Automated Debugging in Java Programs Using HDM*

Hoon-Joon Kouh and Weon-Hee Yoo

School of Computer Science and Engineering, Inha University
#253 Yong-Hyun Dong, NamKu, Incheon, KOREA
hjkouh@hanmail.net, whyoo@inha.ac.kr

Abstract. Debugging has always been a costly part of software development. At the present time, Java programs locate logical errors using traditional debugging methods. Unfortunately, the traditional methods are often inadequate for the task of isolating specific program errors. This paper proposes a method to use HDM(Hybrid Debugging Method) for locating logical errors in Java programs. This method is a debugging technique that combines an algorithmic debugging method with a traditional step-wise debugging method. This approach can reduce the number of user's interaction. In this paper we describe the issues for HDM in the Java programs. Also, we walk through an example.

1 Introduction

Nowadays programs are becoming more large and complex than the past programs. Specially, the programs have many functions/methods and the number of statements in the functions/methods is increasing more and more. So debugging has always been a costly part of software development[4]. Users may spend much time in debugging the various programs using the traditional sequential debugging methods.

Techniques for debugging Java programs, until now, have been traditional debugging methods that are used at procedural-oriented programs. But debugging the object-oriented programs are not the same as procedural-oriented programs. Unfortunately, these traditional methods are often inadequate for the task of isolating specific program errors. So, many researches are in progress for efficiently debugging Java programs. These approaches tried to apply the existing techniques to Java programs. Recent works for efficiently debugging Java programs are slicing[3,8], query-based debugger[9], design by contract methodology[2] and model-based diagnosis[10]. The focus of these techniques is to reduce the user's interaction to the minimum.

This paper proposes a method to use HDM(Hybrid Debugging Method)[6] for locating logical errors in Java programs. HDM is a debugging technique that combines an algorithmic debugging method[1,5,7,11] with a step-wise debugging method. An algorithmic debugging method can easily locate functions/methods including a logical error. And a step-wise debugging method can easily locate statements including a

* This work was support by INHA UNIVERSITY Research Grant. (INHA-22000).

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2660, pp. 849–858, 2003.

logical error in the function/method. So HDM locates an error included in the functions with an algorithmic debugging method and locates the statements with an error with a step-wise debugging method in the erroneous function. The HDM can improve the drawback of an algorithmic debugging method and a step-wise debugging method. In this paper we describe the issues for parameter value presentation, constructor, method, Java API and member variables for HDM in Java programs. Also, we walk through an example.

The rest of this paper is organized as follows: In Section 2, we describe traditional debugging methods in Java and an algorithmic debugging method. In Section 3, we describe the functional structure of HDM and the algorithm. In Section 4, we present issues for HDM in Java programs. We walk through an example in Section 5, and conclude in Section 6.

2 Background

In this section, we describe the problem of the traditional debugging methods in Java programs and the concept of an algorithmic debugging method.

2.1 Traditional Debugging Methods in Java Programs

There are two ways in traditional methods for debugging logical errors in a Java program. First, the users can directly analyze a source program or insert the screen output instructions like "*System.out.println*" in the suspected location. But these methods are not easy to locate the logical errors because it is difficult for them to correctly anticipate the error location. Second, there is a step-wise debugging method. Users can sequentially debug a Java program with instructions like "*step-over*", "*step-into*", "*go*" and "*break-points*", but the method requires much time and much labor from the user because the user should execute all statements in the worst case.

2.2 Concept of an Algorithmic Debugging Method

An algorithmic debugging method is a semi-automated debugging technique that builds an execution tree from a source program and locates the logical errors included in the program from an execution tree.

The execution tree has the trace information such as function name or input/output parameter values etc. And the tree is built from the root to the leaves and from left to right. A called function becomes a child node of the calling function in the execution tree. And then the algorithmic debugging method traverses an execution tree using the top-down method and interacts with the user by asking about the expected behavior at the each node. The user can answer "correct" or "incorrect". And an algorithmic debugging automatically locates the erroneous functions from the information. The "correct" answer means that the current and child functions don't have any logical error. The "incorrect" answer means that the current or child function has one or more logical errors. So if the debugging process stops the traversal, some function *f* may have

some errors in two cases: In one case, the function *f* contains no function call. In the other case, all function called in the function *f* satisfy user's expectation.

3 HDM (Hybrid Debugging Method)

HDM is a debugging technique that combines an algorithmic debugging method with a step-wise debugging method.

This method debugs as follows. First, HDM system translates a source program into a program transformation tree and builds an execution tree from the tree. Second, the debugging system locates a node(function) including an error using an algorithmic debugging method. Third, the HDM system locates a statement including an error with a step-wise debugging method in the erroneous function through an editor view(see Fig. 1). Fourth, the user rebuilds an execution tree again and can debug from the root node of the program to a erroneous node or a last node. Finally, HDM iterates from first step to fourth step until there are no errors in a Java program.

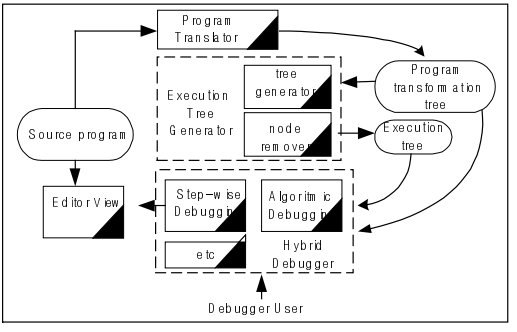

Fig. 1. The functional structure of HDM system

4 Issues for Java Programs

In this section, we describe issues for debugging Java programs using HDM. The issues are parameter values, constructor, method, Java API and member variables. We build an execution tree with the calling relationship between methods(including constructor) instead of functions in Java programs.

4.1 Parameter Value Presentation

An object is a class instance and receives information through the parameters of a constructor when the object is created by a class instance creation expression. Also an object does behave the desirable jobs using methods. At this time the method sends and receives information through many parameters. The primitive data types of these parameters are described as follows.

- Boolean variables are True or False at the input/output variables.
- For integer variables (byte, short, integer, long), the input variables are represented as binary digit, octal digit, decimal digit, hexadecimal digit and the output variables are represented as all decimal digits.
- For float and double variables, the input variables are represented as decimal digit, exponentiation and the output variables are represented as decimal digits.
- Character variables are represented as 'character'.

Parameter values with reference data types are described as follows.

- String object variables are represented as 'strings'.
- Array object variables are represents as sequence of elements. For example, one dimensional array with n+1 elements is represented as $[a_0, a_1, a_2,\ldots, a_n]$. An element a_i is i+1 element of the array
- Class object variables are represented as pairs of variable and value about the member variables. For example, an object with member variables $x_1, x_2, x_3,\ldots, x_n$ and values $v_1, v_2, v_3, \ldots, v_n$ is represented as $(x_1=v_1, x_2=v_2, x_3=v_3,\ldots, x_n=v_n)$.

4.2 Constructor

A constructor is executed the only once when an object is created from class. And it is mostly used to initialize member variables.

To build an execution tree, we assume that the left-hand side variables of the assignment in the constructor are all output variables. Fig. 2 shows an example of a constructor.

```
class SimpleCal  {
     int num1 ,num2;
     public SimpleCal() {
          this(0,0);
     }
     public SimpleCal(int num1, int num2) {
          this.num1=num1;   this.num2=num2;
     }
}
```

Fig. 2. An example of constructor in class *SimpleCal*

The class *SimpleCal* has two constructors in Fig. 2. They are overloaded constructors. The first constructor has no parameters. And the second constructor has two parameters and assigns the parameter values to two member variables. The instructions that create objects *Obj1*, *Obj2* from *SimpleCal* is as follow.

```
SimpleCal Obj1 = new SimpleCal();
SimpleCal Obj2 = new SimpleCal(3, 4);
```

The process that creates object *Obj1,Obj2* for building an execution tree is as follows. The method "*this()*" calls a constructor that has two parameters, input values 0 and 0. And the output variables are all the left-hand side variables of the assignment.

Obj1.SimpleCal();
Obj1.SimpleCal(in:num1=0,num2=0 out:this.num1=0,this.num2=0);
Obj2.SimpleCal(in:num1=3,num2=4 out:this.num1=3,this.num2=4);

An execution tree that is built from these objects is shown in Fig. 3. The nodes of an execution tree are constructed from the information of constructor. The tree makes the hierarchical structure from main method according to the execution order. The HDM system doesn't build the node of no input/output value like a method *Obj1.SimpleCal()*. Two nodes have the same method name, but we can identify the method from the object names and the input/output parameters.

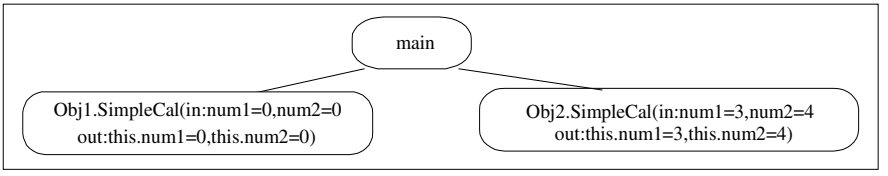

Fig. 3. An execution tree that is built from the constructor *SimpleCal*

4.3 Method

A method defines object's behavior in a class. We define two methods(*incre*, *addsum*) in a class *SimpleCal* (Fig. 4).

```
class SimpleCal  {
     int incre(int num) {
            num = num + 1;   return num;
     }
     int addsum(int a, int b)  {
            int c;
            a = incre(a);  b = incre(b);
            c = a + b;  return c;
     }
}
```

Fig. 4. An example of method in class *SimpleCal*

Suppose that an object Obj1 calls a method *addsum* with arguments 3, 4 and the result value is assigned to a variable *sum,* then we can express the program such as follows in Java program.

sum = Obj1.addsum(3,4);

A method *addsum* adds two integer values and a method *incre* increases a integer variable by one. The execution of these statements is as follow.

Obj1.addsum(in:a=3,b=4 out:c=9);
Obj1.incre(in:num=3 out:num=4);
Obj1.incre(in:num=4 out:num=5);

An execution tree built from these methods is shown in Fig. 5.

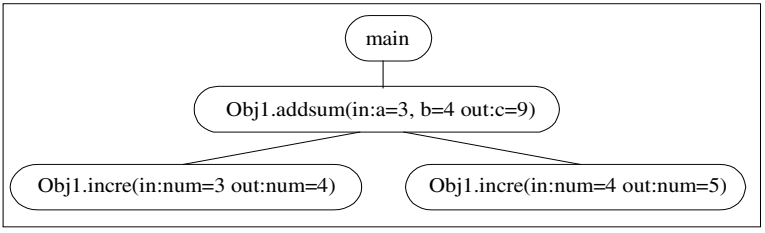

Fig. 5. An execution tree created from methods in a class *SimpleCal*

4.4 Java API

Java API has a lot of the built-in packages. Java has API that is libraries to provide much functions. Java programmers use the predefined software packages in Java API. The packages have a lot of classes and other packages. Also the classes have a great many primitives and library methods. They may reasonably be assumed to behave correctly. And then, debugging users should not be asked about a great many library methods included in classes. The HDM system doesn't build an execution tree with theses methods. Obviously, this will reduce the number of questions that are to be asked and the time taken to build the execution tree.

4.5 Member Variable

Java classifies member variables into an object variable, a class variable and a final variable. Also Java classifies an object variable into an object property variable and an object reference variable. These can be assigned or changed values in the methods of the class. So, changing of the variable's value becomes the important information to a user.

```
class Person {
  long idNum;
  static long personID=0;
  public Person() {
        idNum = personID;
        personID = personID + 1;
  }
}
class  Employee  {
  public static void main(String args[]) {
        Person p1 = new Person();
        Person p2 = new Person();
    }
}
```

Fig. 6. The example of member variables : idNum is an object property variable and personID is a class variable.

A class *Person* has an object property variable idNum, a class variable personID and a constructor *Person*. The class *Employee* instantiates objects p1 and p2 from the class *Person*. Whenever one object is created from the class of Fig. 6, personID is increased a value 1 in the constructor. And the value of the personID is assigned to idNum. But nodes of the class *Person* don't have parameters and only have member variables. So a user doesn't, generally, know that the values of member variables personID and idNum change in the constructor *Person*. We should add member variables to input/output information of each node at the execution tree as Fig. 7.

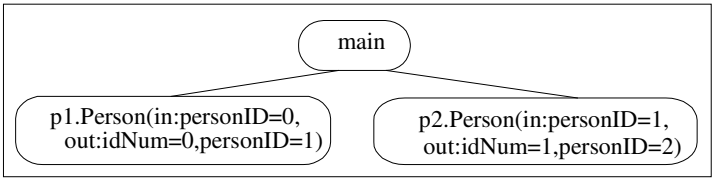

Fig. 7. The information about the definition and assignment of the member variable is added to each node of an execution tree

Also, if the variables are classified to an object property variable, an object reference variable and class variable in the nodes, the user would easily understands the constructor and the result. And so, we define the member variables as follows.

- If an object property variable is defined in the method/constructor, the variable uses a symbol OpvIn(object property input variable). And if an object property variable is used in the method/constructor, the variable uses a symbol OpvOut(object property output variable).
- If an object reference variable is defined in the method/constructor, the variable uses a symbol OrvIn(object reference input variable). And if an object reference variable is used in the method/constructor, the variable uses a symbol OrvOut(object reference output variable).
- If a class variable is defined in the method/constructor, the variable uses a symbol CvIn(class input variable). And If a class variable is used in the method/constructor, the variable uses a symbol CvOut(class output variable).

From the definition of the upper part, the execution tree is modified from Fig. 7 to Fig. 8. Then debugger user can get the correct debugging information from the execution tree.

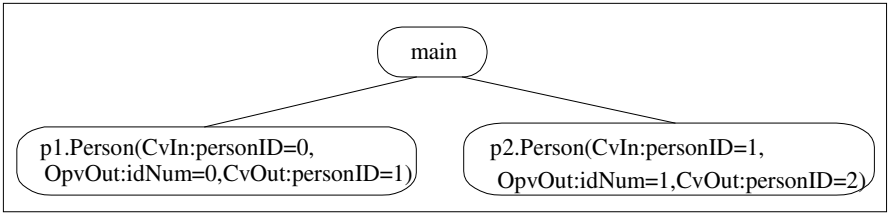

Fig. 8. The information about the kind of the member variables is added to each node of an execution tree

5 An Example of Applying HDM in Java Programs

Now, we can explain the HDM with an example. Fig. 9 shows a Java program with two classes(*SimpleCal*, *Calculation*). A class *SimpleCal* has two constructors and three methods. A class *Calculatio*n has one main method. We assume two cases for the debugging test.

- There is an error in the method *incre*. A statement "num=num-1" is a logical error. And then a statement "num=num+1" is correct.
- There is an error in the method *totalsum*. A statement "var2=var1-num" is a logical error. And then a statement "var3=var2+num" is correct in *totalsum*.

```
class SimpleCal  {
    int num;
    public SimpleCal()
    {    this(0);    }
 public SimpleCal(int num)  {
      this.num=num;
    }
 int incre(int num) {
   num = num - 1;
   return num;
 }
 int addsum(int a, int b)  {
   int c;
   a = incre(a);
   b = incre(b);
   c = a + b;
   return c;
}
```

```
    int totalsum(int a, int b) {
        int var1,var2;
        var1=addsum(a,b);
        var2= var1-num;
       return var2;
    }
   }
public class Calculation  {
public static void main(String args[])
{
     int a, b;
     SimpleCal Obj1 = new SimpleCal();
     a = Obj1.addsum(3,4);
     SimpleCal Obj2 = new SimpleCal(5);
     b = Obj2.totalsum(3,4);
     System.out.println(a+ "   " + b);
 }
}
```

Fig. 9. An example of a Java program

If we start to debug a Java program of Fig. 9, the HDM system builds an execution tree as Fig. 10. The user expects the result 9 and 14, but gets the result 5 and 0.

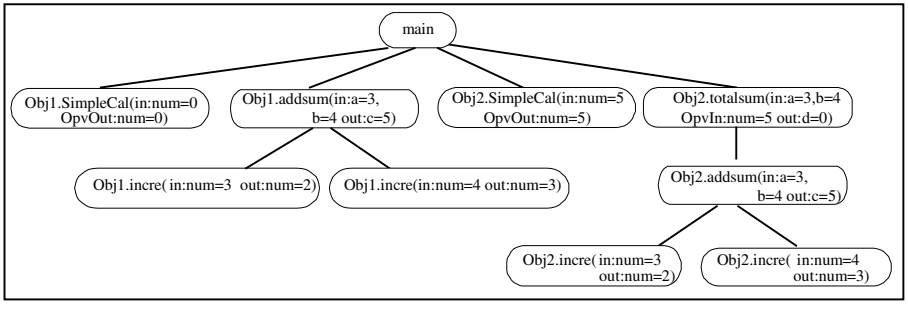

Fig. 10. An execution tree that is built from Fig. 9

The debugger user starts an algorithmic debugging at the HDM. The debugging process from the execution tree is as follows.

Obj1.SimpleCal(in:num=0 OpvOut:num=0)? $ correct
Obj1.addsum(in:a=3, b=4 out:c=5)? $ incorrect
Obj1.incre(in:num=3 out:num=2)? $ incorrect

HDM system locates an error in the method *incre* and shows the source program of the method *incre* in a text editor. The user can start a step-wise debugging and locate a statement including a logical error with "step-over", "go" and "break-points". The user can modify "num=num-1" to "num=num+1". A step-wise debugging stops when a user selects "stop" or the current method is returned. HDM rebuilds an execution tree and traverses an execution tree from the top node.

Obj1.addsum(in:a=3, b=4 out:c=9)? $ correct
Obj2.SimpleCal(in:num=5 OpvOut:num=5)? $ correct
Obj2.totalsum(in:a=3, b=4 OpvIn:num=5 out:var2=4)? $ incorrect
Obj2.addsum(in:a=3, b=4 out:c=9)? $ correct

HDM system locates an error in the method *totalsum* and shows the source program of the method *totalsum* in a text editor. A user can locate and modify a logical error, "var2=var1-num" to "var2=var1+num". Then HDM system restarts the program debugging and builds an execution tree again. When HDM system builds an execution tree again, he/she doesn't build the following nodes because they are the nodes that users answered in "correct" at the previous execution tree.

Obj1.SimpleCal(in:num=0 OpvOut:num=0);
Obj2.SimpleCal(in:num=5 OpvOut:num=5);

The user confirms that the program has no errors.

Obj1.totalsum(in:a=3,b=4 OpvIn:num=0 out:var2=9)? $correct
Obj2.totalsum(in:a=3,b=4 OpvIn:num=0 out var2=14)? $correct

There are no errors. A user has localized two errors with HDM in a Java program of Fig. 9. So HDM is a suitable technique for debugging Java programs.

6 Conclusion

In this paper we have suggested a HDM for debugging Java programs. The HDM is a debugging technique that combines a step-wise debugging method with an algorithmic debugging method. An algorithmic debugging method can easily locate functions/methods including a logical error. And a step-wise debugging method can easily locate statements including a logical error. So users can locate a method/constructor including a logical error with algorithmic debugging method and then locate a statement including a logical error with a step-wise debugging method in the erroneous method/constructor.

We proposed to build an execution tree with methods(including constructor) instead of functions for debugging a Java program. Also we defined the parameter values and solved the problem of the member variable for building an execution tree. And this paper has walk through an example that HDM is a suitable technique for debugging Java programs.

As a result, HDM reduced the number of the user's interaction than the traditional debugging methods in Java programs.

References

1. Z. Alexin, T. Gyim'othy and G. Kokai, IDT: Integrated System for Debugging and Testing Prolog Programs in Proceedings of the Fourth Symposium on Programming Languages and Software Tools, Hungary June 9-10, (1995) 312–323
2. M. Auguston, A language for debugging automation in: Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering, Jurmala, June 1994, Knowledge Systems Institute, pp. 108–115.
3. D. Bartetzko, C. Fischer, M. Moller and H. Wehrheim, Jass-Java with Assertions in Workshop on Runtime Verification, 2001. held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01
4. Z. Chen and B. Xu, "Slicing Object-Oriented Java Programs," ACM SIGPLAN Notices, V.36(4):pp 33–40 April, 2001
5. P. Fritzson, M. Auguston and N. Shahmehri, "Using Assertions in Declarative and Operational Models of Automated Debugging," The Journal of Systems and Software 25, 1994, pp 223–239
6. P. Fritzson, N. Shahmehri, M. Kamkar, T. Gyimothy, "Generalized Algorithmic Debugging and Testing," ACM LOPLAS – *Letters of Programming Languages and Systems.* Vol. 1, No. 4, December 1992.
7. H J Kouh, W H Yoo, Hybrid Debugging Method: Algorithmic + Step-wise Debugging in Proceedings of the 2002 International Conference on Software Engineering Research and Practice, June 2002.
8. G. Kokai, L. Harmath, and T. Gyim'othy, Algorithmic Debugging and Testing of Prolog Programs in Proceedings of ICLP '97, The Fourteenth International Conference on Logic Programming, Eighth Workshop on Logic Programming Environments Leuven, Belgium, 8-12 July 1997, 14–21.
9. G. Kovacs, F. Magyar, and T. Gyimothy, Static Slicing of JAVA Programs in Research Group on Artificial Intelligence(RGAI), Hungarian Academy of Sciences, Jozsef Attila University, HUNGARY, December 1996
10. R. Lencevicius, On-the-fly Query-Based Debugging with Examples in Proceeding of the Fourth International Workshop on Automated and Algorithmic Debugging, AADEBUG '2000, Munich, Germany, August 2000.
11. C. Mateis, M. Stumptner and F. Wotawa, Locating bugs in Java Programs-first results of the Java Diagnosis Experiments(Jade) project in Proceedings of IEA/AIE, New Orleans, 2000, Springer-Verlag.
12. H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages in Proceedings of PLILP'92 – Symposium on Programming Language Implementation and Logic Programming, Leuven, Belgium August 1992. LNCS 631, Springer Verlag.