the most advanced program representations used to slice object-oriented programs is the Java System Dependence Graph (JSysDG) [8], a graph that can represent packages, interfaces, classes, and methods; and that gives support to polymorphism, dynamic binding, and inheritance. The slice shown in Fig. 1b is the minimal slice of the code in Fig. 1a, and it has been computed with the JSysDG.

Although, in general, the JSysDG provides accurate slices, in this paper, we present an incompleteness result: we show that some scenarios exist where the JSysDG's slices are not complete. Therefore, we do not reveal an imprecision problem (the slices contain more code than they need), but an incompleteness problem (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. This is a fundamental problem because completeness is a property required by most applications of program slicing. For instance, the slice computed from a bug symptom could not contain the bug. Or the slice computed from a variable to produce a specialized program could produce a specialized code that is not equivalent to the original program.

The source of the problem is that JSysDG is not prepared to select an object variable as the slicing criterion, and it can produce a slice where only some of its required data members are included. This lack of completeness is caused by the definition of flow dependence in the JSysDG. The classic flow dependence [1] 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 (e.g., by defining or using only one of its data members) in a statement.

Example 1.2 *(JSysDG completeness counterexample).* Consider again the code in Fig. 1a. We can define the slicing criterion ⟨15, a1⟩, which means that we are interested in the whole object a1 after[1] executing the method call in line 15. This means that the slice should include all the data members of a1, and the code needed to define them.

The code in black in Fig. 2a is the result obtained by slicing the JSysDG. The JSysDG includes the definition of a1.x in the slice because the function call a1.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 a1.y in line 2 and its definition in line 5 are not included in the slice. Even the call to A's constructor is missing. The result is an incomplete slice for $\langle 15, a1 \rangle$, which provides no value for data member a1.y, nor for a1 itself. The expected complete slice is the code in black of Fig. 2b.

The problem described for the JSysDG is inherited by its later extensions, such as the *Sub-Statement Level Dependence Graph* (SSLDG) [9]. The SSLDG extends the theoretical model proposed by the JSysDG to accept more concrete slicing criteria.

[1] In the rest of the paper, for simplicity in our examples, we interpret the slicing criterion as "a1 after executing line 15". The complementary interpretation "a1 before executing line 15" can be achieved by simply changing the line of the slicing criterion (i.e., ⟨14, a1⟩).

```
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 a1 = new A(1,2);
14      int i = a1.f(10);
15      a1.setX(3);
16      A a2 = a1;
17    }
18  }
```

(a) JSysDG slice

```
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 a1 = new A(1,2);
14      int i = a1.f(10);
15      a1.setX(3);
16      A a2 = a1;
17    }
18  }
```

(b) Expected slice

Fig. 2. JSysDG and expected slices of the code in Fig. 1 w.r.t. $\langle 15, \texttt{a1}\rangle$.

Both models share the same representation for object variables in method calls, which is the key factor of the incompleteness problem.

This paper presents an approach that solves the problem described in Example 1.2. We augment the JSysDG by replacing the current definition of flow dependence with three more accurate definitions: the standard definition of flow dependence for primitive variables and another pair of new definitions for object variables that we call *object-flow dependence* and *object-reference dependence*. These definitions require the specialization of the 'defined variables (*DEF*)' and 'used variables (*USE*)' sets of each statement, two concepts coming from the JSysDG that are explained later through Definitions 2.1 and 2.2. The specialization of these sets allows the JSysDG to distinguish when object variables are being totally or partially 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 explains and justifies the incompleteness problem of the current JSysDG when slicing an object that is the caller of a method call, illustrating it with an example in Java. Then, in Section 4, the *DEF* and *USE* sets for every statement are redefined in order to properly represent definitions and uses of object variables and its data members. After that, Section 5 formally introduces object-flow dependence and object-reference dependence, and justifies their necessity in OO programs. Section 6 presents some slicing restrictions that need to be added to the slicing algorithm when traversing object-flow dependences; and Section 7 presents the empirical evaluation comparing the JSysDG and the JSysDG extended with object dependences. Finally, Section 8 presents the related work and Section 9 concludes.

2. Background: the Java System Dependence Graph (JSysDG)

This section has been included to keep the paper self-contained. Those readers already familiar with the JSysDG can skip this section, where we explain how the program representation field has evolved to reach the current solution to properly represent the program in Fig. 1a so that we can automatically obtain the slice shown in Fig. 1b. In particular, we explain the JSysDG through its incremental evolution:

$$\text{CFG} \rightarrow \text{PDG} \rightarrow \text{SDG} \rightarrow \text{ClDG} \rightarrow \text{JSysDG}$$

CFG. The starting graph to build a JSysDG is the *Control Flow Graph* (CFG) [10]. 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. Two nodes, *Enter* and *Exit*, are added as the initial and final nodes of the method execution respectively. Additionally, every CFG node is augmented with two sets: the definition set and the use set, that denote the set of variables respectively defined and used at this CFG node. Formally,

Definition 2.1 *(Definition set (DEF)).* Let G be a CFG. Let n be a node in G representing a statement s. The definition set of n is denoted with $DEF(n)$ and it contains all the program variables that are defined (their value is assigned) at statement s.

Definition 2.2 *(Use set (USE)).* Let G be a CFG. Let n be a node in G representing a statement s. The use set of n is denoted with $USE(n)$ and it contains all the program variables that are used (their value is accessed) at statement s.

PDG. From the CFG we can calculate two different dependences that are used to construct a Program Dependence Graph (PDG) [11]. These dependences are the *control dependence* and the *flow dependence*, defined hereunder.

```
1   class A{
2     public int x, y;
3     public A (int a, int b) { x = a; y = b; }
4     public int getX() { return x; }
5     public int getY() { return y; }
6     public void f() { x = x + 1; }
7   }
8   class B extends A{
9     public B (int a, int b){ super(a,b); }
10    public void f() { x = x + 2; }
11  }
12  class Main{
13    public static void main(String[] args){
14      A a;
15      if (Math.random() > 0.5)
16        a = new A(1);
17      else
18        a = new B(2);
19      a.f();
20      g(a);
21    }
22    public void g(A a){
23      System.out.println(a.getX() + a.getY());
24    }
25  }
```

Fig. 3. Fragment of Java code with a polymorphic call.

Definition 2.3 *(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 2.4 *(Flow dependence).* Let G be a CFG. Let n and m be nodes in G. Node m is *flow dependent* on node n if:

(i) $v \in DEF(n)$,
(ii) $v \in USE(m)$, and
(iii) there exists a control-flow path from n to m where v is not redefined.

The PDG of a method 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 methods connected by method calls. For this reason, in order to connect the PDGs of all the methods of a program and simulate parameter passing between calls and definitions, Horwitz et al. defined the *System Dependence Graph* (SDG) [12]. An SDG represents each parameter of a method with a formal-in node, and a formal-out node represents a parameter that may be modified inside the method. Analogously, each method 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 method. The SDG connects method calls with their definitions representing parameter passing with (interprocedural) parameter arcs (*input* arcs connect actual-in with formal-in nodes and *output* arcs connect formal-out with actual-out nodes). Additionally, a *call* arc is generated to connect the call node to the method *Enter* node. Finally, a new kind of arc called *summary* arc is added to the SDG to describe the relation between input and output arguments in method calls. A summary arc 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) [13] augments its representation to consider OO programs. The ClDG defines a *class entry* node for each class, connected to the method *Enter* nodes of all its methods by *class membership* arcs, and to all its data members by *data membership* arcs. In the ClDG graph, inheritance is represented with a *class dependence* arc from the base class to the derived classes.

JSysDG. The *Java System Dependence Graph* (JSysDG) augments the ClDG with a representation for polymorphic calls and dynamic binding. This can be seen in Figs. 3, 4, and 5, which shows a Java program and two portions of its JSysDG that illustrate different aspects of the graph described above. This figure also shows how the JSysDG represents two specific scenarios that are worth mentioning to later understand the source of incompleteness:

1. **A polymorphic object is the caller of a method, and the call's target is only known at runtime**. 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. An example of this situation can be seen in Example 2.5.

   Example 2.5. Consider the code in Fig. 3, where a Java program with two classes with an inheritance relationship (`B` extends `A`) is represented. The program also contains a `Main` class with a `main` method that creates an object of either `A` or `B` dynamic type depending on a randomly generated number. In line 19, the method that would be executed in the call `a.f()` can only be determined at runtime, thus, the static representation of the program needs to define both possibilities. An example of how the JSysDG represents this method call is shown in Fig. 4.

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 [14], where the object parameter is represented in the graph as a tree

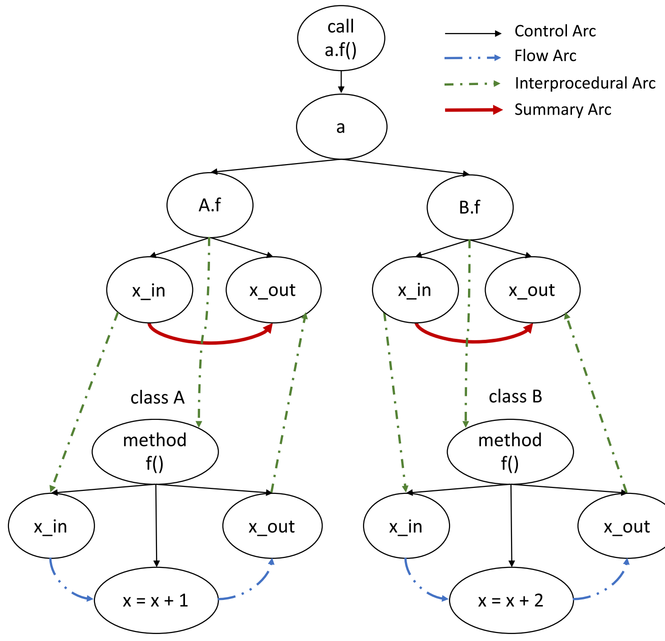


Fig. 4. JSysDG of call `a.f()` in line 19 of Fig. 3.

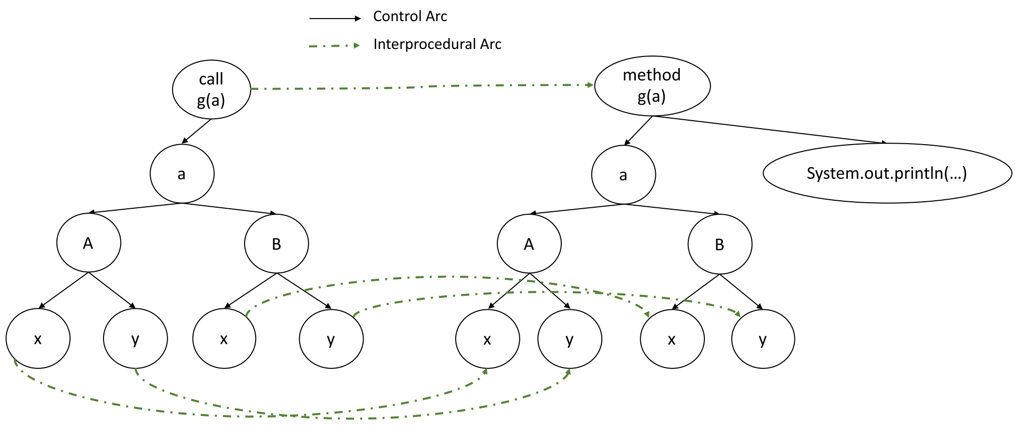


Fig. 5. JSysDG of call `g(a)` in line 20 of the code in Fig. 3.

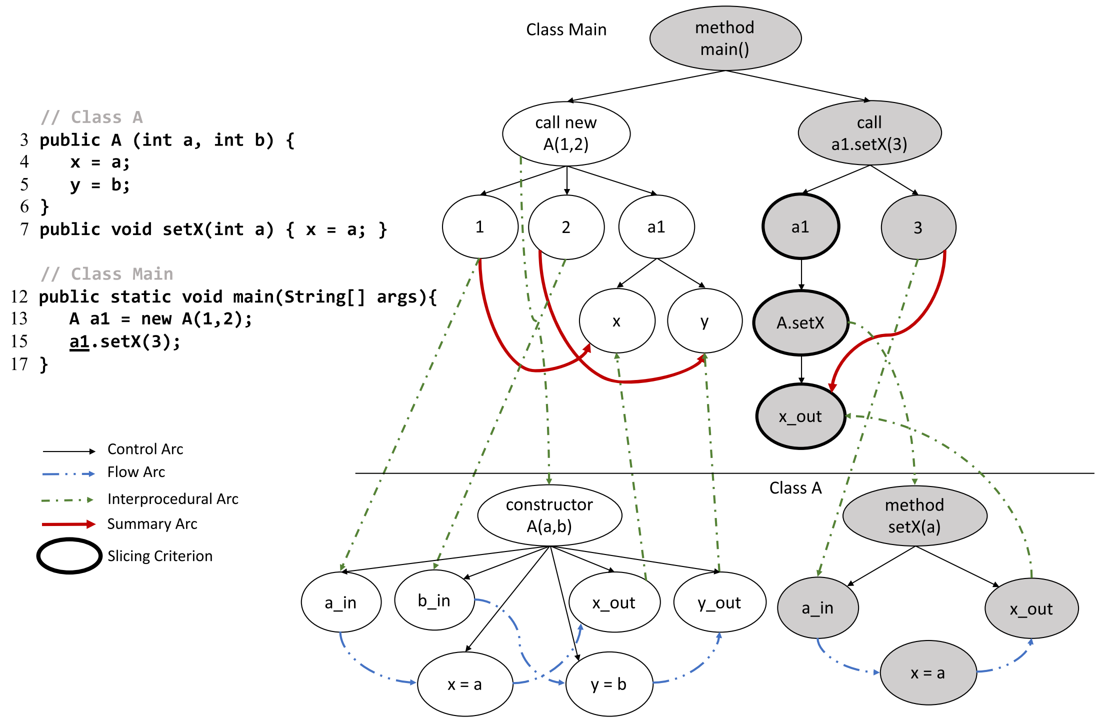
structure, with a subtree for each possible dynamic class, unfolding all its data members in both method call and definition. This scenario is shown in Example 2.6.

Example 2.6. Consider the call to method `g` in line 20 of Fig. 3. This call receives a polymorphic object (variable `a`) as a parameter. The corresponding JSysDG representation is shown in Fig. 5. In the JSysDG, only data members inside each type tree are linked, in order to accurately select only those data member used inside the method definition of `g`. Note that, when we follow the proposed representation, if an object is represented recursively, the tree representation may be infinite. To address this issue, the JSysDG employs a *k-limiting* approach (a tree representation is only unfolded to a level `k`).

3. Limitations of the JSysDG

As it has been proven with Example 2, even though being the current most accurate representation of Java OO programs, the JSysDG can produce incomplete slices. In this section we explain what its cause is.

In Java, program variables can be of two different types. On the one hand, we have primitive 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.



```
   // Class A
3  public A (int a, int b) {
4     x = a;
5     y = b;
6  }
7  public void setX(int a) { x = a; }

   // Class Main
12 public static void main(String[] args){
13    A a1 = new A(1,2);
15    a1.setX(3);
17 }
```

Fig. 6. JSysDG representing lines 13 and 15 of the `main` method in Fig. 1, and slice w.r.t. ⟨15, `a1`⟩ (grey nodes represent the slice, bold nodes the slicing criterion).

On the other hand, there are object variables, which are compositionally formed by a collection of data members. Each data member, in turn, can be a primitive variable, or another object variable.

The JSysDG was proposed by Walkinshaw et al. [8] extending the representation provided by Liang and Harrold for C++ [14]. Both approaches share the same definition for program slice and slicing criterion:

"A program slice is a set of program statements and predicates that might affect the value of a program variable v that is defined or used at a program point p; ⟨p, v⟩ is known as the slicing criterion."

According to this definition, an object variable can be considered as the slicing criterion. If we select an object variable as the slicing criterion, the slice should include all the statements that might affect the value of any of its data members. Nevertheless, as we have seen in Fig. 4, when an object variable is the caller of a method call, only the used and defined data members of the object have a representation in the unfolding tree as argument-in and argument-out nodes respectively. Additionally, all the flow dependences between object definitions and uses in method calls are propagated through their data members, but they never connect the node that represents the object variable itself.

If we put all this knowledge together, we can identify a specific scenario where the slice obtained by the JSysDG is not complete when selecting some object variables as the slicing criterion. This scenario is given in the `main` method of Fig. 1, in particular in lines 13 and 15 (the relevant code is shown in Fig. 6). Line 13 creates object variable `a1` of type `A` instantiating its data members `x` and `y` with a call to `A`'s constructor. Then, line 15 modifies only the value of its data member `x`. Therefore, after line 15, the values of the data members (`x` and `y`) of object `a1` come from two statements: `x` is defined in line 15 (with value 3) and `y` is defined in line 13 (with value 2). However, there is not flow dependence between these two statements because method call `a1.setX(3)` does not depend on any data member of object variable `a1` to define data member `x`. The described situation is represented in Fig. 6.

Fig. 6 represents with grey nodes the slice produced by the JSysDG with respect to ⟨15, `a1`⟩. This slicing criterion is marked in the JSysDG with a bold line. Note that, when the slicing criterion is an object variable, then all the data members are marked as the slicing criterion as well. Thus, the slicing criterion is formed from the control dependence subtree of node `a1`. This slice corresponds to the code shown in Fig. 2a. Contrarily to the expected slice, which is the one in Fig. 2b, the slicing traversal of the JSysDG ends without reaching the declaration of `a1`, the constructor call `new A(1, 2)`, and data member `y` in class `A`. This counterexample reveals that flow dependences of object variables need to be redefined so that slices are correctly computed when a whole object variable is selected as the slicing criterion.

Table 1
The ordered sequence DEF_{os} and the *USE* set of some of the statements in the `main` method of Fig. 1a.

Scenario	Statement	DEF_{os}	*USE*
1	`A a1 = new A(1,2);`	`[a1.x, a1.y, a1]`	$\emptyset$
3	`a1.setX(3);`	`[a1.x, a1]`	{`a1`}
2	`A a2 = a1;`	`[a2.x, a2.y, a2]`	{`a1`}

4. Definitions and uses of object variables

In order to solve the problem identified in the previous section, we need to extend the standard notion of flow dependence (see Definition 2.4) for the case of objects. We start by giving a more accurate description for the *DEF* and *USE* sets when object variables are contained inside them.

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 only 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 4.1 *(Total definition of an object variable).* 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).

Total definitions appear in two different scenarios:

1. **An assignment of an object variable with a constructor call.** This happens, e.g., in `A a1 = new A(1,2)` (see line 13 of Fig. 1), where `a1` is totally defined. Constructor methods always define[2] all the data members of the variable and never use them. Hence, the *DEF* set for this statement ($DEF(\#13)$) includes the object variable itself and all its data members. The order in which all these definitions occur is crucial for the definition of flow dependence. For this reason, we transform the *DEF* set into an ordered sequence called DEF_{os} where all data members are defined first, and the object variable is defined at the end, e.g., the ordered sequence of definitions in the mentioned statement is $DEF_{os}(\#13) = [a1.x, a1.y, a1]$. In this scenario, $USE(\#13) = \emptyset$ since no variable is used.
2. **An alias assignment between two object variables that point to different objects.** This happens, e.g., in `A a2 = a1` (see line 16 of Fig. 1), where `a2` is totally defined.[3] As in the previous scenario, the *DEF* set is transformed into an ordered sequence where data members are defined first, and the object variable is defined at the end. In contrast, $USE(\#16)$ remains as a set of uses, and contains the object variable of the right-hand side of the assignment ({a1}).

Definition 4.2 *(Partial definition of an object variable).* 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.

Partial definitions occur when a statement modifies at least one data member of an object variable, but not the object it points to. Partial definitions appear in two different scenarios:

3. **A method call that defines a data member of the caller.** This happens, e.g., in `a1.setX(3)` (see line 15 of Fig. 1). Method calls may partially define object variables by defining some or all of its the data members. The *DEF* set is also transformed to an ordered sequence which, as it happened in total definitions, includes all the data members defined inside the method followed by 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 needed to make the call.
4. **An assignment of an object variable's data member.** This happens, e.g., in `a1.x = 42`. The JSysDG treats this case as a particular case of the previous scenario because the JSysDG inherits the program assumptions given by Liang and Harrold when building their version of the ClDG: *"We further assume that data members of an object can be accessed only through a method (we replace a direct reference to a data member of an object as a method call)"*. For this reason, before building the JSysDG, `a1.x = 42` is `a1.setX(42)`. Therefore, the DEF set turns into an ordered sequence that contains data member `a1.x` followed by the object variable itself. In this case, the *USE* set contains the object variable defined (`a1`).

Example 4.3. Table 1 provides an example of DEF and USE sets for the `main` method of the program in Fig. 1. In this table there are examples of the described scenarios. Definitions and uses appear in the order described in the previous scenarios.

[2] This definition can be explicit or implicit, because Java initializes all data members by default.
[3] This type of total definitions include also assignments of the form `A a = f();` where an object variable is defined through the return of a method call.

```
   // Class A
public void setX(int a) { x = a; }

   // Class Main
public static void main(String[] args){
    a1.setX(3);
}
```

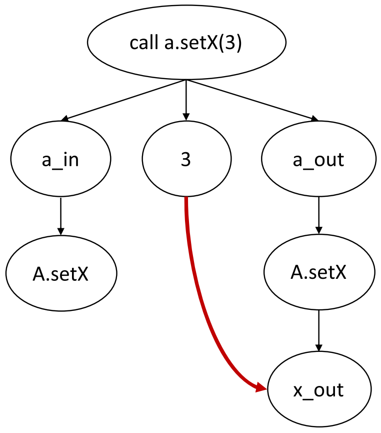


Fig. 7. Input and output scopes in a method call.

```
1        (...)
2        int x = 5;    // def x
3        x = 1;        // def x
4        y = x;        // use x
```

(a) Def-use with primitive variables.

```
1        (...)
2        A a = new A();    // def a
3        a.setX(1);        // def a
4        a.setY(2);        // def a
5        b = a;            // use a
```

(b) Def-use with object variables.

Fig. 8. Flow dependence in presence of primitive and object variables.

This new method to annotate definitions and uses of object variables in program statements is of fundamental importance for the redefinition of flow dependence. Moreover, it comes with an important advantage over the previous formulation: It allows us to differentiate the specific moment in which a data member and an object is defined or used and, thus, it allows us to slice a program with respect to a specific instant in the computation (for instance, we can slice `a.setX(3)` with respect to the value of `a` 'before' or 'after' the method invocation). To incorporate this feature to the JSysDG, we have enhanced the graph representation for object variables at the scope of a method call. On the one hand, we provide an object tree for the scope to represent its value before the call (*scope_in*). The root of this tree represents the use of the object variable, and the tree contains the data members used inside the function call. On the other hand, if the object is modified during the method invocation, we provide another object tree that represents its value after the call (*scope_out*). The root of this second tree represents the definition of the object variable and it contains the data members defined inside the function call. For instance, considering the function call `a.setX(3)`, the resulting JSysDG is shown in Fig. 7.

5. Definition of object-flow and object-reference dependences

In Java, as it happens in most programming languages, a variable cannot be used without being previously defined. The definition of a primitive variable is always total, but, as shown in Section 4, the definition of an object variable can be partial. This poses new difficulties in the computation of the usual flow dependence. In fact, the standard definition (see Definition 2.4) is insufficient. Let us illustrate the problem with a simple example.

Example 5.1. Consider the code in Fig. 8a, where the primitive variable `x` is defined twice and then used. According to Definition 2.4, `x` in line 4 depends on `x` in line 3, but `x` in line 4 does not depend on `x` in line 2.

If we consider, however, the code in Fig. 8b, we have an analogous situation: `a` is defined twice and then used. Therefore, according to the standard definition of flow dependence, `a` in line 5 flow depends on `a` in line 4 (which is correct), but `a` in line 5 does not depend on `a` in line 3 (which is clearly wrong).

The rationale why Definition 2.4 does not work with object 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 2.4) with object variables. The solution is to extend the definition of flow dependence to account for objects. Interestingly, this extension allows for novel situations. For instance, in Fig. 8b, `a` in line 5 depends on `a` in line 3 (even though it is redefined in line 4), but also, a (partial) definition can depend on another definition. In particular, `a` in line 4, which is a (partial) definition, depends on `a` in line 3, which is another definition.

With all these ideas, we identify (and formally define) two new dependences that complement the classical flow dependence with a specific treatment for object variables (it does not apply to primitive variables, which continue using the classic

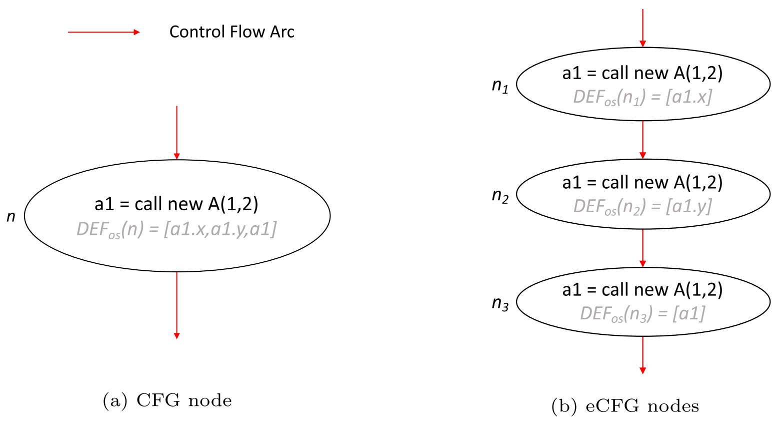


Fig. 9. CFG and eCFG nodes corresponding to statement `A a1 = new A(1,2)`.

flow dependence definition (Definition 2.4)). We call these new dependences *object-flow dependence* and *object-reference dependence*. The former can be formally defined with an extended CFG.

Definition 5.2 *(Extended control-flow graph).* Given a CFG $G = (N, A)$, its extended version eCFG is a graph G' with the same nodes and arcs as G with one exception: every node $n \in N$ that defines an object variable and contains $m > 1$ variable definitions ($DEF_{os}(n) = [v_1, v_2 \dots v_m]$) is replaced by a sequence of nodes ($n_1, ..., n_m$, connected by arcs) where each variable definition is represented by one single node.

The eCFG is useful to explicitly represent the order in which a sequence of operations happen inside a single statement. These operations cannot be distinguished in a CFG because all of them are represented with one single node, while they can be distinguished in the eCFG because they are represented by a sequence of nodes.

Example 5.3. Consider the assignment `A a1 = new A(1,2)` in line 13 of Fig. 1. This assignment contains a total definition of variable `a1` and its CFG node together with its sequence of definitions and uses are shown in Fig. 9a. Note that the sequence of definitions appears in the order described by scenario 1 in Section 4. Since the node contains three different definitions, the node must be split into different nodes in the corresponding eCFG. Fig. 9b shows how the initial CFG node is split into three different nodes (one per variable definition) which are sequentially connected to form the corresponding eCFG representation.

We can now formally define object-flow dependence.

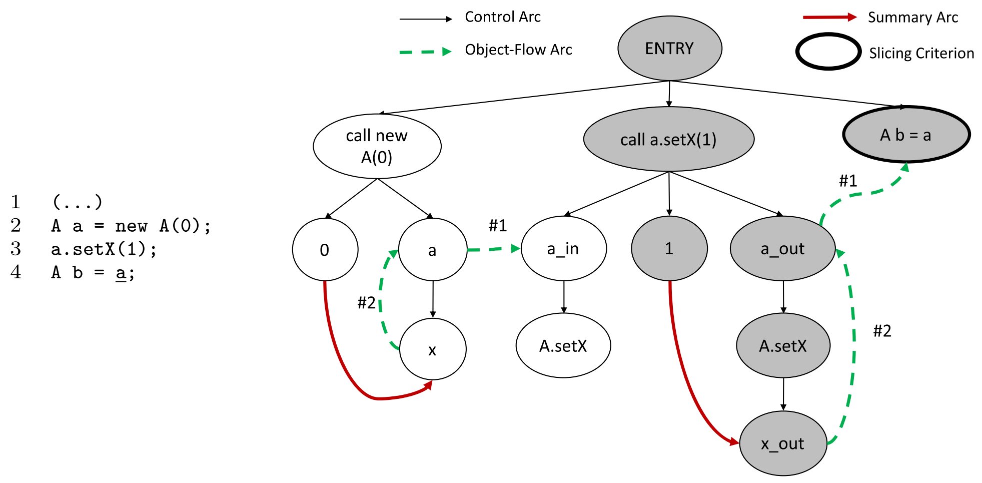
Definition 5.4 *(Object-flow dependence).* Let n and m be nodes in an eCFG. m is *object-flow dependent* on n if:

#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 redefined.
or
#2 (i) n defines a data member x of an object o,
(ii) m defines object o,[4] and
(iii) there exists a control-flow path from n to m where data member x of o is not redefined.

The first set of conditions corresponds to the classic definition of flow dependence, the definition-use dependence, which also applies to object variables, even if the definition is partial. The second one considers a definition-definition dependence, 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.

Object-flow dependence represents all situations in which the values of the data members of an object are propagated. In particular, it is able to identify multiple partial definitions of an object and properly connect all of them to produce the whole value of an object. But object-flow dependence does not consider the *object reference* of objects. Unlike primitive variables, object variables have a pointer to a memory position where a specific object is stored. This pointer is updated

[4] I.e., in the eCFG, $DEF_{os}(m) = [v]$, where v is an object variable that points to object o.



```
1   (...)
2   A a = new A(0);
3   a.setX(1);
4   A b = a;
```

Fig. 10. Code with data member redefinition (left) and its JSysDG with object-flow (right).

every time a total definition of an object variable is executed (see Definition 4.1). Example 5.5 shows a scenario where object-flow dependence is insufficient to include the reference of an object in a slice.

Example 5.5. Consider the code in Fig. 10, where an object a of class A (which has one single data member x) is totally defined in line 2. Then, line 3 redefines data member x (even though its unique data member is redefined, the object reference remains unchanged). Finally, line 4 contains a use of variable a. All object-flow dependences in this code are represented in the graph of the figure labeled with #1 and #2. These object-flow dependences are the ones generated by the first (#1) and second (#2) sets of conditions in Definition 5.4, respectively.

With these dependences, the slice computed for $\langle 4, a\rangle$ is shown with grey nodes. Clearly, this slice is incomplete, because line 2 should be included in the slice. The problem is that there is a missing dependence between *a_out* in line 3 and a in line 2. The missing part is the object reference of a, which is defined in line 2. The rationale is the following: even though all data members of *a_out* are defined in line 3, the object reference of *a_out* is not. Therefore, *a_out* in line 3 must depend on the code that defines its object reference (line 2).

To account for these missing dependences, we define a new type of dependence called *object-reference dependence* that complements object-flow dependence. It connects objects with their object reference. Formally,

Definition 5.6 *(Object-reference dependence).* Let G be a CFG. Let m and n be nodes in G. n is *object-reference dependent* on m if m totally defines an object o, n partially defines object o, and there exists a control-flow path from m to n where object o is not totally defined.

Object-flow and object-reference dependences are the key elements to solve the incompleteness problem of the JSysDG. The completeness of both dependences with respect to any definition-use combination of statements with object variables is an important result that we formulate here and prove in Appendix A and Appendix B.

Theorem 5.7 *(Object-flow completeness). Let $s_1; s_2; \dots s_n;$ be a sequence of statements. Let x be an object variable or a data member of an object variable defined at s_1. If the value of an object o at s_n data depends on the execution of s_1, then there is a transitive object-flow dependence between s_1 and s_n.*

Theorem 5.8 *(Object-reference completeness). Let $s_1; s_2; \dots s_n;$ be a sequence of statements. Let x be an object variable totally defined at s_1. If the reference of an object variable v at s_n depends on the total definition of s_1, then there is a path formed by object-flow and/or object-reference arcs between s_1 and s_n.*

The final JSysDG that includes our new dependences is shown in the following example. With object-reference dependence it solves the problem explained in Example 5.5.

Example 5.9. Consider again the code in Fig. 1a. Fig. 11 shows its associated JSysDG. The upper part of the figure corresponds to the representation of class `Main`, where we can see how object dependences are also defined over the tree structure of method calls. Although object dependences add arcs between object variables, the original definition of flow dependence is still applied to primitive variables. This happens in the call `a1.f(10)`, where data member `y` of the constructor call is

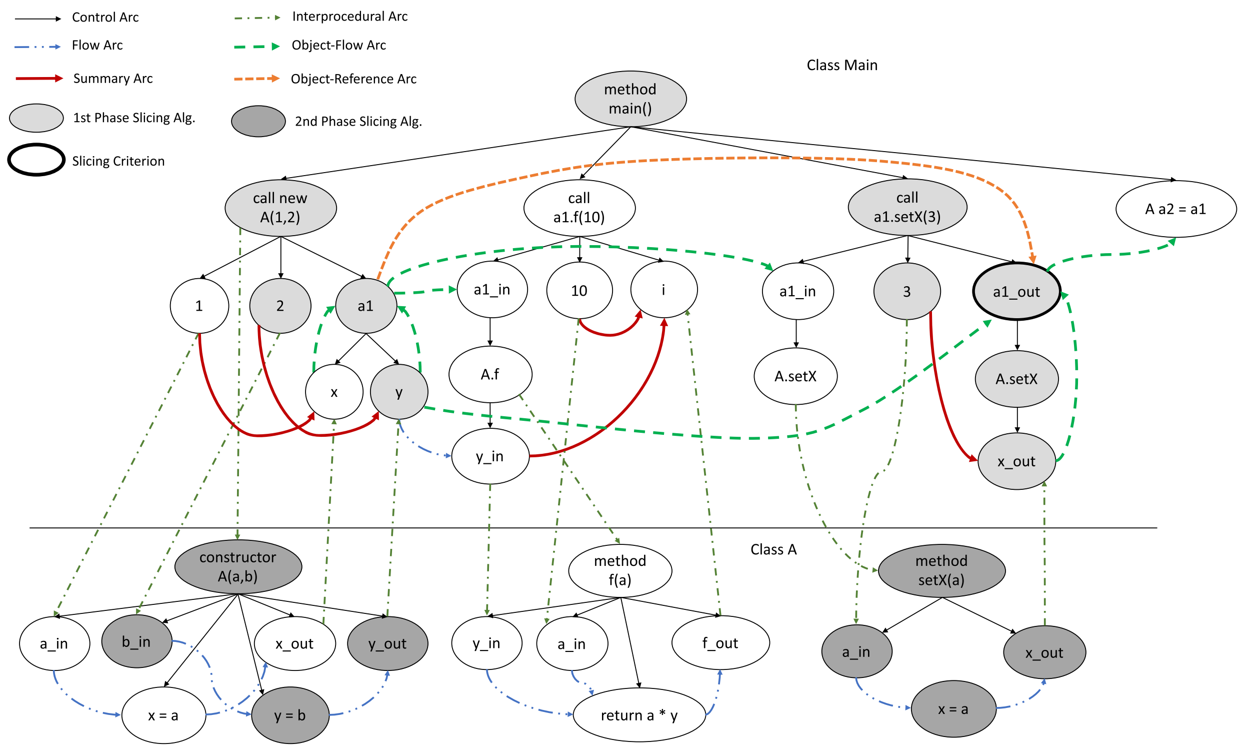


Fig. 11. JSysDG of the program in Fig. 1 and slice w.r.t. $\langle 15, \texttt{a1}\rangle$.

linked to the argument-in node `y_in`. We can see that the object-reference arc is needed to connect the object variable `a1` after the call `a1.setX(3)` with the construction of this object in `call new A(1,2)`. This JSysDG extended with object dependences can properly slice the graph from any slicing criterion. For instance, the slice computed for the slicing criterion $\langle 15, \texttt{a1}\rangle$ in Fig. 1a would be the expected slice, i.e., the code in black in Fig. 2b.

6. The slicing algorithm

The JSysDG slicing algorithm is the standard one proposed by Horwitz et al. [12]. It computes slices in two phases: (i) traverse the graph backwards from the slicing criterion collecting all nodes reached using any arc except for *output* arcs, (ii) traverse the graph backwards from any node in the slice collecting all nodes reached using any arc except *call* and *input* arcs. The overall process has a linear time complexity. This algorithm can be used with our graph, producing the same precision as with the JSysDG. However, this algorithm does not take advantage of the new object-flow dependences, producing a loss of precision. The standard algorithm 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 Fig. 11, if we consider the node y inside the method call $new\ A(1,2)$, the algorithm would unnecessarily include in the slice data member x and its value 1 (due to arc $a1 \overset{\#2}{\longleftarrow} x$).

The problem can be solved by limiting the traversal of the object-flow arcs in certain cases. When the traversal reaches a node n, an incoming object-flow arc can only be traversed if one of the following three conditions is true:

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

Condition 1 ensures that the traversal of object-flow arcs is still transitive. Conditions 2 and 3 provide a starting point to traverse object-flow arcs. When the slicing criterion is an object variable (Condition 2), we need to traverse all object-flow arcs to include in the slice the value of all its data members. On the other hand, in Condition 3, when the traversal reaches a predicate (e.g., the condition of an `if` or a `while` statement) we need to follow object-flow arcs to include the data members used by its condition in order to keep the slice complete. The restriction imposed to the traversal of object-flow arcs increases the precision of the algorithm in the presence of object variables, while keeping its linear time complexity. Algorithm 1 includes all these conditions in the original slicing algorithm proposed in [12], making it suitable to traverse the JSysDG after adding object-flow dependences.

Algorithm 1 Slicing algorithm for the JSysDG with object-flow dependence.

```
Input: A JSysDG $G$ and the slicing criterion node $n_{sc}$.
Output: The set of nodes that compose the slice $S$ of $G$ w.r.t. $n_{sc}$.
Initialization: $S_0 \leftarrow \{\langle n_{sc}, none\rangle\}$.

1: function MarkNodesOfSlice($G, n_{sc}$)
2:   $S_1 \leftarrow$ MarkReachingNodes($G, S_0, n_{sc}, \{Output\}$)                ▷ Phase 1
3:   $S_2 \leftarrow$ MarkReachingNodes($G, S_1, n_{sc}, \{Call, Input\}$)          ▷ Phase 2
4:   $S \leftarrow \{n \mid \langle n, arcType\rangle \in S_2\}$
5:   return $S$

6: function MarkReachingNodes($G, N, n_{sc}, ArcTypes$)                         ▷ Change ($A$)
7:   $WorkList \leftarrow N$
8:   while $WorkList \neq \emptyset$ do
9:     select some $\langle n, lastArcType\rangle \in WorkList$                       ▷ Change ($B$)
10:    $WorkList \leftarrow WorkList \setminus \langle n, lastArcType\rangle$
11:    for all $arc \in$ getIncomingArcs($n$) do
12:      if $arcType \in ArcTypes$ then
13:        continue
14:      $m \leftarrow$ getSourceNode($arc$)
15:      if $arcType = ObjectFlow$ then
16:        if $lastArcType \neq ObjectFlow \wedge n \neq n_{sc} \wedge \neg$isPredicate($m$) then
17:          continue
18:      $N \leftarrow N \cup \langle m, arcType\rangle$
19:      $WorkList \leftarrow WorkList \cup \langle m, arcType\rangle$
20:  return $N$
```

Algorithm 1 illustrates the slicing process by including a couple of relevant changes to Horwitz's algorithm. The first one (A) is that the slicing criterion n_{sc} is now used as a parameter of function MarkReachingNodes, since this function needs to identify in Line 16 whether the current node is the slicing criterion (to check Condition 2). Additionally, (B) the `WorkList` in function MarkReachingNodes contains now tuples of two elements (see Line 9) instead of a single element, storing also the information about the type of the last traversed arc (to check Condition 1). The three aforementioned conditions are checked in lines 15-17. If any of them holds then the current node is not included in the slice.

Example 6.1 shows how the application of Algorithm 1 solves the incompleteness problem of the JSysDG presented in Section 3.

Example 6.1. The graph in Fig. 11 represents the JSysDG of the code in Fig. 2 augmented with object-flow and object-reference dependences. It shows the slice computed with respect to the object variable `a1` in line 15 after the method call `a.setX()` (the slicing criterion node is marked with a bold line in the graph). To clearly show the two traversal phases of the algorithm, the slice is divided into two sets of nodes. The nodes marked in light grey are added to the slice during Phase 1. On the other hand, the dark grey nodes are the nodes added to the slice during Phase 2. The slice code is exactly the expected slice shown in Fig. 2b. It is worth mentioning that the change proposed in Section 6 to Horwitz et al.'s algorithm has a direct impact on the slice's precision.

Note that the traversal algorithm improves its accuracy by preventing the `x` data member of object variable `a1` in method call `new A(1,2)` to be included in the slice. This happens because `x` can only be reached from `a1` (through an object-flow arc) and this arc can only be traversed from another object-flow arc, according to Condition 1 of the slicing algorithm. Even though `a1` can be reached from two object-flow arcs, none of them belong to the slice, thus `x` is never included in the slice.

7. Empirical evaluation

In this section we compare our implementation, which include object-flow and object-reference dependences, with the original JSysDG. We have compared both graph implementations by measuring the graph generation time, the slicing time, and the size of the slice, comparing the results obtained by both slicer executions.

All the algorithms and ideas described in this paper have been implemented in a prototype slicer for Java programs called *JavaSlicer*. JavaSlicer is open source and publicly available.[5]

To compare the performance difference between the JSysDG and JavaSlicer, we used the publicly available Java library *re2j*, a library developed by Google to work with regular expressions in Java. In particular, we used the most recent release of this library (version 1.6[6]). *re2j* is a project with 8100 lines of code, distributed in 19 different Java files. In order to evaluate the techniques proposed throughout this work, we used both the JSysDG and JavaSlicer to build and slice the whole *re2j* library. Then, to make the comparison meaningful, we selected as slicing criteria all the object variables (root node of the

[5] Available at https://github.com/mistupv/JavaSlicer.

[6] Available at https://github.com/google/re2j/releases/tag/re2j-1.6.

Table 2
Summary of experimental results, comparing the slices of *re2j* produced by the JSysDG (A) and JavaSlicer (B).

Size range	SCs	Slice time (A)	Slice time (B)	Slower	Size (A)	Size (B)	Improv.
[0, 100)	49	0.076 ± 0.001 ms	0.927 ± 0.018 ms	48.876	9.10	39.59	693.74%
[100, 1000)	95	130.98 ± 1.823 ms	217.315 ± 2.874 ms	0.782	608.69	738.68	23.19%
[1000, 1400)	122	622.67 ± 10.833 ms	1164.423 ± 13.093 ms	0.857	1284.66	1664.99	29.32%
[1400, 1800)	146	881.09 ± 13.101 ms	1584.023 ± 12.429 ms	0.779	1535.62	1948.16	26.77%
[1800, ∞)	31	1603.05 ± 11.769 ms	2943.965 ± 15.702 ms	0.842	1994.42	2522.74	26.73%
[0, ∞)	443	602.13 ± 9.078 ms	1095.440 ± 12.039 ms	6.126	1130.99	1439.91	100.47%

Build times: JSysDG: 10.85 ± 0.09 s, JavaSlicer: 13.35 ± 0.07 s (23% slower).

object tree representations) that are directly connected to a return statement.[7] The result of this selection is a total of 443 different slicing criteria located over the 19 Java files.

All experiments were done on an Intel Core i5-7600 processor with 16 GB RAM (DDR4 at 2400MT/s) under a Linux OS with kernel version 5.18.1. The experiment was run with the OpenJDK Java Virtual Machine (version 11.0.15+10). During the execution of the experiment, all processes and services except for the program slicer and the shell it was launched from were completely stopped to prevent interferences in the CPU performance.

The methodology used to measure the performance of both slicers was the following: each time measurement (the graph generation or a specific slice) was repeated 101 times, and the first iteration was always discarded (to avoid influence of dynamically loading libraries to physical memory, data persisting in the disk cache, etc.). Finally, we computed the average with error margins (with 99% confidence) with the remaining 100 values.

The results of the experiments performed are summarized in Table 2.[8] In it, the slicing criteria are grouped according to the size of the JSysDG slice (in nodes). This is due to the strong influence the size of the slice has on the time required to compute it and on the relative sizes of slices produced by both graphs. The columns of Table 2 are described as follows:

- **Size Range**: the sizes of the JSysDG slices (in nodes) that are contained in each row.
- **SCs**: the number of slicing criteria contained in a particular range.
- **Slice Time (A/B)**: the average time required to slice the corresponding JSysDG (A) and JavaSlicer (B).
- **Slower**: how much slower is JavaSlicer in comparison to the JSysDG. It is computed as $(Time_B - Time_A)/Time_A$.
- **Size (A/B)**: the average number of nodes in the JSysDG (A) and JavaSlicer (B) slices.
- **Improv.** (*improvement*): the average increase in size of the JavaSlicer slice compared to the JSysDG slice. It is computed as $(Size_B - Size_A)/Size_A$.

To interpret the results, we first need to focus on the usage of the SDG. Typically, a graph will be built once and sliced multiple times, and thus its creation can be (and often is) much more costly than its traversal. Regarding complexity, creation is polynomial and traversal is linear. Our results are as expected: creating the graph is between one and four orders of magnitude more time-consuming (depending on the final slice's size). Regarding the graph creation, we can observe a 23% increase between the JSysDG and JavaSlicer, which is attributable to the great number of object trees that are featured on the graph. The graph itself consists of 42000 nodes, most of which represent objects, their polymorphism or their members. Thus, computing object-flow and object-reference arcs represents a noticeable increase in time consumption.

If we turn our attention to the slicing portion of the results, we can see that the slices can be classified into two distinct groups. The first group consists of slices whose size is below 100 in the JSysDG tend to "blow up", as the addition of object-flow and object-reference adds a significant number of nodes to the slice (relative to the size of the original slice). Thus, the relative columns of the first row of results show a slicer that includes almost 700% more nodes and takes about 50 times longer. Due to the small size of the resulting slices and the short time it takes the JSysDG to compute them (0.08 ms), a 50x increase only manages to bring the average up to 0.9 ms, which is negligible in most applications of program slicing. This first group can be considered an outlier, and it affects the averages of the table as a whole. The second group is represented by the rest of the table (slices with sizes above 100 nodes), where the results show that slices increase between 23% and 30%, with the cost being a 80% time increase. Throughout the table, time has increased linearly with slice size, as can be seen in Fig. 12. Whether the cost is worth the improvement in completeness is up to the application of slicing. For example, in compiler optimization, completeness is an important requirement for slicers; while for dependency highlighting or light debugging, a faster but incomplete slice may be more appropriate. Another important remark is the comparison between the slice results computed over the two models. We verified that every JavaSlicer's slice included all the statements of the associated JSysDG slice. The rationale behind this is that the graph representation used by JavaSlicer is a conservative extension of the JSysDG (i.e., it also contains all the nodes and arcs of the JSysDG), where object-flow and

[7] Note that, to compare the performance of both models, selecting an object variable during the slicing traversal is preferable because any slicing criterion that produces a slice in the JSysDG without object variables would produce the same slice in our augmented JSysDG, as the two representations are identical except for object-flow and object-reference arcs.

[8] The full dataset produced by the slicer execution is publicly available at https://mist.dsic.upv.es/git/program-slicing/SDG/-/snippets/9.

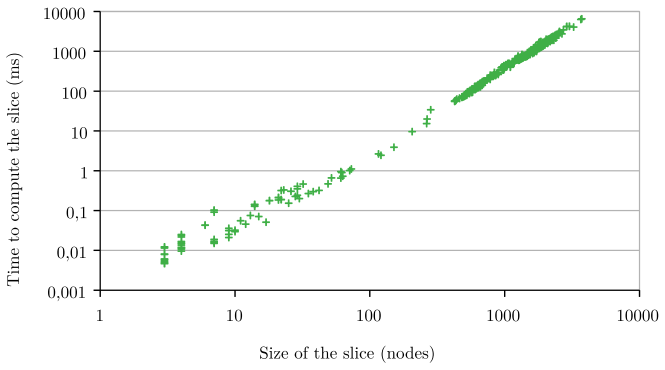


Fig. 12. Relationship between the size and time required to compute a slice (logarithmic scale).

object-reference arcs are added. Therefore, for any possible slicing criterion, the slice computed with the JSysDG (S_{JSys}) will always be contained in the slice computed with JavaSlicer (S_{JS}):

$$S_{JSys} \subseteq S_{JS}$$

8. Related work

In 1996, Larsen and Harrold [13] proposed the first graph representation able to slice OO programs, the Class Dependence Graph (ClDG). Their approach was the first to provide a way to represent inheritance, polymorphism, and dynamic binding. The ClDG connected all the methods and data members of a class in a single graph. Their proposal was later improved by other approaches like those of Tonella et al. [15], or Liang and Harrold [14]. Although all these proposals focused on OO programs, none of them noticed the difference between total and partial definitions in object variables and their impact in flow/data dependence.

When we review the literature looking for specific slicing approaches for Java, we find interesting related papers. For instance, Hammer and Snelting [16] defined a new object unfolding process for method calls in presence of recursive data types, improving the results of the *k-limiting* approach proposed in [14]. They defined an algorithm to completely unfold object variables without unfolding the same object twice in the same object tree and without loosing dependences based on point-to information. Kashima et al. [17] compared four different backward slicing techniques for Java: static execute before (SBE), based in the CFG; context-insensitive slicing (CIS), ignoring the call context; hybrid model (HYB), where the slice is defined as the intersection of SBE and CIS; and improved slicing (IMP), based on the Hammer and Snelting's work [16]. They compared their precision, scalability, and tradeoffs, determining IMP as the more accurate but not applicable to large programs.

Other techniques focused on the representation of polymorphic objects in Java. This is the case of the JSysDG [8] and the SSLDG [9], mentioned at the beginning of this paper. Both graphs include a multiple-layer representation that contemplates Java programs at different levels: package level, class level, method level, and statement level. At statement level, both graphs use the unfolding of object variables in method calls using a tree-like representation for data members proposed in [14]. There are three main differences between these two graphs: the first one in that the SSLDG includes a sub-statement layer where object variables outside method calls are further split into their data members, making the representation of direct accesses to object fields explicit in the graph; the second one is that the SSLDG proposal enhances the information of the slicing criterion when slicing polymorphic objects, using a triplet as slicing criterion where the new element indicates the dynamic type of the object being sliced to further reduce the computed slice; and the third difference is the modification of the slicing algorithm to adapt its graph for forward slicing.

Other works dealt with object-oriented programs in other programming languages like C++ or Python. In [18], Pani et al. present an algorithm for finding dynamic slices for object oriented programs in presence of function overloading on C++; and in [19], Jain and Poonia proposed a mixed static and dynamic slicing for C++ OO programs, where they generate dynamic slices in a faster and more accurate way by using object-oriented information in C++. These works are centered on dynamic slicing, and they are not focused in the representation of object variables, but in taking advantage of the information given by the compiler to compute the slice. On the other hand, a program slicing approach for Python is presented by Xu et al. in [20]. This work defines the Python System Dependence Graph (PySDG), used to slice Python First-Class objects. The PySDG takes all the first-class objects including functions, methods, classes and modules into consideration, to construct the dependencies between the definition and use statements of these first-class objects. In this model, the authors also introduce a new kind of dependence (*Entity Dependence*), which describes the dependency relationship between the statement which defines the entity object and the statement which calls the entity object.

There are also some other works mainly focused on modeling data dependences for slicing in object-oriented programs. Chen and Xu [21] augmented the PDG of each method with tags, used to distinguish the different definitions and dependences inside a statement. The authors defined five different sets: $Def(s)$, $Ref(s)$, $Def(s, x)$, $Dep\_D(s, x)$, and $Dep\_R(s, x)$. These sets were used to annotate data dependence arcs with the program variables involved in each data dependence. Despite the perspective is interesting, its purpose is different to ours. It gives extra information to data dependences by annotating them, limiting the graph traversal at slicing level when reaching a node looking for just a particular variable. Contrarily, our approach focuses on establishing a dependence between an object variable and the value of all its corresponding data members in any program point. The work by Orso et al. [22] exhaustively analyzes data dependence in the presence of pointers. They considered two different aspects: the classification of definitions and uses, and the classification of different kinds of paths in the CFG. In their work, they differentiate 24 kinds of data dependence and allowed the possibility of slicing with respect to only some of them including the considered dependences as part of the slicing criterion. Unfortunately, their data dependence is more suitable for point-to analysis than for the OO paradigm, as it is based on point-to relationships.

Our approach may seem similar to object slicing, introduced by Liang and Harrold [14], or class slicing, defined by Chen and Xu in [23], but there are some differences between their approaches and ours. In object slicing, the slicing criterion is defined with a tuple $\langle v, p, o \rangle$ where v is a variable in a program statement p, and o is an object variable of the program. Object slicing determines which statements of o's class affect the slicing criterion through object variable o. First, a standard slice is computed for the criterion $\langle v, p \rangle$. Then, considering the computed slice, a new process isolates all method calls with o as the caller object. Afterwards, method definitions corresponding to o's detected method calls are identified. Finally, o's slice is computed by extracting from the initial slice all the statements corresponding to those method definitions. Note that, in object slicing, the slicing criterion is not the object variable itself, but a mechanism to reduce the statements included in the original slice. Our objective is different: we are interested in considering the object variable o as the slicing criterion, extracting from the whole program (not only from o's class) the code that affects its whole value (the value of all its data members) in a particular statement. Class slicing [23] is similar to object slicing: it is also defined over a slicing criterion $\langle v, p, c \rangle$, but this time, instead of a specific object o, a class c is selected. Class slicing is restricted to a single class. It extracts any statement in c that affects the slicing criterion $\langle v, p \rangle$ for every object instance of class c that is part of the slice. This approach is a bit far from what we are interested in, because it considers a set of objects, not a single one and, once again, it focuses on a single class while we are interested in the whole program.

9. Conclusions

Object-oriented programs are challenging for program slicing since they introduce features such as inheritance, polymorphism, and dynamic binding; for which the standard program representation (the SDG) was not prepared. The scientific community has iteratively improved the SDG to solve the above problems, being the current solution the JSysDG, which subsumes previous approaches and introduces a precise representation of call sites where polymorphic objects are involved as the caller or as a parameter. Its representation allows us to accurately slice the data members of an object variable being used in a method, or defined through it.

Nevertheless, we have shown in this paper that the JSysDG is not prepared to slice programs when an object variable is selected as the slicing criterion because it can produce imprecise or even incomplete slices. Our first contribution is the design of a counterexample that reveals the JSysDG incompleteness. Further, we have identified the source of that incompleteness, and we have explained the rationale of this problem.

In order to solve the problem, we have properly redefined the *DEF* and *USE* sets for any statement that involves objects, specifically for caller variables in method calls. This is our second contribution, which has been mapped to the program representation so that it can be used for slicing. Since caller variables in method calls are unfolded in a tree representation, we have associated the definition and use of the caller to different nodes of the call. This has produced an important advantage of our program representation: in contrast to the standard approach, we allow a caller object to be selected as slicing criterion both before and after the method call was performed, obtaining a complete and accurate slice in both cases.

Our third contribution is a new definition of flow dependence with a specific treatment for object variables. This extension allows us to express the relationship between object variables and its data members in a more accurate way. Hence, flow dependence is split into flow dependence for primitive variables, and flow dependence for object variables (object-flow dependence and object-reference dependence). It is important to remark that the new definitions come with a proof of completeness, showing that they capture all dependences not considered in the previous approach.

Our fourth contribution is the extension of the JSysDG with the new object dependences and a new slicing algorithm that properly treats them. The new algorithm not only solves the described problems of the JSysDG, ensuring completeness and improving precision, but it also keeps the linear-time performance of the standard slicing algorithm.

Finally, our last (fifth) contribution is a new slicer for Java, the JavaSlicer, with the JSysDG in its basis that includes object-flow and object-reference dependences. Our experimental evaluation has shown that this incompleteness situation is frequently given in our benchmarks. Our approach adds around a quarter additional nodes to larger slices (100 or more nodes) and increases sizes seven-fold for smaller slices. The cost to this increase in completeness is a 23% slower graph creation and a linearly slower traversal to generate each slice.

Although our proposal targets Java, it can be easily adapted for similar object-oriented programming languages like C++, where the relation between objects and data members is analogous to Java's.

Appendix A and Appendix B include the proofs of Theorems 5.7 and 5.8, respectively.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Appendix A. Completeness proof of Theorem 5.7

Before proving Theorem 5.7, in order to ease the proof by reducing the number of possible scenarios, we enunciate and prove the following lemma:

Lemma 1. Let s be a statement in a program P. Let v be an object variable totally defined at s that points to object o. The value of v at s is not object-flow dependent on any previous statement s'.

Proof. According to Definition 4.1 (see also the scenarios 1 and 2 of Section 4), a statement s that totally defines an object variable also defines all its data members. In Definition 5.4, a statement s can be object-flow dependent on another statement s' iff: (i) s uses an object o, or (ii) s defines an object o and there is a data member x of o not defined between s' and s. Since s is a definition of o and also defines all o's data members, neither (i) nor (ii) are fulfilled and, thus, s cannot be object-flow dependent on any previous statement. □

Theorem 5.7 *(Object-flow completeness). Let $s_1; s_2; \ldots s_n$; be a sequence of statements. Let x be an object variable or a data member of an object variable defined at s_1. If the value of an object o at s_n data depends on the execution of s_1, then there is a transitive object-flow dependence between s_1 and s_n.*

Proof. First of all, according to Lemma 1, when s_n is a total definition the theorem is trivially proved because s_n defines the value of o and its data members and no object-flow path can end at s_n. With respect to the rest of possibilities, we divide the proof in three different scenarios:

1. **o is defined at s_1 and not redefined in $s_2 \ldots s_{n-1}$.**
   (a) If s_n uses object o and $x = o$, i.e., object variable x points to object o, the claim follows trivially by case #1 of Definition 5.4.
   (b) If s_n partially defines object o and x is a data member of o not defined at s_n the claim follows trivially by case #2 of Definition 5.4.
   (c) If s_n uses object o and x is a data member of o, then s_1 also defines o according to scenario 3 described in Section 4. Thus, the claim follows by the object-flow path formed by cases 1a and 1b.
   (d) If s_n partially defines object o and $x = o$ there exists an object-flow path from s_1 to s_n iff s_1 also defines a data member of o not defined in s_n, which makes this case equivalent to case 1b.
2. **o is not defined at s_1.**
   If s_1 defines x where $x \neq o$ or x is a data member of a different object p, object-flow dependence cannot be applied, because in both cases #1 and #2 of Definition 5.4 it is mandatory for both statements to operate over the same object. Hence, there cannot be an object-flow path between s_1 and s_n.
3. **x is redefined in $s_2 \ldots s_{n-1}$.**
   In this case, we can assume that $x = o$ or x is a data member of object o because in any other case we will find ourselves in case 2. We can prove the claim for $n = 3$ ($s_1; s_2; s_3;$) because it does not matter the number of transitive dependencies. The proof is the same for each transitive step. We can analyze all cases separately. We use the following notation: $s_i(A, B)$ with $A = D$ to denote that x is defined at s_i and with $A = U$ to denote that x is used at s_i; and with $B = O$ to denote that x is an object variable that points to object o at s_i and with $B = DM$ to denote that x is a data member of object o at s_i. Since s_1 and s_2 perform the same definition over x they must share the same notation in all cases. This fact leaves 8 possible scenarios:
   (a) $s_1(D, O); s_2(D, O); s_3(D, O)$;
   (b) $s_1(D, DM); s_2(D, DM); s_3(D, O)$;
   Scenarios 3a and 3b are trivially proved because s_3 is a total definition of object o and total definitions does not depend on any previous statement according to Lemma 1. Hence, s_3 cannot be object-flow dependent on a previous statement s_1.
   (c) $s_1(D, O); s_2(D, O); s_3(U, O)$;
   (d) $s_1(D, O); s_2(D, O); s_3(D, DM)$;

In scenario 3c, s_3 is trivially object-flow dependent on s_2 (case #1 of Definition 5.4) while in scenario 3d s_3 is object-flow dependent on s_2 if s_2 defines a data member of o different from the one defined at s_3 (case #2 of Definition 5.4). In both scenarios there is not direct object-flow dependence between s_1 and s_3 because the redefinition of x in s_2 prevents it. Additionally, there cannot be a transitive dependence either because s_2 cannot be object-flow dependent on s_1 due to Lemma 1.

(e) $s_1(D, DM); s_2(D, DM); s_3(U, O)$;

(f) $s_1(D, DM); s_2(D, DM); s_3(D, DM)$;

To prove scenarios 3e and 3f it is important to remember that when a statement s_i defines a data member of an object o, it also defines o (see scenario 3 of Section 4). In scenario 3e, s_3 is object-flow dependent on s_2 because it is the last existent definition of object o (case #1 of Definition 5.4). In scenario 3f, s_3 is object-flow dependent on s_2 if s_2 defines a data member of o different from the one defined at s_3 (case #2 of Definition 5.4). In scenarios 3e and 3f, due to the existence of s_2, s_3 cannot be directly dependent on s_1 according to Definition 5.4. In turn, in both scenarios s_2 is object-flow dependent on any previous statement that defines a different data member of o (case #2 of Definition 5.4). Since s_1 and s_2 define the same data member x s_2 is not object-flow dependent on s_1 and there is no transitive object-flow path between s_1 and s_3.

(g) $s_1(D, O); s_2(D, O); s_3(U, DM)$;

(h) $s_1(D, DM); s_2(D, DM); s_3(U, DM)$;

Considering scenarios 3g and 3h, note that the theorem considers “the value of an object o” at s_3. In these two cases, s_3 uses a data member of an object o. If the data member of o is itself an object, these scenarios would be equivalent to scenarios 3c and 3e respectively. Finally, if the data member o is a primitive, this case would be out of the scope of the proof. In this case, there would not be a path to the last definition of this data member formed by object-flow edges, but for flow edges. □

Appendix B. Completeness proof of Theorem 5.8

Before proving Theorem 5.8, in order to ease the proof by reducing the number of possible scenarios, we enunciate and prove the following lemma:

Lemma 2. Let s be a statement in a program P. Let v be an object variable totally defined at s that points to object o. The reference of v at s is not object-reference dependent on any previous statement s'.

Proof. According to Definition 5.6, a statement s is object-reference dependent on another statement s' if s partially defines an object variable v. Since s totally defines v, s also defines v's reference (see Definition 4.1) and the condition is not fulfilled. Thus, s cannot be the target of any object-reference dependence. □

Theorem 5.8 *(Object-reference completeness). Let $s_1; s_2; \ldots s_n;$ be a sequence of statements. Let x be an object variable totally defined at s_1. If the reference of an object variable v at s_n depends on the total definition of s_1, then there is a path formed by object-flow and/or object-reference arcs between s_1 and s_n.*

Proof. First of all, according to Lemmas 1 and 2, when s_n is a total definition the theorem is trivially proved because s_n cannot be the target of any object-flow or object-reference dependence. With respect to the rest of possibilities, we divide the proof into three different scenarios:

1. **x is defined at s_1 and not redefined in $s_2 \ldots s_{n-1}$.**
   (a) If s_n uses object variable v and $x = v$ the claim follows trivially by the object-flow arc generated by case #1 of Definition 5.4.
   (b) If s_n partially defines object variable v and $x = v$ the claim follows trivially by Definition 5.6.
2. **v is not defined at s_1.**
   If s_1 defines x where $x \neq v$, object-flow and object-reference dependences cannot be applied, because in both Definitions 5.4 and 5.6 it is mandatory for both statements to operate over the same object. Hence, there cannot be a path formed by object-flow and/or object-reference arcs between s_1 and s_n.
3. **x is redefined in $s_2 \ldots s_{n-1}$.**
   In this case, we can assume that $x = v$ because in any other case we will find ourselves in case 2. We can prove the claim for $n = 3$ ($s_1; s_2; s_3;$) because it does not matter the number of transitive dependencies. The proof is the same for each transitive step. We can analyze all cases separately. We use the following notation: $s_i(A)$ with $A = D_T$ to denote that x is totally defined at s_i, with $A = D_P$ to denote that x is partially defined at s_i, with $A = U$ to denote that x is used at s_i, and with $A = *$ to denote that A can be either D_T, D_P or U. There are several possible scenarios:
   (a) $s_1(*); s_2(*); s_3(D_T)$;
   Scenario 3a represents the set of cases where s_3 totally defines variable v. These scenarios are trivially proved by Lemma 2, since s_3 cannot be object-referent dependent on any previous statement.

(b) $s_1(D_P); s_2(*); s_3(*)$;
Scenario 3b illustrates the set of cases where s_1 partially defines x. Note that this scenario is not contemplated by the theorem because "x is totally defined at s_1" is a condition described in the theorem that this scenario does not contemplate.
(c) $s_1(D_T); s_2(D_T); s_3(U)$;
(d) $s_1(D_T); s_2(D_T); s_3(D_P)$;
In scenarios 3c and 3d, s_3 is trivially object-flow dependent on s_2 by cases #1 and #2 of Definition 5.4, respectively. In both scenarios there is neither direct object-flow, nor object-reference dependence between s_1 and s_3 because the total definition of x in s_2 prevents it. Additionally, there cannot be a transitive dependence because s_2 cannot be object-flow nor object-reference dependent on s_1 due to Lemmas 1 and 2.
(e) $s_1(D_T); s_2(D_P); s_3(U)$;
In scenario 3e, s_3 is trivially object-flow dependent on s_2 according to case #1 of Definition 5.4. Additionally, since s_2 partially defines an object variable that is totally defined in s_1, s_2 is object-reference dependent on s_1 according to Definition 5.6. Therefore, in this scenario, there is a transitive path formed by object-flow and object-reference arcs that connect the use of an object variable to its last total definition in s_1.
(f) $s_1(D_T); s_2(D_P); s_3(D_P)$;
Finally, in scenario 3f, s_3 may be object-flow dependent on s_2 according to case #2 of Definition 5.4 if they define different data members of the same object. Either way, both s_3 and s_2 are always object-reference dependent on s_1 for being partial definitions of an object variable totally defined at s_1. Consequently, there is at least one path from the last total definition in s_1 to the later partial definition in s_3 formed by an object-reference arc. □

References

[1] F. Tip, A survey of program slicing techniques, J. Program. Lang. 3 (1995) 121–189.
[2] J. Silva, A vocabulary of program slicing-based techniques, ACM Comput. Surv. 44 (2012).
[3] M. Weiser, Program slicing, in: Proceedings of the 5th International Conference on Software Engineering, ICSE '81, IEEE Press, Piscataway, NJ, USA, 1981, pp. 439–449.
[4] K.J. Ottenstein, L.M. Ottenstein, The program dependence graph in a software development environment, Softw. Eng. Notes 9 (1984) 177–184, https://doi.org/10.1145/390010.808263.
[5] A. Hajnal, I. Forgács, A demand-driven approach to slicing legacy COBOL systems, J. Softw. Maint. 24 (2012) 67–82, http://dblp.uni-trier.de/db/journals/smr/smr24.html#HajnalF12.
[6] R.A. DeMillo, H. Pan, E.H. Spafford, Critical slicing for software fault localization, SIGSOFT Softw. Eng. Notes 21 (1996) 121–134, https://doi.org/10.1145/226295.226310.
[7] C. Ochoa, J. Silva, G. Vidal, Lightweight program specialization via Dynamic Slicing, in: Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming, WCFLP '05, ACM, New York, NY, USA, 2005, pp. 1–7.
[8] N. Walkinshaw, M. Roper, M. Wood, The Java system dependence graph, in: Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation, 2003, pp. 55–64.
[9] W. Lulu, L. Bixin, K. Xianglong, Type slicing: an accurate object oriented slicing based on sub-statement level dependence graph, Inf. Softw. Technol. 127 (2020) 106369, https://doi.org/10.1016/j.infsof.2020.106369, https://www.sciencedirect.com/science/article/pii/S0950584920301385.
[10] F.E. Allen, Control flow analysis, SIGPLAN Not. 5 (1970) 1–19.
[11] J. Ferrante, K.J. Ottenstein, J.D. Warren, The program dependence graph and its use in optimization, ACM Trans. Program. Lang. Syst. 9 (1987) 319–349.
[12] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, ACM Trans. Program. Lang. Syst. 12 (1990) 26–60.
[13] L. Larsen, M.J. Harrold, Slicing object-oriented software, in: Proceedings of the 18th International Conference on Software Engineering, ICSE '96, IEEE Computer Society, Washington, DC, USA, 1996, pp. 495–505, http://dl.acm.org/citation.cfm?id=227726.227837.
[14] D. Liang, M.J. Harrold, Slicing objects using system dependence graphs, in: Proceedings of the International Conference on Software Maintenance, ICSM '98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 358–367, http://dl.acm.org/citation.cfm?id=850947.853342.
[15] P. Tonella, G. Antoniol, R. Fiutem, E. Merlo, Flow insensitive C++ pointers and polymorphism analysis and its application to slicing, in: Proceedings of the 19th International Conference on Software Engineering, 1997, pp. 433–443.
[16] C. Hammer, G. Snelting, An improved slicer for Java, in: Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 2004, pp. 17–22.
[17] Y. Kashima, T. Ishio, K. Inoue, Comparison of backward slicing techniques for Java, IEICE Trans. Inf. Syst. 98 (2015) 119–130.
[18] S.K. Pani, P. Arundhati, M. Mohanty, An effective methodology for slicing C++ programs, Int. J. Comp. Eng. Technol. 1 (2010) 72–82.
[19] S. Jain, M.S. Poonia, A new approach of program slicing: mixed SD (static & dynamic) slicing, Int. J. Adv. Res. Comput. Commun. Eng. 2 (2013).
[20] Z. Xu, J. Qian, L. Chen, Z. Chen, B. Xu, Static slicing for Python first-class objects, in: 2013 13th International Conference on Quality Software, 2013, pp. 117–124.
[21] Z. Chen, B. Xu, Slicing concurrent Java programs, SIGPLAN Not. 36 (2001) 41–47, https://doi.org/10.1145/375431.375420.
[22] A. Orso, S. Sinha, M.J. Harrold, Effects of pointers on data dependences, in: Proceedings 9th International Workshop on Program Comprehension, IWPC 2001, IEEE, 2001, pp. 39–49.
[23] Z. Chen, B. Xu, Slicing object-oriented Java programs, SIGPLAN Not. 36 (2001) 33–40, https://doi.org/10.1145/375431.375418.