Algorithmic Debugging of Java Programs

R. Caballero[1] ,[2]

Facultad de Informática
Universidad Complutense de Madrid
Madrid, Spain

C. Hermanns [3]

Institut für Wirtschaftsinformatik
Universität Münster
Münster, Germany

H. Kuchen[1] ,[4]

Institut für Wirtschaftsinformatik
Universität Münster
Münster, Germany

Abstract

In this paper we propose applying the ideas of declarative debugging to the object-oriented language Java as an alternative to traditional trace debuggers used in imperative languages. The declarative debugger builds a suitable computation tree containing information about method invocations occurred during a wrong computation. The tree is then navigated, asking the user questions in order to compare the intended semantics of each method with its actual behavior until a wrong method is found out. The technique has been implemented in an available prototype. We comment the several new issues that arise when using this debugging technique, traditionally applied to declarative languages, to a completely different paradigm and propose several possible improvements and lines of future work.

Keywords: Declarative Debugging, Object-Oriented Languages.

1 Introduction

Nowadays the concept of *encapsulation* has become a key idea of the software development process. Applications are seen as the assembly of different encapsulated software components, where each component can in turn be composed of other simpler components. The encapsulation here means that the programmer only needs to know *what* each component does (the semantics), without worrying about *how*

[1] This author has been funded by the projects TIN2005-09207-C03-03 and S-0505/TIC/0407.
[2] Email: rafa@sip.ucm.es
[3] Email: hermannc@wi.uni-muenster.de
[4] Email: kuchen@uni-muenster.de

this is done (the implementation). Object oriented languages such as Java [16] are based on this concept, allowing the definition of flexible, extensible, and reusable software components, thus saving time and avoiding errors in the final program.

While this view of the software as an assembly of components has been successful and it is widely used during the phases of design and implementation, it has had little influence on later phases of the software development cycle such as testing and debugging. Indeed, the debuggers usually included in the modern IDEs of languages such as Java, are sophisticated *tracers*, and they do not take advantage of the component relationships.

Declarative debugging, also known as *algorithmic debugging*, was introduced by E. Y. Shapiro in [14] as an alternative to trace debuggers for the Logic Programming paradigm, where the complex execution mechanism makes traditional debugging less suitable. The idea was afterwards employed in other declarative programming paradigms such as functional [11] and functional-logic [3] programming.

A declarative debugger starts when the user finds out some unexpected behavior while testing a program, i.e. an initial symptom. Then the debugger builds a tree corresponding to the computation that produced the initial symptom. Each node of this tree corresponds to the result of some subcomputation, and has a fragment of program code associated, namely the fragment of code needed for producing the result. The children of a node correspond to the results of those subcomputations that were necessary to obtain the parent result. In particular the root of the tree corresponds to the result of the main computation. The user navigates the tree looking for a node containing a non-valid result but whose children produced valid results. Such a node is considered a *buggy node* and its associated fragment of code is pointed out as erroneous because it has produced a wrong output result from the correct input results of its children.

In this paper we apply the idea of declarative debugging to the object oriented language Java. Starting from some erroneous computation, our debugger locates a *wrong method* in the debugged program. The task of the user during a debugging session is reduced to checking the validity of the results produced by some method calls occurring during the computation. Thus, our tool abstracts away the details of the implementation of each method, deducing the wrong method from the intended semantics of the methods and the structure of the program.

In order to improve the efficiency of the tool we propose using a test-case generator such as GlassTT [8,9,10]. This tool divides the possible input values of each methods into equivalence classes using some coverage criteria. The correct/incorrect behavior of a method call for any representative of an equivalence class will entail the validity/non-validity of the method for the other members of the class. Therefore the debugger can use this information in order to infer the state of several nodes from a single user answer. Although still in the early stages of study, we think that this improvement can dramatically reduce the number of nodes considered during a debugging session.

The idea of using declarative debugging out of declarative programming is not new. In 1989 N. Shahmehri and P. Fritzson presented in [13] a first proposal, further developed by the same authors in [5]. In this works a declarative debugger for the imperative language *Pascal* was presented. The main drawback of the proposal was that the computation tree was obtained by using a program transformation, which limited the efficiency of the debugger. The differences of our approach w.r.t. these earlier proposals are:

- Java is a language much more complex than Pascal. The declarative debugging of programs including objects and therefore object states introduces new difficulties that had not been studied up to now and that we tackle in this paper.
- Modern languages such as Java offer new possibilities for implementing the declarative debugger. In our case we have based the implementation of our prototype on the *Java Platform Debugging Architecture* (JPDA) [7]. JPDA has an event-based architecture and uses the method-entry and method-exit events in order to produce the computation tree. The result is a much more efficient tool.

A more recent related work [6] presents a declarative debugger for Java that keeps information about most of the relevant events occurred during a Java computation, storing them in a deductive database. The database can be queried afterwards by the user in order to know the state of the program (variables, thread, etc.) in different moments of the execution in order to infer where the bug is located. The main difference between both proposals is that our debugger concentrates only on the logic of method calls, storing them in a structured way (the computation tree) which allows the debugger to deduce the wrong method from the user answers.

In the next section we present the application of the ideas of declarative debugging to the case of Java programs. Section 3 introduces the idea of using a test-case generator in order to reduce the number of questions that the user must answer before finding the bug. In Section 4, we discuss the limitations of our prototype. Finally the work ends presenting some conclusions and future work.

2 Declarative Debugging

In this section we present the ideas of declarative debugging applied to the object-oriented language Java, and its prototype implementation.

2.1 Computation Trees

We start by defining the structure of the computation trees used by our debugger. Since the aim of the tool is detecting wrong methods in Java programs, each node of the tree will contain information about some method call which occurred during the computation being analyzed. Let N be a node in the computation tree containing the information of a method call for some method f. Then the children nodes of N will correspond to the method calls occurring in the definition of f that have been executed during the computation of the result stored at N. For example, consider a method f defined as:

```
public int f(int a) {    if (a>0) return g(a); else return h(a); }
```

Then any node associated to a method call for `f` will have exactly one child node, that will correspond to either a method call to `g` (if the parameter is positive) or to `h` (in other case). Thus, as we will see in the example of the next subsection, a method call occurring inside a loop statement can produce several children nodes.

This structure of the computation tree guarantees that a buggy node, i.e. a non-valid node with valid children, will correspond to a wrong method and that therefore the proposed debugging technique is correct. However checking the validity of a node in a Java computation tree is far more complex than for instance in a functional language. This is because apart of returning results, methods in object-oriented languages can change both the caller object and the parameters states. All this information must be available to the user at the moment of the debugging session in order to detect the validity of the nodes. Hence the information stored at each node of our computation trees will be:

- The full qualified name of the method corresponding to the call.
- The input values of the parameters and, in case of objects and arrays, the output values if they have been modified by the method.
- The returned value.
- The state (i.e. the attributes) of the caller object which contains the method. As in the case of the parameters both the input and the output states are needed in order to check the validity of the node.

In order to simplify the debugging, the debugger marks in a different color the names of the parameters and attributes changed during the method execution. The *object inspector* of the debugger allows the user to check the state changes in detail.

The next subsection presents an example of a debugging session using our declarative debugger prototype which will show these ideas in practice.

2.2 A Debugging Session Example

Figure 1 shows a program for ordering an array using the well-known *HeapSort* algorithm [4], which we will use as a running example. Heapsort first transforms an array to the well-known heap data structure (see below) and then successively picks the maximal element from it, while preserving the heap structure.

The intended semantics of each method in class HeapSort is the following:

- sort(a): sorts the elements of array a in ascending order. The result is stored in the attribute h.
- buildHeap(): rearranges the elements of array a in such a way that they form a heap h, i.e. $h(i) \geq h(2i+1)$ for $0 \leq i \leq h.length/2 - 1$ and $h(i) \geq h(2i+2)$ for $0 \leq i \leq h.length/2 - 2$.
- sink(int l, int r): starting from i=l element $h(i)$ is successively interchanged with the maximum of $h(2i+1)$ and $h(2i+2)$, until this maximum is not larger than $h(i)$ or the end of array h is reached.
- getMax(int r): picks the maximum at the root h(0) of the remaining heap h(0),. . . ,h(r) and maintains the heap structure of the remaining elements.

Method getMax includes an error in line 19 which should be h[0] = h[r];. We will use the declarative debugger for locating the error. The example also includes a class TestHeapSort for testing the class Heapsort. The class contains a method testSort which checks if the sort method of the Heapsort class orders a particular array correctly. The debugging session starts when the user runs the program

```
public class Heapsort {

  protected int[ ] h;

  public void sink(int l, int r){
    int i = l;
    int j = 2*l+1;
    int x = h[l];
    if (j<r && h[j+1]>h[j]) j++;
    while (j <= r && h[j]>x){
      h[i] = h[j];
      i = j;
      j = 2*j+1;
      if (j < r && h[j+1]>h[j]) j++;}
    h[i] = x;}

  public void buildHeap(){
    for(int i = h.length/2-1; i>=0; i–)
      sink(i,h.length-1);}

  public int getMax(int r){
    int max = h[0];
    h[0] = h[r-1];
    sink(0,r-1);
    return max;}

  public void sort(int[] a){
    h = a;
    buildHeap();
    for(int r = h.length-1; r>0; r–)
      h[r] = getMax(r);}
}
```

```
public class TestHeapSort {

  public static void main(String[] args){
    System.out.println(”Result: ”+testSort()); }

  public static boolean testSort(){
    boolean test = true;
    int sortedArray[] = {4,5,12,17,29,42,89,93};
    int testArray[] = {89,5,93,12,29,4,42,17};
    Heapsort alg = new Heapsort();
    alg.sort(testArray);
    for (int i=0; test && i<testArray.length; i++)
      test = test && (sortedArrayi]==testArray[i]);
    return test; }
}
```

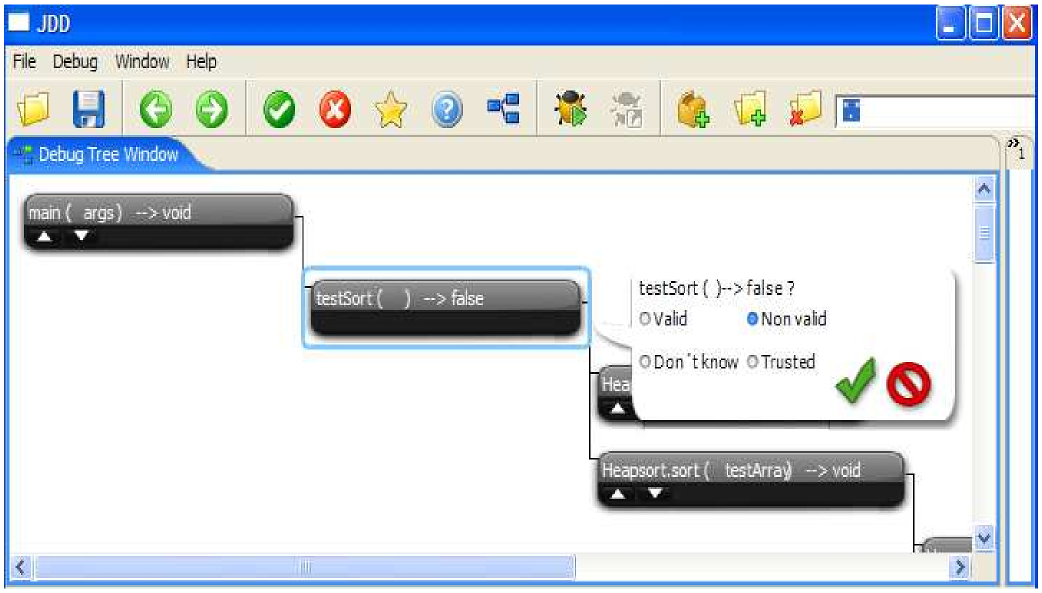
Fig. 1. Heapsort example.

obtaining an unexpected result false as outcome of the test. The debugger then repeats the computation producing a computation tree with 23 nodes. The root corresponds to the initial method call for main. Its only child corresponds to the method call to testSort, which returns the unexpected value false. The user then right-clicks over the node or uses the icons of the toolbar to mark this node as non-valid, as can be seen in Fig. 2.

From this starting point the user can select the option of automatic navigation, and the debugger will choose the nodes to be checked. These are the nodes involved in the debugging session of out example:

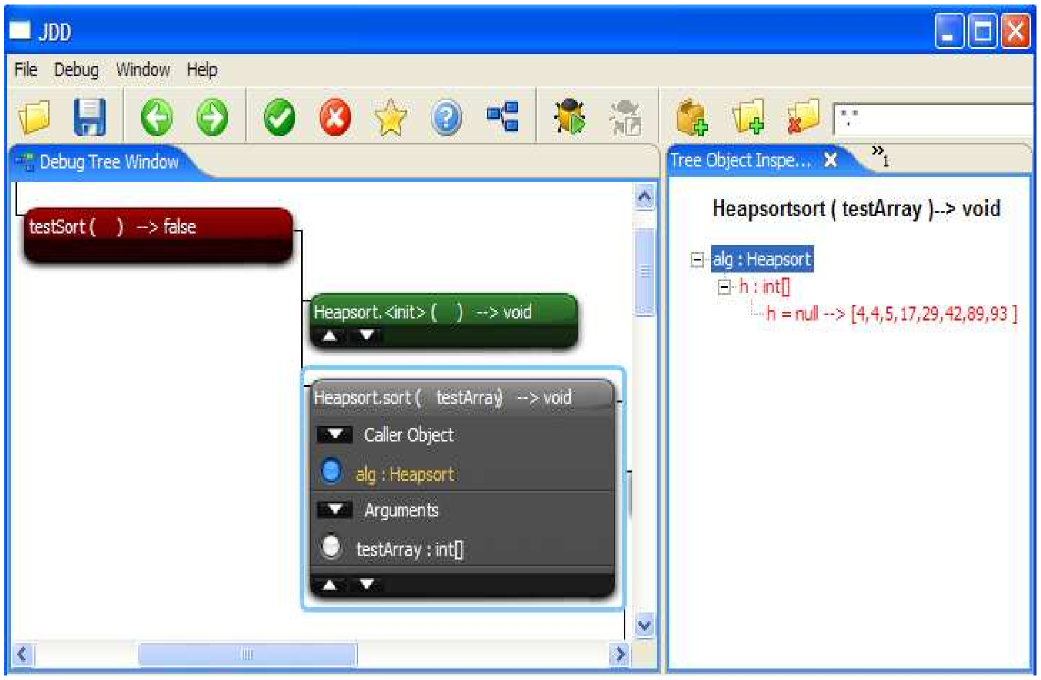
- The node Heapsort.Init(), which corresponds to the constructor of class Heapsort, is easily detected as valid. The navigation proceeds by the next sibling.
- The next node contains a call to the method Heapsort.sort(). Expanding the node (see Figure 3), the user checks the value of the argument testArray, which contains the initial value [89,5,93,12,29,4,42,17] . Then we click over the caller object alg, which is displayed in yellow to show that its state has changed after the method call:

  The object inspector (see Fig. 3, right hand side) shows that the attribute



testSort ( )--> false ?
Valid [✓ Non valid]
Don't know Trusted

Fig. 2. Screenshot of the declarative debugger.



Heapsortsort (testArray)--> void

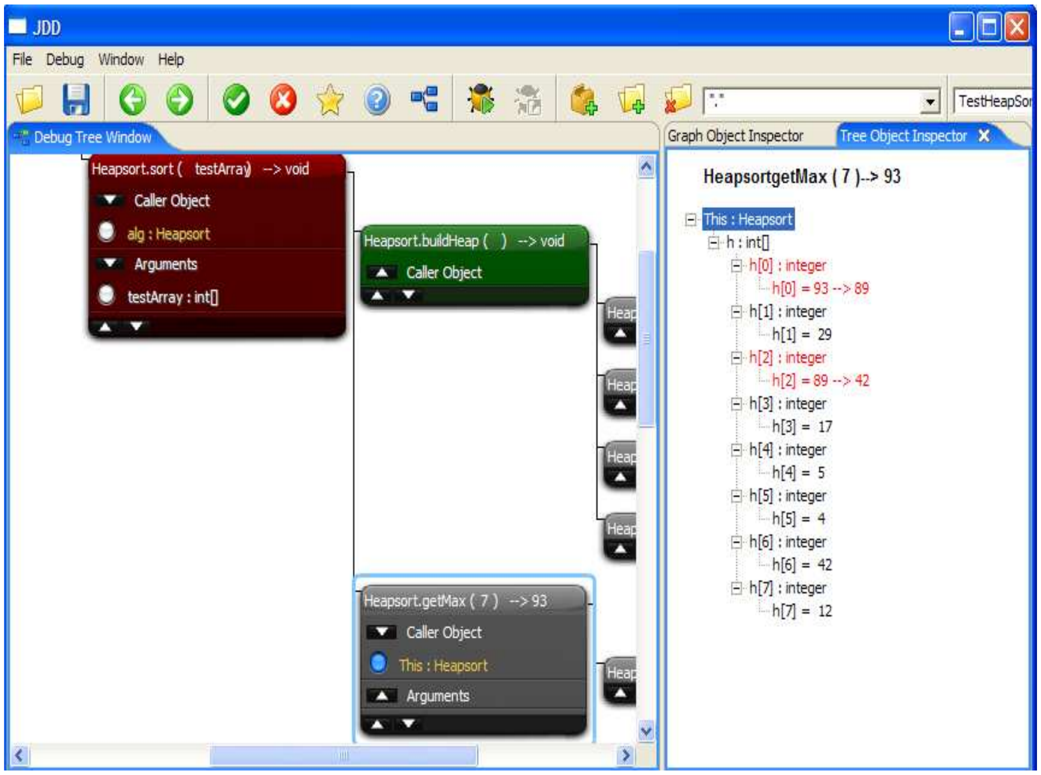
alg : Heapsort
h : int[]
h = null --> [4,4,5,17,29,42,89,93]

Fig. 3. Screenshot of the declarative debugger.

h has changed from the value null to [4,4,5,17,29,42,89,93]. The node is non-valid because it should contain the same elements as the input parameter but in ascending order. However, the value 12 is missing while 4 occurs twice. The navigation proceeds asking about the validity of the first child of this node.

- The next node contains a call to HeapSort.buildHeap(). The user checks that it is valid and the navigation proceeds by the next sibling.
- For the subsequent call to HeapSort.getMax(7), we check the value of attribute h

in the object inspector (see Fig. 4).



```
HeapsortgetMax ( 7 )--> 93
This : Heapsort
  h : int[]
    h[0] : integer
      h[0] = 93 --> 89
    h[1] : integer
      h[1] = 29
    h[2] : integer
      h[2] = 89 --> 42
    h[3] : integer
      h[3] = 17
    h[4] : integer
      h[4] = 5
    h[5] : integer
      h[5] = 4
    h[6] : integer
      h[6] = 42
    h[7] : integer
      h[7] = 12
```

Fig. 4. Screenshot of the declarative debugger.

- In Fig. 4 we see that h contained [93,29,89,17,5,4,42,12] at the moment of the method call, and that the values 93 and 89 where replaced by 89 and 42, respectively, after the method call. Thus the value of the attribute h at the end of the method is [89,29,42,17,5,4,42,12]. From the intended interpretation of the method we have that after a call to getMax(7) the attribute h should contain in its 7 first elements the resulting heap after eliminating its first element. The first 7 elements of h after the method call are [89,29,42,17,5,4,42], and the repetition of 42 means that the call is non-valid, because the value 42 was not repeated in [93,29,89,17,5,4,42,12].
- In a similar way the debugger proceeds asking about the validity of the only child of the previous node, which corresponds to a call to sink. After examining the attribute h, the user determines that this node is valid.

At this moment the debugger points to the method getMax as buggy, ending the debugging session. The user must then check the method and correct the error.

As we have seen, some of the questions that occur during a debugging session can be very complex. Notice however that the same questions will occur implicitly during a debugging session using a normal trace debugger. Moreover, the use of the declarative debugger facilitates the debugging process by allowing the user to compare the input and output values of each attribute and parameter modified in a method call. The directed navigation also helps by reducing the number of questions to those nodes with a non-valid node (5 questions out of 23 nodes in the example).

Two additional features of the tool can be used to further reduce the number of nodes that the user needs to check before finding the erroneous method:

- Before the debugging process the user can exclude some packages and classes which can be trusted. This reduces the size of the tree and therefore the number of questions.
- At any moment during the navigation the user can mark the method associated to any node as *trusted* which means that all the associated method calls are automatically valid.

In spite of this features, the number of questions performed by the debugger can still be large. The next section presents a proposal that can be very helpful in this sense.

3 Reducing the Number of Questions

Without any further improvements our declarative debugger will ask a lot of questions such that debugging is still tedious and time consuming. An obvious improvement is that we make sure that no question is asked several times. Unfortunately, it rarely occurs in practical applications that exactly the same questions would be asked. Thus, we are interested in a generalization of this idea. We could try to avoid *equivalent* questions. This leaves us with the need to find an appropriate notion of equivalence which ensures that answers to equivalent questions will be the same, at least with high probability.

When looking for an appropriate notion of equivalence, we came across the approaches for glass-box testing [12]. Here the idea is to generate a system of *test cases* (i.e. pairs of inputs and corresponding outputs of a component which is being tested) which in some sense covers the possible control and/or data flows of that component. Note that it is usually impossible to test all possible control and/or data flows, since there are too many and often infinitely many of them. Thus, one is usually content with some reasonable coverage. Typical goals are the coverage of all nodes and edges of the control-flow graph (see Fig. 5 for an example) or the coverage of all so-called *def-use chains* [1].

A def-use chain is a pair of a statement, where the value of some variable is computed, and a statement, where this value is used. Moreover, the value of the variable must not be changed between the definition and the use. In our example in Fig. 1, the assignment in line 06 and the assignment in line 13 constitute a def-use chain for variable x, denoted by (x,06,13) for short. Other examples of def-use chains are (j,11,08) and (j,11,11). These example demonstrate that a definition needs not be textually above a use, in particular in the presence of loops. On the other hand, (j,05,12) is not a def-use chain, since the value of j is modified on every path from 05 to 12, in this case in line 11.

Testers assume that a component is correct, if all test cases pass without indicating an error. Each test case is a representative of a class of equivalent test cases causing an equivalent behavior, i.e. they produce the same coverage. We would like to pick up this notion of equivalence. If a question concerning the soundness of a method call $\mathsf{m}(\mathsf{a}_1,\ldots,\mathsf{a}_n)$ shall be asked by the debugger and if there was a previous

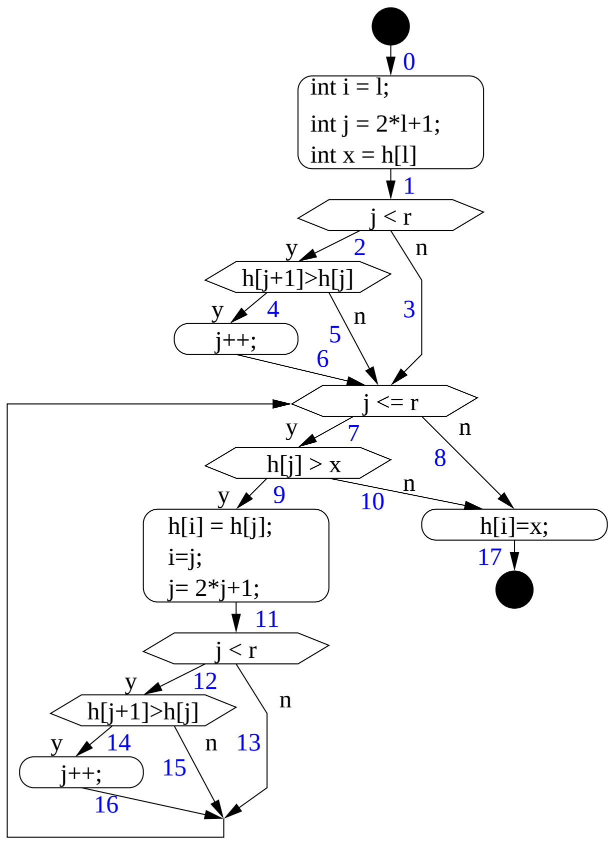


Fig. 5. Control-flow graph of method sink in the Heapsort example. The blue edge numbers correspond to those in Figure 7 b).

question corresponding to some method call $\mathsf{m}(\mathsf{b}_1,\ldots,\mathsf{b}_n)$, where $\mathsf{m}(\mathsf{a}_1,\ldots,\mathsf{a}_n)$ and $\mathsf{m}(\mathsf{b}_1,\ldots,\mathsf{b}_n)$ are equivalent w.r.t. to the corresponding coverage of the control or data flow, the second question will not be asked but the result of the first question will be re-used.

a)

```
01 public static int percent(int x, int y){
02     result = x/y;     // should be: x/y*100;
03     return result;
04 }
```

b)

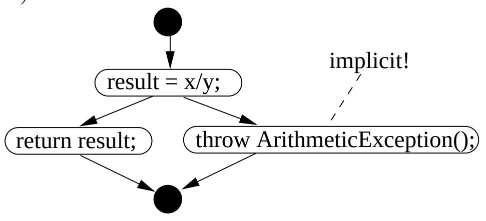


Fig. 6. Erroneous method percent and corresponding control-flow graph.

Unfortunately, the mentioned coverage criteria cannot guarantee the absence of errors. It is well-known that a program may still contain errors, although all test cases constructed according to the criterion have passed. For instance, the erroneous method percent in Figure 6 contains the three def-use chains (x,01,02),(y,01,02), and (result,02,03). All of them are covered by the test case with input parameters x=0 and y=1 and expected output 0 without exposing the error. If we add a test case with input parameters x=0 and y=0 and with an ArithmeticException as expected output, then also all edges (and nodes) of the control-flow graph (see Figure 6 b)) are covered, again without exposing the error. This would require another test case, e.g. with input x=1 and y=1 and expected output 100.

Since most but not all errors can be detected based on a coverage criterion, no matter which of them we select, we will allow the user to switch off the coverage-based inference of answers. The general approach will work as follows. In the beginning the user enables the elimination of equivalent questions. Just as the coverage criteria allow to find most errors in a software component, this configuration will allow the user to perform most of the debugging quickly and easily by answering as few questions as possible. As soon as no further errors can be found this way, the debugger is switched into a mode where it does ask equivalent questions w.r.t. the coverage criterion. Only identical questions and questions, where the answer can surely be inferred from previous answers, will be eliminated.

Now it remains to find a way to check method calls for equivalence. We intend to do this based on the test-case generator GlassTT [8,9,10]. This tool automatically generates a system of test cases which guarantees a selected coverage criterion to be met. Each test case generated corresponds to a solution of a system of constraints which describes the respective equivalent class.

GlassTT is based on a symbolic Java virtual machine (SJVM) and a system of constraint solvers. The SJVM executes the Java byte code symbolically. The input parameters are understood as a kind of logic variables whose values are not yet known and which will be described by constraints which appear during the symbolic computation when a branching instruction is encountered. In this case the SJVM will check with the help of a system of constraint solvers which alternatives remain valid. These alternatives will be tried one by one using a backtracking mechanism.

The SJVM contains in addition to the heap and frame stack of the usual Java virtual machine some components which are known from abstract machines for logic programming languages such as the Warren Abstract Machine (WAM) for Prolog [2,17]. In particular, it provides a choice point stack and a trail. These components enable the mentioned backtracking.

One way for checking the equivalence of method calls is to check whether they correspond both to solutions of the same set of constraints. Another way would be just to compare the sets of covered def-use chains or edges and nodes of the control-flow graph. We could even go one step further and combine the information gathered from several previous answers given by the user. We could compute the union of the corresponding coverage sets and check whether the coverage caused by the considered method call is subsumed by this union. We need more experience in order to tell whether this generalization is helpful in practice or not.

Figure 7 a) shows the hierarchy of method calls in our running example. As pointed out already the tester has indicated that buildHeap() worked properly. Since buildHeap() causes several calls to sink, they have also worked correctly. Since these calls cover all def-use chains covered by the call sink(0,6), the corresponding question in the body of getMax is redundant and the result can be inferred from the previously collected information. In fact, the sets of def-use chains for sink(0,6) and sink(0,7) are the same. Thus, already the (implicit) answer to the question, whether sink(0,7) works properly, can be used to infer that sink(0,6) works properly, too. If we use edge coverage in the control-flow graph rather the def-use chain coverage, we also

```
a) sort([89,5,93,12,29,4,42,17])
       buildHeap()
          sink(3,7)
          sink(2,7)
          sink(1,7) (+)
          sink(0,7)
       getMax(7)
          sink(0,6) (*,+)

       getMax(6)
          sink(0,5) (+)
       getMax(5)
          sink(0,4)
       getMax(4)
          sink(0,3) (*,+)
       getMax(3)
          sink(0,2) (*,+)
       getMax(2)
          sink(0,1) (*,+)
       getMax(1)
          sink(0,0) (*,+)
```

```
b) sink(3,7):
     D ={(l,03,04),(l,03,05),(l,03,06),
         (r,03,07),(r,03,08),(r,03,12),
         (h,03,06),(h,03,08),(h,03,9.2),
         (i,04,09),(i,10,13),(x,06,08),(x,06,13)
         (j,05,07),(j,05,08),(j,05,08.2),(j,05,09),(j,05,10),
         (j,05,11),(j,11,12),(j,11,12),(j,11,08)}
     E = {0,1,2,3,5,7,8,9,11,13,17}

   sink(2,7):
     D ={(l,03,04),(l,03,05),(l,03,06),
         (r,03,07),(r,03,08),(h,03,06),(h,03,07),(h,03,07.2),(h,03,08),
         (i,04,13),(x,06,08),(x,06,13),
         (j,05,07),(j,05,07.2),(j,05,07.3),(j,05,07.4),(j,7.4,08),(j,7.4,08.2)}
     E = {0,1,2,4,6,7,10,14,17}

   sink(1,7):
     D ={(l,03,04),(l,03,05),(l,03,06),
         (r,03,07),(r,03,08),(r,03,12),
         (h,03,06),(h,03,07),(h,03,07.2),(h,03,08),(h,03,9.2)
         (i,04,09),(i,10,13),(x,06,08),(x,06,13),
         (j,05,07),(j,05,07.2),(j,05,07.3),(j,05,07.4),(j,7.4,08),
         (j,7.4,08.2),(j,7.4,09),(j,7.4,10),(j,7.4,11),(j,11,12),(j,11,08)}
     E = {0,1,2,4,6,7,8,9,11,13,17}

   sink(0,7) and sink(0,6):
     D ={(l,03,04),(l,03,05),(l,03,06),
         (r,03,07),(r,03,08),(r,03,12),(h,03,06),(h,03,07),
         (h,03,07.2),(h,03,08),(h,03,9.2),(h,09,12),(h,09,12.2),
         (i,04,09),(i,10,13),(x,06,08),(x,06,13),
         (j,05,07),(j,05,07.2),(j,05,07.3),(j,05,07.4),(j,7.4,08),
         (j,7.4,08.2),(j,7.4,09),(j,7.4,10),(j,7.4,11.2),(j,11,12),
         (j,11,12.2),(j,11,12.3),(j,11,12.4),(j,12.4,08),(j,12.4,08.2)}
     E = {0,1,2,4,6,7,9,10,11,12,14,15,16,17}
```

Fig. 7. a) Hierarchy of method calls in the Heapsort example. For calls marked with (*) the answer can be inferred from the previous answers based on def-use chain coverage. For calls marked with (+) the answer can be inferred from the previous answers based on edge coverage of the control-flow graph in Figure 5. b) Sets D of def-use chains and sets E of edges in the control-flow graph (see Figure 5) covered by calls to sink. If there are several occurrences of a considered variable in some line XX, XX.j refers to the j-th occurrence of that variable ($j = 2, \ldots$).

observe that the sets of edges covered by sink(0,6) and sink(0,7) are the same (see Figure 7 b)). Thus, for both coverage criteria the debugger can directly conclude in our example that a question corresponding to the call sink(0,6) can be omitted and the error is located within the body of getMax.

If we would have been interested in also investigating the rest of the computation tree, all the questions corresponding to calls marked with (*) would also have been omitted, since there answers could also have been inferred based on def-use chain coverage. With edge coverage of the control-flow graph questions related to the calls marked with (+) could have been omitted. Figure 7 b) shows all the def-use chains and edges covered by the calls to sink.[5]

Let $D_1,\ldots,\ D_n$ be the sets of def-use chains corresponding to the previous n calls of the considered method, and let D be the set of def-use chains covered by the considered call to that method. Analogously, let $E_1,\ldots,\ E_n$ be the sets of edges of the control-flow graph covered by the previous n calls of the considered method, and let E be the set of edges covered by the considered call to that method. These pieces of information give us a couple of options, how to combine them. Depending on how conservative we want to be, we can infer that a considered call works properly, if

(i) $\exists\ i : 1 \leq i \leq n\ \wedge\ D \subset D_i$

[5] Note that we have here attributed definitions and uses to the whole array h. This could be refined by attributing them to individual elements.

(ii) $\exists\ i : 1 \leq i \leq n\ \wedge\ E \subset E_i$

(iii) $\exists\ i : 1 \leq i \leq n\ \wedge\ (D \subset D_i\ \ \vee\ \ E \subset E_i)$

(iv) $\exists\ i : 1 \leq i \leq n\ \wedge\ D \subset D_i\ \ \wedge\ \ E \subset E_i$

(v) $D \subset \bigcup_{i=1}^{n} D_i$

(vi) $E \subset \bigcup_{i=1}^{n} E_i$

(vii) $D \subset \bigcup_{i=1}^{n} D_i\ \ \vee\ \ E \subset \bigcup_{i=1}^{n} E_i$

(viii) $D \subset \bigcup_{i=1}^{n} D_i\ \ \wedge\ \ E \subset \bigcup_{i=1}^{n} E_i$

(ix) $\ldots$

The tester can select one of these strategies by modifying a configuration option of the tool. Moreover, it is possible to change this option while testing. Thus, it is recommendable to start with a “generous” strategy (e.g. (vii)) in order to eliminate most errors quickly and easily and then switch to a more conservative one (e.g. (iv)), as soon as no more errors can be found with the selected option. At the end, all these options will be switched off and only identical questions and questions related to trusted methods will be omitted.

In our running example, all eight strategies would allow us to omit the question related to the call sink(0,6).

When combining negative information (i.e. the call was not working properly), we have to take $\supset$ instead of $\subset$ in (i) - (iv). The other strategies make no sense in this case.

4 Limitations

Our present prototype does not yet support threads. We don’t think that it would be difficult to support them, we just have not done it. In contrast to a conventional debugger, the declarative debugger would even have the advantage that the computation tree records one fixed run of the program and this run can be investigated as long and as often as the user likes. We don’t have the problem to repeat an error when debugging. Thus, we get techniques such as instant replay [15] for free.

Another limitation of our prototype is that it only indicates a wrong method rather than the buggy statement in that method. We could switch to a trace debugging mode for finding the error more precisely. However, a well-designed program should not contain long methods. Thus, it should not be a problem to find the erroneous line in the indicated method.

Moreover, there are some difficulties when debugging event-driven computations. Methods such as actionPerformed are not called directly from main or some other own method but implicitly from the event-processing mechanism. We can only record and process the computation tree below the call to such a call-back method.

Surprisingly, our prototype is useful for debugging non-terminating computations, too. Here, the user must abort the program and the debugger will show the partially constructed tree. Some of the nodes will not contain a result, but others will and they can be used for debugging. Something similar happens, if the program ends with a non-captured exception.

Another problem is the debugging of very large programs or programs working with large data structures, which will require a lot of computer memory to keep the whole computation tree. In the future we plan to overcome this difficulty by storing the computation tree in a database. Since only part of the tree is displayed at each moment on the screen this will not affect the navigation phase. Notice however that from the point of view of the effort required from the user to find a bug our declarative debugger is not worse than a trace debugger. On the contrary, the higher-level of abstraction can dramatically reduce the amount of material the user has to investigate, especially with the improvements discussed in the last section. Moreover, we strongly suggest to take the test cases produced by GlassTT also for debugging. They have not only the advantage that they cover the code systematically, but they are also very small w.r.t. the data structures used as parameters. Among all equivalent test cases leading to the same coverage, GlassTT tries to generate that test case with the smallest possible data structures as parameters. Thus, the generated test cases are the smallest ones that enable to detect an error caused by a particular coverage. Experiments show that usually very small data structures with very few elements (mostly less than five) are sufficient. Thus, there is typically little need to handle arrays of thousands of elements when debugging.

5 Conclusions and Future Work

We have shown that the concept of declarative debugging known from declarative programming can be applied to imperative and object oriented programming, too. In particular, we have developed and presented a tool which enables the declarative debugging of Java programs. A major difference to the situation found in declarative languages is that Java enables side effects. Thus, it is not sufficient to show just the parameters and result of some method call to the tester. It is also necessary to show the attributes and possibly also local variables. This requires a well-designed user interface, which does not overstrain the tester with too much information, but allows to expand the required pieces of the state space on demand and to fold them again later on. It also has to allow the demand driven navigation of the object graphs which are accessible from the variables and attributes

The major advantage of declarative debugging compared to conventional debugging is that it works on a higher level of abstraction. Rather than inspecting the state space after each instruction starting from some break point, the tester is asked a sequence of questions related to method calls. This gives us semantical information, which can be used to avoid identical or equivalent questions and hence to reduce the search space enormously.

A particular novelty of our approach is the idea of using classical code-coverage criteria such as def-use chain coverage or coverage of the edges and nodes of the control-flow graph as basis for defining the equivalence of method calls and a means for reducing the amount of questions. In order to check the equivalence of method calls we have resorted to our test-case generator GlassTT, which automatically generates a system of test-cases guaranteeing a selected coverage criterion.

In a couple of experiments, our declarative debugger has behaved quite nicely and we could find errors rather quickly.

As future work we intend to supply empirical evidence for the advantage of our approach. We plan to let groups of students search for errors using conventional and declarative debugging and to investigate the differences based on some example applications of different sizes.

Acknowledgement

We thank Francisco González-Blanch Rodríguez, Reyes de Miguel Roses and Susana Serrano Soria for their work in the implementation of the prototype.

References

[1] A.V. Aho, R. Sethi, J.D. Ullman: Compilers: Principles, Techniques and Tools. Addison Wesley, 186.

[2] H. Aït-Kaci: Warren's Abstract Machine: A Tutorial Reconstruction, MIT Press, 1991.

[3] R. Caballero and M. Rodríguez-Artalejo. A Declarative Debugging System for Lazy Functional Logic Programs. Electronic Notes in Theoretical Computer Science, 64, 2002.

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest: Introduction to Algorithms, section 7, MIT Press, 1990.

[5] P. Fritzson, N. Shahmehri, M. Kamkar and T. Gyimothy. *Generalized algorithmic debugging and testing* ACM Letters on Programming Languages and Systems (LOPLAS) archive Volume 1 , Issue 4 (December 1992), pp. 303 - 322, 1992.

[6] H. Z. Girgis and B. Jayaraman: JavaDD: a Declarative Debugger for Java. Tech. Report 2006-7, Department of Computer Science and Engineering, University at Buffalo, 2006.

[7] JPDA: http://java.sun.com/j2se/1.5.0/docs/guide/jpda.

[8] C. Lembeck, R. Caballero, R. Müller, H. Kuchen: Constraint Solving for Generating Glass-Box Test Cases, Proceedings of International Workshop on Functional and (Constraint) Logic Programming (WFLP), 19-32, Aachen, 2004.

[9] R. Müller, C. Lembeck, H. Kuchen: A Symbolic Java Virtual Machine for Test-Case Generation, Proceedings IASTED, 2004.

[10] R. Müller, C. Lembeck, H. Kuchen: GlassTT - a Symbolic Java Virtual Machine Using Constraint Solving Techniques for Glass-Box Test Case Generation, Technical Report 102, University of Münster, Department of Information Systems, 2003.

[11] H. Nilsson: How to look busy while being lazy as ever: The implementation of a lazy functional debugger. Journal of Functional Programming 11(6), pages 629–671, 2001.

[12] R.S. Pressman: Software Engineering – A Practitioner's Approach, McGraw-Hill, 1982.

[13] N. Shahmehri and P. Fritzson. *Algorithmic debugging for imperative programs with side-effects*. Res. Rep. LiTH-IDA-R-89-49. Dept. of Computer and Information Science, Linköping Univ. Sweden. 1989.

[14] E.Y. Shapiro: Algorithmic Program Debugging. The MIT Press, 1982.

[15] K.Shen, S. Gregory: Instant Replay debugging of concurrent logic programs, New Generation Computing,volume 14 (1): 79–107, 1996.

[16] Sun Microsystems: Java 2 Platform, 2006. http://java.sun.com/javase/.

[17] D.H.D. Warren: An Abstract Prolog Instruction Set, Technical Note 309, SRI International, Menlo Park, CA, October 1983.