Dynamic data flow analysis for Java programs

A.S. Boujarwah*, K. Saleh, J. Al-Dallal

Department of Electrical and Computer Engineering, Kuwait University, P.O. Box 5969, Safat 13060, Kuwait

Received 23 July 1999; received in revised form 14 February 2000; accepted 11 April 2000

Abstract

A large portion of high-level computer programs consists of data declaration. Thus, an increased focus on testing the data flow aspects of programs should be considered. In this paper, we consider testing the data flow in Java programs dynamically. Data flow analysis has been applied for testing procedural and some object-oriented programs. We have extended the dynamic data flow analysis technique to test Java programs and show how it can be applied to detect data flow anomalies. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Dynamic data flow analysis; Instrumentation; Java; Software testing

1. Introduction

Software applications are normally produced in large software development teams, where a large number of software engineers cooperate. Due to the size and complexity of such applications, they are usually susceptible to errors. Since the aim is to develop highly reliable software products, software testing is a critical and important verification activity during the application software development life cycle. Although testing is a costly and time-consuming life cycle activity, it increases the level of confidence in the developed software.

In modern high-level programs, a large part of source code consists of data declarations. Data flow analysis is a useful technique with which certain classes of data declaration and manipulation errors can be detected. Therefore, it is mainly concerned with the proper use of data through the software under test. The improper declaration and manipulation of program data lead to data anomalies. These anomalies may lead to serious software functionality problems at run-time. Data flow analysis of software can be accomplished statically by inspecting the source code and tracking the sequence of uses of the variables of the program under test without running it, or dynamically by executing the software and tracking its sequences of actions. These sequences of actions can be obtained by inserting probes into the original source code. Dynamic analysis is more powerful than static analysis in evaluating array subscripts, pointers, and parameter passing and in detecting anomalies associated with each. However, the static technique is generally more appropriate due to its broad-range detection of data flow anomalies, unlike the limited detection of the dynamic technique, which detects anomalies only along specifically executed code paths. Thus, the two approaches are considered complementary. Dynamic data flow analysis systems for programs written in FORTRAN [1], Cobol [2], Pascal [3], and C [4,5] have been developed. In such programming languages, data is used in functions and manipulated by basic operators. The dynamic data flow analysis has been extended to test C++ programs considering its object-oriented nature [6]. In object-oriented programming languages, all the data, operators and functions for an object are encapsulated within the object itself. Thus, a simple assignment can lead to variety of actions. Chen and Low used the idea originated from Price [7] of tracking the use of absolute memory locations associated with the data to track the data flows in C++ programs. This idea solves the data hiding problems caused by object operators.

Java is a general-purpose object-oriented programming language developed by Sun Microsystems [8]. Java programs have machine-independent byte-code representation. Thus, they can be run anywhere on the Internet using Java-enabled web browsers on multi-vendor stand-alone personal computers or workstations. Since Java is a relatively new programming language, much research and experimentation is already underway. With an increased emphasis on reliability, careful testing is particularly important as a pre-release measure for Java programs.
In a related work, we considered testing Java compilers [9], and method interfaces [10]. In this work, our aim is to extend the dynamic data flow analysis to test Java programs. Since Java does not allow any manipulation of references, we show how to deal with data encapsulation problems without referencing data memory locations.

The paper is organized as follows. In Section 2, we give an overview of Java. Section 3 discusses briefly the dynamic data flow analysis concept. In Section 4, we describe the extensions of dynamic data flow analysis to test Java programs. In Section 5, we apply the extensions to an example of a Java program. Finally, in Section 6, we conclude the paper.

2. Overview of Java

Java is a new object-oriented programming language close to C/C++. Using Java, developers can create dynamic and interactive programs that can run inside Web pages. Moreover, it can be used to create animations, games, and stand-alone applications. In Java, a class is a template for creating multiple objects or instances with similar features. Many instances can be created from one class and each can be instantiated to different initial state (i.e. combination of attribute values). A class consists of attributes and behavior. Attributes are individual characteristics used to differentiate one object from another and usually describe the object’s state. They are considered as global variables for the entire object, and are therefore called instance variables. The class defines the type of attributes, and each object stores its own values for these attributes. Attributes that are declared static are global for the class and all its instances and reside only in the class definition. The behavior of a class defines what its objects can do to read or change their internal state or communicate with other classes or objects of the same or other classes. To define the object’s behavior, functions (also called methods) are defined inside the class. Variables declared inside a method definition are called local variables. Within a method body, formal parameters are treated as local variables. The object according to which the method was called can be referenced in the method body using the this keyword. Similarly, the super keyword allows referencing methods and attributes declared in the superclass of the object class. Java allows single inheritance only. Thus, each class can have only one superclass. When a subclass is created, we have to define only the differences between that class and its superclass. By default, a new class with no explicit inheritance inherits from the Object class, the top class in Java. When an object is created, a slot for each non-static variable defined in the object class and in all its superclasses is reserved. Moreover, the object can access all the methods of its class and all the methods of its superclasses. An interface in Java refers to a collection of method signatures without actual definitions. Interfaces provide templates of behavior that other classes are expected to implement. A class, in Java, can implement one or more interfaces.

In Java programs, there are two data types: primitive types and reference types. Primitive types define a range of basic data values that can be stored in a variable. The reference types define references to objects of classes, which contain collections of variables and methods that are described by the classes. Primitive type data can be of either arithmetic or boolean type. Class, interface, and array types are examples of reference types. Any declared data in a Java program can be of any of the primitive or reference types. In the rest of the paper, we will refer to these three options as usage types. Standard Java documentation is discussed extensively in Refs. [8,11].

3. Overview of dynamic data flow analysis

Dynamic data flow analysis is a method of analyzing the sequence of actions on data in a program as it is being run. Fosdick and Osterweil [12] mention that there are three types of actions that can be performed on a data item, namely, define (d), reference (r) and undeﬁne (u). A variable is said to be deﬁned if a value is assigned to it; referenced if the value is fetched from the memory; and undeﬁned if the value becomes unknown or inaccessible. During program execution, a variable can be in one of the following four states: state D (deﬁned), state R (referenced), state U (undeﬁned), and state A (abnormal). Huang [13] introduced tracing the data ﬂow anomalies through state transitions instead of sequence of actions. When an action is applied on a variable, its state follows transitions according to the state transition diagram shown in Fig. 1. When a variable enters the abnormal state, this indicates a data flow anomaly. Thus, there are three anomalous paths PduP, PddP, and PurP, where P is an arbitrary path expression.

To detect data flow anomalies dynamically, Huang [13] introduced program instrumentation, which is achieved by inserting software probes into the original source program to collect information during the program execution.
4. Extension of dynamic data flow analysis

Unlike C++, Java programs do not offer a way to find the address or the size of a data in a Java program. Thus, the data hiding problem cannot be solved by tracking the use of absolute memory locations associated with the data in Java programs. Instead, we have to use the explicit data name itself. In this section, we show how to deal with data hiding problems in Java programs. First, we explain the information needed to be included in the inserted probes. Then, we discuss the locations at which the probes should be inserted. Next, we extend the state transition diagram. Finally, we discuss the dynamic data flow analysis method for Java programs.

4.1. Instrumented information

Since each of the usage types of Java variables has different semantics, we focus on each of them alone. Moreover, we consider some variables and operations that should be treated in a special way as follows.

4.1.1. Local variables

A local variable can be declared within a method body or can exist as a method’s formal arguments. In addition, it can be of a primitive or a reference type. The reference type variable can be either an object, or array or string (special kinds of objects).

To differentiate one variable from another, we have to consider its name and location. The location of a local variable contains the class where the method is declared and the method where the variable is declared. Since we may have overloaded methods, we have to consider the method name and the argument types. No two local variables at the same location can have the same name with different types. Thus, considering the variable type is not necessary. For example, in the following code:

```java
class A {
    void m(int i) { int j = 0; int k = 1; }
    void m() { int j = 2; }
    void n(int i) { int j = 1; }
}

class B { void m() { int j = 3; } }
```

For the definition of variable \( j \) in \( m(int\ i) \), if the variable name is not considered in the inserted probe, it will be ambiguous with the definition of variable \( k \) in the same method. Similarly, the ambiguity occurs with variables \( j \) in \( n() \), \( j \) in \( m() \) in class A, and \( j \) in \( m() \) in class B, if the method name, method argument types, and class name, respectively, are not included in the inserted probe.

For a local object variable, we have to consider each of its attributes separately. A further ambiguity is that attributes can be either instance or class variables, a point that we will discuss later in this paper.

4.1.2. Instance variables

Instance variables are dealt with as attributes of objects declared as local variables. For such attributes, we have to consider two arguments in addition to the attribute name and location (at which the object is declared). The first is the object name; the second is the class or the interface at which the attribute is declared. We will refer to this class as the actioned class/interface. The actioned class/interface can be the object type or one of its superclasses or interfaces. In the following example:

```java
class A { int i; void m(int i) { A a = new A(); a.i = i; }
    class B extends A{
        int i;
        void m(int i) { B b = new B(); a.i = i; }
        void m() { B a = new B(); a.i = 1; }
        void n(int i) { B a = new B(); a.i = i; }
    }

    class B { int m() { ... }
        int m(int j) { ... }
        int n() { ... }
        void n() { A a1 = new A(); A a2 = new A(); int j = a1.m(); ... }
    }

    class B { int m() { ... }
        void n() { A a1 = new A(); ... }
    }
```

For the definition of variable \( i \) in B.m(int i), if the variable name is not considered, it will be ambiguous with the definition of the instance variable \( j \) in class B. Similarly, the ambiguity occurs with instance variables \( i \) in \( n(int\ i) \), \( i \) in \( m() \), and \( i \) in \( m(int\ i) \) in class A, attribute \( i \) for object \( b \), and overridden attribute \( i \) declared in class A, if the method name, method arguments, class name, object name, and actioned class name, respectively, are not included in the inserted probe.

Also, we can consider the returned variables of the methods as instance variables. In this case, we do not have a variable name. To identify this variable, we should consider the method name, its arguments, and the class at which it is declared (actioned class). Moreover, we have to consider the object name—a variable which is considered as one of the object’s attributes—and the object location.

```java
class A {
    int m() { ... }
    int m(int j) { ... }
    int n() { ... }
    void n() { A a1 = new A(); A a2 = new A(); int j = a1.m(); ... }
}

class B {
    int m() { ... }
    void n() { A a1 = new A(); ... }
}
```

If, in the reference of returned variable by method \( m() \) in the calling statement \( j = a1.m() \), the method name is not considered, it will be ambiguous with the returned value of method \( r \). Similarly, if the method arguments, the actioned class, and the object name and location, respectively, are not considered in the inserted probe, the ambiguity occurs with the returned variable of method \( m(int\ j) \) in A, the returned variable of method \( m() \) in B, the returned
variable from \texttt{m()} in A for object \texttt{a2} in \texttt{A\_n()}, and the returned variable from \texttt{m()} in B for object \texttt{a1} in \texttt{B\_n()}.

4.1.3. Class variables

Class variables are global for all objects of the class or the interface at which the variables are declared. Therefore, to identify a class variable we only need to know its name and the class or interface at which it is declared (actioned class/interface). Consider the following example:

```java
class A {
    static int j;
    static int k;
    void m(int i) {
        A a1 = new A();
        A a2 = new A();
        a1.j = 1;
    }
}

class B {
    static int j;
}
```

For the definition of variable \texttt{a1.j} in \texttt{m()}, if the variable name is not included in the inserted probe, it will be ambiguous with the instance variable \texttt{k}. Similar ambiguity occurs with class variable \texttt{j} in class \texttt{B} if the actioned class is not determined. Adding the object name and the method name and arguments is not necessary, because the variable is global (i.e. variables \texttt{a1.j} and \texttt{a2.j} share the same memory location).

4.1.4. Class/interface fields

Class/interface fields are members of class/interface. When a Java program is executed, these fields occupy part of the memory. When an object is created, the defined fields are referenced to define the corresponding attributes of the object. As with identification of class variables, identification of a class/interface field is achieved through an inserted probe that contains the field name and the class/interface name.

4.1.5. Reference variables

When a variable with a reference type is assigned to another variable, the two variables share the same memory location. Therefore, instead of inserting probes that indicate definition and reference actions included in the assignment statement, we can instrument the following probe: (\texttt{variable1 = variable2}). The use of such a probe in the analysis process is discussed in Section 4.4.

4.2. Placement of probes

For dynamic testing purposes, probes should be inserted within the code of the program under test. The probe should include the set of information required to identify the variable on which the action is done and the type of action. In Java, actions occur inside class bodies and inside or outside their method bodies. Actions outside the method bodies appear at the variable declaration statements. However, inserting a probe statement outside method blocks is, syntactically, an error. Thus, all inserted probes, including those for actions outside method blocks, must be within method bodies. To simplify our analysis, we will consider the following cases.

4.2.1. Case 1

At the beginning of the first executed method (e.g. \texttt{main} or \texttt{paint}) block, all actions held in the declaration statements of all class/interface fields of all classes and interfaces in the program have to be considered. Moreover, if such a method accepts an input parameter, the definition action of the parameter has to be conditionally considered. The condition should check if a value is passed. In the following example:

```java
class A {
    static int j;
    static int k = 1;
    int m = 1;
    public static void main(String args[]) { .... }
}
```

we have to consider the definition of the \texttt{k} and \texttt{m} class fields at the beginning of the main method block. The definition of class variable \texttt{j} is not considered because its declaration does not hold a definition. A conditional statement that checks for the length of \texttt{args} array should be inserted. If the array is not empty, a definition action for that array should be considered.

4.2.2. Case 2

At the beginning of the body of any method, only definition actions of all method arguments of primitive types need be considered. This is because when a method is called, all its primitive type arguments are assigned variables and all reference type arguments point to the passed parameter memory locations. Consider the following example:

```java
void m(int j, int[] k) {...}
```

At the beginning of the method block, only the definition of variable \texttt{j} has to be considered since it is of a primitive type. The \texttt{k} array is of a reference type, so its definition does not have to be considered.

4.2.3. Case 3

Within a method code, a probe statement should follow each statement that includes action(s) for any variable. In the following example:

```java
int m() { int j = 1, k; k = j; ...
```

we have to consider the definition of variable \texttt{j} after the declaration statement. Thus, after the \texttt{k = j;} statement, the definition and the reference to variables \texttt{k} and \texttt{j}, respectively, should be considered.
4.2.4. Case 4

The `return` statement includes two actions: the reference for the returned variable and the definition of the data returned by the method. However, if the returned value is constant, we can ignore its reference action. Since the `return` statement is the last executed statement in the method, we have to consider the above actions before the `return` statement. Consider the following example:

```java
int m(int j) { if (j != 1) return j; else return 1; }
```

Before the first `return` statement, we have to consider the reference action of the variable `j` and the definition action of the `int` variable returned by the method. However, for the second `return` statement, it is not necessary to consider the reference action since the returned value is constant.

4.2.5. Case 5

Within a method code, if any object is declared as a local variable, probes for all actions on its non-global attributes included in the declaration statement, must be inserted before the declaration statement. Before these probes, we have to consider the reference actions of the object type fields. In the following example:

```java
class A {
    int j = 10; int k; static int n = 10;
    void m() { A a = new A(); .... }
}
```

before the declaration of object `a`, the hidden reference and definition of fields `j` in the class and `a.j`, respectively, have to be considered. No actions need be considered for variables `k` and `n`, since `k` is not defined and `n` is a global variable (class variable).

4.2.6. Case 6

If a method with a returned value is called, we have to instrument a probe after the calling statement to indicate the reference action of the returned variable. Moreover, if the returned variable is not of a reference type, we have to instrument a probe with a killing action for the returned variable after the calling statement. In the following example:

```java
class A {
    void m1() { .. }
    int m2() { return 1; }
    int[] m3() { int a[] = new int[2]; ... return a }
    void m4() {
        A a = new A();
        int j, b[] = new int[2];
        a.m1();
        j = a.m2();
        b = a.m3();
    }
}
```

We should not consider any action for the calling of `m1()` because it does not return a value. However, for the returned values of `m2()` and `m3()`, it is necessary to consider the reference actions before each calling statement. Moreover, we have to consider the killing action of the returned value of `m2()` after its calling statement, because it is of a primitive type. The killing action of the returned value of `m3()` need not be considered since it is of a reference type.

4.2.7. Case 7

If a method with a reference argument, is called and the passed reference type argument at the calling statement holds definitions, then these definitions have to be considered as actions on the variable name declared in the called method signature. The probes for these actions should be inserted before the calling statement. In addition, we have to consider the killing actions of all the reference-type elements (array elements or object instance variables) after the calling statement, since these elements will no longer be accessible. Consider the following example:

```java
class A {
    int j = 1;
    void m1(A a) { .. }
    void m2() { this.m1(new A()); }
}
```

In method `m2()`, we have to consider the definition and the killing of variable `a.j` and instrument the probes for these actions before and after the calling statement, respectively.

4.2.8. Case 8

If the `finalize()` method is called for an object, then we have to instrument probes for the killing actions of all non-global attributes of that object. These probes should follow the finalize method calling statement.

4.2.9. Case 9

Probes for killing actions of all non-global variables declared within a method body and before a return statement should be inserted. This case includes also, all method primitive type arguments. If a method has more than one return statement, the killing action probes should precede each one of them. If a method does not return a value, the probes should, immediately, precede the closing bracket of the method block. In the following example:

```java
class A {
    int j; static int k = 1;
    void m(int p, int[] r) { int n; A a = new A; }
}
```

Probes for killing actions of the local variables `p`, `n`, and `a.j` should be inserted at the end of the method block. However, a probe for killing action of the variable `a.k` should not be considered because it is a global variable. In
addition, we need not consider the killing action of rf since it is an argument passed by reference.

4.2.10. Case 10
Before the last executable statement in the first executed method (e.g. main or paint), the probes for killing actions of all class/interface fields in the program have to be inserted.

4.2.11. Case 11
Before a reference variable assignment statement, an assignment probe should be inserted. In the following code:

```java
class A {
    A m1(a A) { A a1 = new A(); ... return a1; }
    void m2() {A a2 = new A(); A a3 = m1(a2); }
}
```

Two probes with the (a = a2) and (a3 = a) arguments should, respectively, precede and succeed the calling statement for the m1 method.

4.3. Extension of state transition diagram
Because of the encapsulation property of the object-oriented programming languages, there might be some hidden actions that are not meant by normal programming. Consider the following example:

```java
class A { int j = 1; }
class B extends A {
    int j = 2;
    void m() { B b = new B(); .... }
}
```

The variable j in class B overrides the one in class A. However, when creating the object b, the two variables are defined and accessible (the overridden one can be accessed using the super keyword). Normally, we access only the override variable. Thus, if the overridden one is implicitly defined, not referenced, and then killed, it cannot be considered an anomaly case.

We denote such abnormal implicit definitions as id. When a variable has such definition, it is in an ID state. If a variable in an ID state is explicitly defined without being referenced, it enters State A. However, it is not an anomaly to reference or to kill such a variable when it is at an ID state. Fig. 2 shows the extended state transition diagram.

4.4. Analysis method
To overcome the data flow analysis difficulties for Java programs, we divide the analysis process into three steps. First, data is tabulated; then, all necessary information is instrumented; and, finally, the sequence of actions is dynamically tracked.

4.4.1. Step 1
To facilitate the instrumentation process, we construct three tables. These tables summarize the hierarchy and hidden data associated to the objects, classes, and interfaces contained in the program under test.

The first table is the instance/class variable table. This table summarizes the states of all declared instance and class variables of all classes and interfaces. It contains four fields: the class/interface name; the variable name; the usage type (class or instance); and actions (within the variable declaration statement).

The second table is the Inheritance table, which summarizes the classes and interfaces hierarchies. The table has three fields: the class/interface name; the name of its immediate superclass; and the name(s) of its immediate superinterface(s).

The Object table is the third table. It includes information about each created object in the program under test. The object name, the object type (class or interface name), and the location at which the object is declared (class and method) are the three fields of the table.

When an action is taken for an attribute of an object in a method, we can search for its object type using the object table. To find the actioned class/interface, we attempt to match the attribute name and the object type with an entry in the instance/class variable table. If it matches then the object type is the actioned class/interface. Otherwise, we find the superclass/superinterface of the object type by using the inheritance table, and we then use the instance/class variable table as outlined above. We do this repeatedly until finding the actioned class/interface.

4.4.2. Step 2
In this step, we instrument the code of the program under test. As a result of what is discussed in Section 4.2, each probe contains the following tuple (v, u, m, r, c, o, mn, mr, mc, ac, t). The first five arguments denote the variable name, its usage type, name of the method at which it is declared, the method arguments, and the name of the class where the method is declared, respectively. If the variable is an attribute of an object, then m, o, and ac denote the method at
which the object is declared, the object name, and the actioned class, respectively. For the returned variable of a method, the $mn$, $mr$, and $mc$ denotes the method name, the method arguments, and the class at which the method is declared, respectively. The argument $t$ denotes the type of the action. In some cases, some of the probe arguments are not used (e.g. for local variables, we use the last and the first five arguments only).

Mostly, determining these parameters statically is simple and straightforward. However, in some cases the object name is variable and not fixed. Consider the following case:

```java
int m1(int k) { this.j = k + 1; return k; }
void m2() { ... n = a1.m1(2); e = a2.m1(3); ... }
```

The name of the referenced object by `this` keyword is not fixed, but determined outside the method at the method calling statement. In addition, the returned value is an instance variable of the object which the method was called on. Similarly, the object name is not fixed. To overcome this problem, we have to create a new class that has the following declarations:

```java
class determineObject {
    static String objectName, methodName, methodArgs, className;
}
```

The class should be added only for programs that have such cases. Since the class fields are of class usage type, they are global for the whole program.

When a method, which has a returned value or uses `this` or `super` keywords, is called on an object, an object of `determineObject` class type must be created. The object fields are then used at the called method to identify the returned value or the object referenced by `this` or `super` keywords.

For methods other than the first executed method, the return statement or the usage of `this` or `super` keywords may be preceded by a calling statement for a method. In the following example:

```java
int m1() { ... k = a1.m2(); return k; }
int m2() { ... }
```

A calling statement for method $m2()$ precedes the return statement. When method $m1()$ is called, the name and the location of the object at which the method is called on are assigned to the added object fields. These arguments are used to identify the returned variable of method $m1$. However, the object fields are assigned different contents before the calling statement that precede the return statement. To solve this problem, we have to assign the object attributes contents to temporary locations before changing them and reassign them to their original contents after the calling statement. Since the first executed method does not have a returned value and cannot use `this` or `super` keywords, it is not necessary to consider the above problem.

### 4.4.3. Step 3

In this step, we execute the instrumented program, which prints the sequence of actions of program data. The printed probes are grouped in such a way that each group contains all probes for actions on a memory location associated with a variable. In this step, we have to consider reference variables that share the same memory location but have different names (determined in the assignment probes) in one group. If one such variable points to another memory location, the succeeding actions should be contained in a new group. Finally, we track the sequence of actions of each group using the extended state transition diagram.

### 5. Example

In this section, we consider the example listed in Appendix A. In this example, there are four classes. Class A inherits the Object class and have two instance variables, two class variables, and one method. Class B inherits class A and defines an additional instance variable. Class C inherits class B and defines an object instance variable. Finally, class ex inherits class C and implements two methods. In the following, we apply the three steps discussed in Section 4.4 to analyze the program, dynamically.

### 5.1. Step 1

In this step, we construct the three tables—Tables 1–3.
5.2. Step 2

In this step, we instrument the program. Since method \textit{m1} has a returned value and uses \textit{this} keyword, we add the \textit{determineObject} class. The instrumented program is listed in Appendix B. The reason for adding each probe is added as a comment following the instrumentation statement.

5.3. Step 3

When the program with probes is executed, the sequence of actions on program variables is printed as shown in Appendix C. We then group the sequence of actions on each memory location as shown in Table 4. Finally, we use the extended state transition diagram to detect the program data flow anomalies. As a result, the example contains two \textit{du} anomalies for variables numbered 3 and 8.

6. Conclusion and future work

In this paper, we have proposed an extension to the conventional dynamic data flow analysis to test Java programs. We show how to overcome problems with data hiding caused by classes, interfaces, and objects. The analysis is performed in three steps: construction of tables; instrumentation; and the tracking of the program under test data actions. Three types of instrumented statements are introduced: probes for data actions; object assignments; and those for the additional class that solve the non-fixed object name references. Consequently, the state transition diagram is extended to suit our goals. In our future research, we plan to extend our work to test Java programs involving concurrent processes.

Acknowledgements

We would like to acknowledge the support of this work by Kuwait University Research Grant EE112. We would also like to thank the anonymous referees for their useful comments.

Appendix A

The following code is, for the example, Java program:

```java
import java.awt.*;
class A { int j; int n = 1; static int k = 1; static int m; int m1(A a) {
int n = a.k;; if (this.n = n) System.out.println("true"); return n;
} class B extends A { int n = 10; } class C extends B {
B b = new B();
}
```

Table 3

<table>
<thead>
<tr>
<th>Object name</th>
<th>Object type</th>
<th>Location: class/method</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>a</td>
<td>A</td>
<td>A[m1(A)]</td>
</tr>
<tr>
<td>e</td>
<td>Ex</td>
<td>ex[m2(ex,int)]</td>
</tr>
<tr>
<td>x</td>
<td>Ex</td>
<td>ex[main(String)]</td>
</tr>
</tbody>
</table>

Table 4

<table>
<thead>
<tr>
<th>v</th>
<th>u</th>
<th>m</th>
<th>r</th>
<th>c</th>
<th>o</th>
<th>mn</th>
<th>mr</th>
<th>mc</th>
<th>ac</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>k</td>
<td>Class</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>A</td>
<td>dru</td>
</tr>
<tr>
<td>2</td>
<td>n</td>
<td>Instance</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>dru</td>
</tr>
<tr>
<td>3</td>
<td>n</td>
<td>Instance main</td>
<td>(String)</td>
<td>ex</td>
<td>x</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>B</td>
<td>du</td>
</tr>
<tr>
<td>4</td>
<td>n</td>
<td>Instance main m2</td>
<td>(String) (ex,int)</td>
<td>ex</td>
<td>x.b</td>
<td>e.b</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>n</td>
<td>Instance</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>dru</td>
</tr>
<tr>
<td>6</td>
<td>n</td>
<td>Instance main</td>
<td>(String)</td>
<td>ex</td>
<td>x</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>A</td>
<td>idru</td>
</tr>
<tr>
<td>7</td>
<td>n</td>
<td>Instance main</td>
<td>(String)</td>
<td>ex</td>
<td>x.b</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>A</td>
<td>idu</td>
</tr>
<tr>
<td>8</td>
<td>n</td>
<td>Instance m1</td>
<td>(A)</td>
<td>A</td>
<td>a</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>A</td>
<td>du</td>
</tr>
<tr>
<td>9</td>
<td>n</td>
<td>Local m1</td>
<td>(A)</td>
<td>A</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>dru</td>
</tr>
<tr>
<td>10</td>
<td>–</td>
<td>Instance main</td>
<td>(String)</td>
<td>ex</td>
<td>x</td>
<td>m1 (A)</td>
<td>A</td>
<td>A</td>
<td>dru</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>j</td>
<td>Instance m1</td>
<td>(A)</td>
<td>A</td>
<td>a</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>A</td>
<td>u</td>
</tr>
<tr>
<td>12</td>
<td>j</td>
<td>Local m2</td>
<td>(ex,int)</td>
<td>ex</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>dru</td>
</tr>
<tr>
<td>13</td>
<td>j</td>
<td>Instance m2 Main</td>
<td>(ex,int) (String)</td>
<td>ex</td>
<td>e</td>
<td>x</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>A</td>
</tr>
<tr>
<td>14</td>
<td>j</td>
<td>Instance main</td>
<td>(String)</td>
<td>ex</td>
<td>x.b</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>A</td>
<td>u</td>
</tr>
<tr>
<td>15</td>
<td>Args</td>
<td>Local main</td>
<td>(String)</td>
<td>ex</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>u</td>
</tr>
<tr>
<td>16</td>
<td>j</td>
<td>Instance</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>A</td>
<td>u</td>
</tr>
</tbody>
</table>
class ex extends C
{
    void m2(ex e, int j)
    {
        e.j = e.k;
        e.m = e.j;
        if (e.b.n != e.b.m) System.out.println("false");
        if (e.b.n != j) System.out.println("false");
    }
}
public static void main(String args[])
{
    ex x = new ex();
    if (x.m1(new A()) == 1) System.out.println("true");
    x.m2(x, 1);
}

Appendix B

The following is the instrumented version of the program:

import java.awt.*;

class determineObject {
    static String objectName, className, methodName, methodArgs;
}

class A {
    int j;
    int n = 1;
    static int k = 1;
    static int m;
    int m1(A a)
    {
        int n = a.k;
        System.out.println("(k,class,-,-,-,-,-,-,-,A,r)"); //case 3
        System.out.println("(n,instance,m1,(A),A,-,-,-,A,r)"); //case 3
        System.out.println("(m,class,-,-,-,-,-,-,-,A,d)"); //case 3
        if (this.n != n) System.out.println("true");
        determineObject d = new determineObject();
        System.out.println("(n,instance," +
                d.methodName + "," + d.methodArgs +
                "," + d.className + "," + d.objectName + ",-,-,-,A,r)"); //case 3
        System.out.println("(n,local,m1,(A),A,-,-,-,A,d)"); //case 3
        System.out.println("(n,instance,m2,(ex,int),ex,e,-,-,-,A,d)"); //case 3
        System.out.println("(j,instance,m2,(ex,int),ex,e,-,-,-,A,r)"); //case 3
        System.out.println("(j,instance,m2,(ex,int),ex,e,-,-,-,A,r)"); //case 3
        System.out.println("(j,instance,m2,(ex,int),ex,e,-,-,-,B,r)"); //case 3
        System.out.println("(j,instance,m2,(ex,int),ex,e,-,-,-,B,r)"); //case 3
        System.out.println("(j,instance,m2,(ex,int),ex,e,-,-,-,u)"); //case 9
    }
}

class B extends A {
    int n = 10;
}

class C extends B {
    B b = new B();
}

class ex extends C {
    void m2(ex e, int j)
    {
        System.out.println("(j,local,m2,(ex,int),
                ex,-,-,-,-,-,d)"); //case 2
        e.j = e.k;
        System.out.println("(" +
                "(k,class,-,-,-,-, -,-,-,A,r)" +
                "(j,instance,m2,(ex,int),ex,e,-,-,-,A,r)" +
                "(j,instance,m2,(ex,int),ex,e,-,-,-,B,r)" +
                "(j,instance,m2,(ex,int),ex,e,-,-,-,u)" +
                "); //case 3
        System.out.println("(n,instance,m2,(ex,int),ex,e,-,-,-,A,d)"); //case 3
        if (e.b.n != e.b.m) System.out.println("false");
        System.out.println("(m,class,-,-,-,-, -,-,-,A,d)"); //case 3
        System.out.println("(" +
                "); //case 3
        System.out.println("(" +
                "); //case 3
        System.out.println("(" +
                "); //case 3
        System.out.println("(" +
                "); //case 3
        System.out.println("(" +
                "); //case 3
        System.out.println("(" +
                "); //case 3
        System.out.println("(" +
                "); //case 3
    }
}

class B extends A {
    int n = 10;
}

class C extends B {
    B b = new B();
}

class ex extends C {
    void m2(ex e, int j)
    {
        System.out.println("(j,local,m2,(ex,int),
                ex,-,-,-,-,-,d)"); //case 2
        e.j = e.k;
        System.out.println("(" +
                "(k,class,-,-,-,-, -,-,-,A,r)" +
                "(j,instance,m2,(ex,int),ex,e,-,-,-,A,r)" +
                "(j,instance,m2,(ex,int),ex,e,-,-,-,B,r)" +
                "(j,instance,m2,(ex,int),ex,e,-,-,-,B,r)" +
                "(j,instance,m2,(ex,int),ex,e,-,-,-,u)" +
                "); //case 3
        System.out.println("(n,instance,m2,(ex,int),ex,e,-,-,-,A,d)"); //case 3
        if (e.b.n != e.b.m) System.out.println("false");
        System.out.println("(m,class,-,-,-,-, -,-,-,A,d)"); //case 3
        System.out.println("(" +
                "); //case 3
        System.out.println("(" +
                "); //case 3
        System.out.println("(" +
                "); //case 3
        System.out.println("(" +
                "); //case 3
    }
}

class B extends A {
    int n = 10;
}

class C extends B {
    B b = new B();
}

class ex extends C {
    void m2(ex e, int j)
    {
        System.out.println("(j,local,m2,(ex,int),
                ex,-,-,-,-,-,d)"); //case 2
        e.j = e.k;
        System.out.println("(" +
                "(k,class,-,-,-,-, -,-,-,A,r)" +
                "(j,instance,m2,(ex,int),ex,e,-,-,-,A,r)" +
                "(j,instance,m2,(ex,int),ex,e,-,-,-,B,r)" +
                "(j,instance,m2,(ex,int),ex,e,-,-,-,B,r)" +
                "(j,instance,m2,(ex,int),ex,e,-,-,-,u)" +
                "); //case 3
        System.out.println("(n,instance,m2,(ex,int),ex,e,-,-,-,A,d)"); //case 3
        if (e.b.n != e.b.m) System.out.println("false");
        System.out.println("(m,class,-,-,-,-, -,-,-,A,d)"); //case 3
        System.out.println("(" +
                "); //case 3
        System.out.println("(" +
                "); //case 3
        System.out.println("(" +
                "); //case 3
        System.out.println("(" +
                "); //case 3
    }
}

class B extends A {
    int n = 10;
}

class C extends B {
    B b = new B();
}

class ex extends C {
    void m2(ex e, int j)
    {
        System.out.println("(j,local,m2,(ex,int),
                ex,-,-,-,-,-,d)"); //case 2
        e.j = e.k;
        System.out.println("(" +
                "(k,class,-,-,-,-, -,-,-,A,r)" +
                "(j,instance,m2,(ex,int),ex,e,-,-,-,A,r)" +
                "(j,instance,m2,(ex,int),ex,e,-,-,-,B,r)" +
                "(j,instance,m2,(ex,int),ex,e,-,-,-,B,r)" +
                "(j,instance,m2,(ex,int),ex,e,-,-,-,u)" +
                "); //case 3
        System.out.println("(n,instance,m2,(ex,int),ex,e,-,-,-,A,d)"); //case 3
        if (e.b.n != e.b.m) System.out.println("false");
        System.out.println("(m,class,-,-,-,-, -,-,-,A,d)"); //case 3
        System.out.println("(" +
                "); //case 3
        System.out.println("(" +
                "); //case 3
        System.out.println("(" +
                "); //case 3
        System.out.println("(" +
                "); //case 3
    }
}
Appendix C

The following are the printed probes for the program example when it is executed:

(k.class,-,-,-,,-,-,A,u)
(n.instance,-,-,-,-,-,-A,d)
(n.instance,-,-,-,-,-,-B,d)
(n.instance,-,-,-,-,-,-A,r)
(n.instance,-,-,-,-,-,-A,r)
(n.instance.m1,(A),A,a,-,-,-A,d)
(k.class,-,-,-,-,-,-,-,-A,r)
(n.instance.main,(String),ex,x,-,-,-B,d)
(n.instance.main,(String),ex,x.b,-,-,-B,d)
(n.instance.main,(String),ex,x.b,-,-,-A,id)
(n.instance.main,(String),ex,x.b,-,-,-A,id)
(n.instance,-,-,-,-,-,-A,r)
(n.instance.m1,(A),A,a,-,-,-A,d)
(k.class,-,-,-,-,-,-,-,-A,r)
(n.instance.main,(String),ex,x,-,-,-A,r)
(n.instance.m1,(A),A,-,-,-,-r)
(n.instance.m1,(A),A,-,-,-,-r)
(n.instance.main,(String),ex,x.m1,(A),A,A,d)
(n.instance.m1,(A),A,-,-,-,-u)
(-instance.main,(String),ex,x.m1,(A),A,A,d)
(-instance.main,(String),ex,x.m1,(A),A,A,r)
(-instance.main,(String),ex,x.m1,(A),A,A,u)
(j.instance.m1,(A),A,a,-,-,-A,u)
(n.instance.m1,(A),A,a,-,-,-A,u)
e = x
(j.local,m2,(ex,int),ex,-,-,-,d)
(k.class,-,-,-,-,-,-,-,A,r)
(j.instance.m2,(ex,int),ex,e,-,-,-A,d)
(j.instance.m2,(ex,int),ex,e,-,-,-A,r)
(m.class,-,-,-,-,-,-,-A,d)
(m.class,-,-,-,-,-,-,-A,r)
(n.instance.m2,(ex,int),ex,e.b,-,-,-B,r)
(j.local,m2,(ex,int),ex,e.b,-,-,-B,r)
(n.instance.m2,(ex,int),ex,e.b,-,-,-B,r)
(n.instance.m2,(ex,int),ex,e.b,-,-,-B,r)
(n.instance.main,(String),ex,x,-,-,-B,u)
(n.instance.main,(String),ex,x.b,-,-,-B,u)
(n.instance.main,(String),ex,x.b,-,-,-A,u)
(j.instance.main,(String),ex,x,-,-,-A,u)
(j.instance.main,(String),ex,x.b,-,-,-A,u)
References