Available online at www.sciencedirect.com
SciVerse ScienceDirect
Electronic Notes in Theoretical Computer Science 282 (2012) 47–60
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Implementation of an Optimal Strategy for Algorithmic Debugging[1]

David Insa[2] Josep Silva[3]

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València
E-46022 Valencia, Spain.

Abstract

One of the most automatic debugging techniques is Algorithmic Debugging because it allows us to debug a program without the need to inspect the source code. In order to find a bug, an algorithmic debugger asks questions to the programmer about the correctness of subcomputations in an execution. Reducing the number and complexity of these questions is an old objective in this field. Recently, an strategy for algorithmic debuggers that minimizes the number of questions has been released. This new strategy is called *Optimal Divide and Query* and, provided that all questions can be answered, it finds any bug in the source code with a minimum set of questions. In this work we discuss the implementation of such a strategy in different algorithmic debugging architectures.

Keywords: Debugging, Algorithmic Debugging.

1 Introduction

Debugging is often a hard task. Specially when we try to debug source code that we have not written ourselves or that we wrote a long time ago. When this happens, we usually know or remember what our functions or methods do, but we hardly remember how they do it. In these cases, the technique *Algorithmic Debugging* [10,11] can be very useful because it allows us to debug programs

[1] This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación (Secretaría de Estado de Investigación)* under grant TIN2008-06622-C03-02 and by the *Generalitat Valenciana* under grant PROMETEO/2011/052.
[2] Email: dinsa@dsic.upv.es
[3] Email: jsilva@dsic.upv.es

doi:10.1016/j.entcs.2011.12.005

without the need to inspect the source code. The technique automatically generates a series of questions and uses the answers of the programmer to find the location of a bug. The questions are always whether a given result of a method or function activation with given input values is actually correct. Therefore, the programmer only needs to know what a function is supposed to do (instead of how) in order to debug it.

Example 1.1 Consider this simple Haskell program that wrongly (it has a bug) checks whether two sets are equal:

```
main = equal_sets [1,3] [3]

element _ []     = False
element x (y:ys) = (x==y) || element x ys

subset [] _      = True
subset (x:xs) ys = element x ys || subset xs ys

equal_sets xs ys = subset xs ys && subset ys xs
```

An algorithmic debugging session for this program is the following (`YES` and `NO` answers are provided by the programmer):

```
Starting Debugging Session...
(1)  subset [1,3] [3] = True? NO
(2)  element 1 [3] = False? YES
(3)  subset [3] [3] = True? YES

Bug found in rule:
subset (x:xs) ys = element x ys || subset xs ys
```

The debugger points out the part of the code that contains the bug. In this case `||` should be `&&`. Note that, to debug the program, the programmer only has to answer questions. It is not even necessary to see the code.

Typically, algorithmic debuggers have a front-end that produces a data structure representing a program execution—the so-called *execution tree* (ET) [8]—; and a back-end that uses the ET to ask questions and process the answers of the programmer to locate the bug. For instance, the ET of the program in Example 1.1 is depicted in Figure 1.

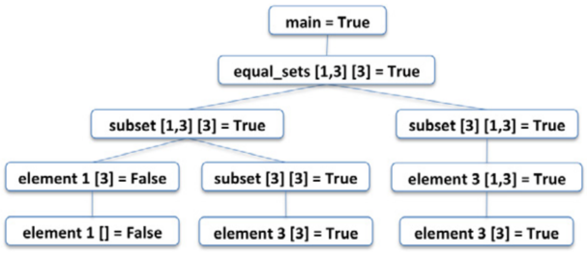


Fig. 1. ET of the program in Example 1.1

The internal algorithm used by algorithmic debuggers to decide what nodes of the ET should be asked is crucial for the performance of the technique. In [6], we conducted a series of experiments to compare the performance of different algorithms, and *Divide & Query* (D&Q) and its variants [11] showed the best performance. In that work, we also proved that the variant *Optimal D&Q* asks (as an average) an optimal number of questions.

In this paper we present an implementation of Optimal D&Q that has been integrated into the Declarative Debugger for Java [5]. We show how to implement this algorithm in different implementation contexts that present different architectures.

The rest of the paper has been organized as follows. In Section 2 we recall and formalize the strategy D&Q and its improved version Optimal D&Q. Then, in Section 3 we discuss the implementation of Optimal D&Q. Finally, Section 4 concludes.

2 Algorithmic Debugging and Optimal D&Q

In this section we formalize the strategy D&Q [10] focussing on the improved version by Hirunkitti [4]. We start with the definition of *marked execution tree*, that is an ET where some nodes could have been removed because they were marked as correct (i.e., answered YES), some nodes could have been marked as wrong (i.e., answered NO) and the correctness of the other nodes is undefined.

Definition 2.1 [Marked Execution Tree] A marked execution tree (MET) is a tree $T = (N, E, M)$ where N are the nodes, $E \subseteq N \times N$ are the edges, and $M : N \to V$ is a marking total function that assigns to all the nodes in N a value in the domain $V = \{\mathit{Wrong}, \mathit{Undefined}\}$.

Initially, all nodes in the MET are marked as *Undefined*. But with every answer of the user, a new MET is produced. Concretely, given a MET $T = (N, E, M)$ and a node $n \in N$, the answer of the user to the question in n produces a new MET such that: (i) if the answer is YES, then this node and its subtree is removed from the MET. (ii) If the answer is NO, then, all the nodes in the MET are removed except this node and its descendants. [4]

Therefore, the size of the MET is gradually reduced with the answers. If we delete all nodes in the MET then the debugger concludes that no bug has been found. If, contrarily, we finish with a MET composed of a single

[4] It is also possible to accept *I don't know* as an answer of the user. In this case, the debugger simply selects another node [2]. For simplicity, we assume here that the user only answers *YES* or *NO*.

node marked as wrong, this node is called *buggy node* and it is pointed as responsible of the bug of the program.

All this process is defined in Algorithm 1 where function *selectNode* selects a node in the MET to be asked to the user with function *askNode*. Therefore, *selectNode* is the central point of this paper. In the rest of this section, we assume that *selectNode* implements D&Q. In the following we use E^* to refer to the reflexive and transitive closure of E.

Algorithm 1 General algorithm for algorithmic debugging

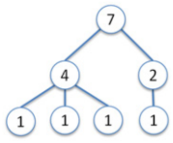
```
Input: A MET $T = (N, E, M)$
Output: A buggy node or $\perp$ if no buggy node exists
Preconditions: $\forall n \in N$, $M(n) = Undefined$
Initialization: buggyNode = $\perp$

begin
(1)  do
(2)      node = selectNode($T$)
(3)      answer = askNode(node)
(4)      if (answer = $Wrong$)
(5)      then $M$(node) = $Wrong$
(6)              buggyNode = node
(7)              $N = \{n \in N \mid (\text{node} \to n) \in E^*\}$
(8)      else $N = N \backslash \{n \in N \mid (\text{node} \to n) \in E^*\}$
(9)  while ($\exists n \in N, M(n) = Undefined$)
(10) return buggyNode
end
```

D&Q assumes that the individual weight of a node is always 1. Therefore, given a MET $T = (N, E, M)$, the weight of the subtree rooted at node $n \in N$, w_n, is defined as its number of descendants including itself (i.e., $1 + \sum \{w_{n'} \mid (n \to n') \in E\}$).

D&Q tries to simulate a dichotomic search by selecting the node that better divides the MET into two subMETs with a weight as similar as possible. Therefore, given a MET with n nodes, D&Q searches for the node whose weight is closer to $\frac{n}{2}$. In particular, it always selects the node whose weight is closer to $\frac{n}{2}$ between:

- the heaviest node n' whose weight is as close as possible to $\frac{n}{2}$ with $w_{n'} \leq \frac{n}{2}$,
- the lightest node n' whose weight is as close as possible to $\frac{n}{2}$ with $w_{n'} \geq \frac{n}{2}$

Example 2.2 Consider the following MET where nodes are labeled with their weight.



In this MET, D&Q divides the MET by selecting the node with weight 4 because it is the closer one to $\frac{7}{2}$.

2.1 Optimal Divide & Query

In [6], Optimal Divide & Query was introduced as a new variant of D&Q that optimally divides the remaining tree with every question. It is presented in Algorithm 2 where w_n represents the weight of node n (i.e., the weight of the subtree rooted at n), and wi_n represents the individual weight of node n (i.e., the weight of the single node n without taking into account its descendants). It is important to note that, in this algorithm, the weight of a subtree with root n is computed with the sum of the individual weights of all nodes in the subtree, but the individual weight of n is only added if it is marked as Undefined. In the case that it is marked as Wrong, then it is ignored.

Algorithm 2 Optimal D&Q (SelectNode)

```
Input: A MET $T = (N, E, M)$ whose root is $n \in N$,
       $\forall n_1, n_2 \in N, wi_{n_1} = wi_{n_2}$ and $\forall n_1 \in N, wi_{n_1} > 0$
Output: A node $n' \in N$

begin
(1)  Candidate = $n$
(2)  do
(3)     Best = Candidate
(4)     Children = $\{m \mid (\text{Best} \rightarrow m) \in E\}$
(5)     if (Children = $\emptyset$) then return Best
(6)     Candidate = $n' \in$ Children | $\forall n'' \in$ Children, $w_{n'} \geq w_{n''}$
(7)  while ($w_{Candidate} > \frac{w_n}{2}$)
(8)  if ($M$(Best) = $Wrong$) then return Candidate
(9)  if ($w_n \geq w_{Best} + w_{Candidate} - wi_n$)
(10)  then return Best
(11)  else return Candidate
end
```

Essentially, Algorithm 2 traverses the MET top-down from the root until it finds the buggy node. In order to do this, it compares nodes to discard some of them and define a path until the buggy node. It is based on four properties that are summarized in Figure 2: In cases 1 and 4, the heaviest node is better. In case 2, the lightest node is better. And in case 3, the best node must be determined with the equation $w_{root} \geq w_{n_1} + w_{n_2} - wi_{root}$ that is implemented in Line (9) of the algorithm. Observe that these cases allow the algorithm to determine the path to the optimal node that is closer to the root by comparing a reduced number of nodes.

3 Implementation of Optimal Divide and Query

In this section we present our implementation of the strategy Optimal D&Q and we discuss how can it be adapted to different architectural contexts. Our

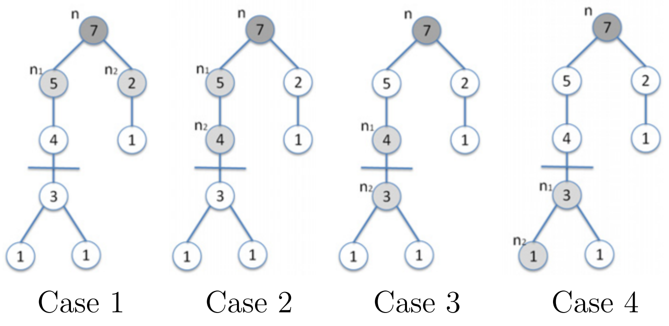


Fig. 2. Determining the best node in a MET (four possible cases)

algorithmic debugger, the Declarative Debugger for Java (DDJ) [5], debugs Java programs and it has been also written in Java itself. Consequently, in this debugger the information of the nodes contain method invocations and their effects.

Along this section, we will assume the existence of an object that implements the strategy Optimal D&Q. We will refer to this object with the usual reference *this*, and thus, we can access the methods of this object as usual (e.g., $this.node.getState()$, $this.moveNodeToChild(indexChild)$, etc.).

The implementation presented includes the methods that compose the *OptimalDivideQuery* class. Some of these methods must be adapted depending on the architecture of the host debugger. In particular, we discuss these methods for three different architectures:

(i) *Standard architecture.* Where nodes are pruned after every answer according to Algorithm 1; and where all the nodes of the MET are already generated when the first question is asked.

(ii) *Fix MET architecture.* Where the MET is never pruned; and thus nodes have a state attribute that indicates to the strategy whether this node can be buggy or not. Here again, it is assumed that all the nodes of the MET are already generated when the first question is asked.

(iii) *Dynamic MET architecture.* Where nodes are pruned after every answer according to Algorithm 1; and where the nodes in the MET can be dynamically generated while the questions are asked.

The code of the class *OptimalDivideQuery* that is common for the three architectures is shown in Frame 1. The code that is only used in the standard architecture is shown in Frame 2 (it continues the code in Frame 1). The code that is only used in the fix MET architecture is shown in Frame 3 (it continues the code in Frame 1). The code that is only used in the dynamic MET architecture is shown in Frame 4 (it continues the code in Frame 1).

Clearly, the strategy needs a mechanism to explore the MET and extract

```
private Node node;

public Node selectNode()
begin
this.node = this.root;
double weight = this.calculateWeight();
double weightFather = weight;
boolean thru = true;

mainLoop:
do {
   int numChildren = this.getNodeChildren();

   // Heaviest child
   int indexChild = -1;
   double weightChild = -1;
   for (int index = 0; index < numChildren; index++) {
      this.moveNodeToChild(index);
      double weightCandidate = this.node.getWeight();
      if (this.node.getState() == Undefined && weightCandidate > weightChild)
      {
         weightChild = weightCandidate;
         indexChild = index;
      }
      this.moveNodeToFather();
   }

   // Leaf
   if (indexChild == -1)
      return this.node;

   // Continue going down or equation
   thru = weightChild > weight / 2;
   if (thru || this.node.getState() == Wrong ||
   !this.equation(weight, weightFather, weightChild, this.root.getIndividualWeight()))
   {
      this.moveNodeToChild(indexChild);
      weightFather = weightChild;
   }
} while (thru);

return this.node;
end

private boolean equation(double rootWeight, double weightNode1, double weightNode2,
double individualWeight)
begin
if (weightNode1 == weightNode2)
   return true;
if (weightNode1 > weightNode2)
   return rootWeight ≥ weightNode1 + weightNode2 - individualWeight;
return rootWeight < weightNode1 + weightNode2 - individualWeight;
end
```

Frame 1: class *OptimalDivideQuery* (code independent of the architecture)

the information from nodes. For this, we use a pointer that can point to any node in the MET. This pointer is the attribute *node* of the strategy that initially points to the root of the MET and can be moved to any node by means of the method *selectNode*. This method (*selectNode*) implements the Optimal D&Q strategy presented in Algorithm 2, and it uses 7 different methods during its execution:

- *node.getState()* returns the state of the current node:
  - · *Undefined* if no information is known about this node,
  - · *Wrong* if this node has been marked as wrong,
  - · *Right* if this node has been marked as correct,
  - · *Trusted* if the method associated to this node is trusted [7] and cannot contain the bug, and
  - · *Unknown* if this node has been marked as unknown, e.g., because its associated question was too difficult.
- *node.getIndividualWeight()* returns the probability that the node contains the bug,
- *node.getWeight()* returns the probability that the subtree of the node contains the bug,
- *this.getNodeChildren()* returns the number of children from the node attribute,
- *this.moveNodeToChild(index)* updates the node attribute in order to point to one of its children,
- *this.moveNodeToFather()* updates the node attribute in order to point to its father,
- *this.calculateWeight()* calculates the weight of the node attribute taking its state into account.

Only four of these methods depend on the architecture selected from the three discussed above. Therefore, these methods are implemented at the level of the strategy (i.e., *this.getNodeChildren()*, *this.moveNodeToChild(index)*, *this.moveNodeToFather()* and *this.calculateWeight()*) and not at the level of node as *node.getState()*, *node.getWeight()* and *node.getIndividualWeight()*.

The code of *selectNode()* is divided into three parts: Lines (1) to (4) initialize the main loop. Here, variable *weight* represents the weight of the root, variable *weightFather* represents the weight of the best node found so far and it is initialized with the weight of the root node, and variable *thru* is used to decide when the loop must terminate; Lines (6) to (34) implement a loop that finds the optimal node; and finally Line (36) returns the optimal node.

The loop traverses the MET in a top-down manner in order to find the optimal node. We start the search in the root node and in each iteration we descend to one of its children. Once we have selected one child, the other children are discarded. Lines (8) to (22) determine this child and stores it in variable *indexChild*. If this child does not exist (e.g., the node is a leaf) then Lines (24) to (26) return the node itself as the optimal node.

After selecting the child, Line (29) checks whether the optimal node is in the subtree of the child (otherwise it must be in the child itself or in its father). If the optimal node is in the subtree then variable *thru* remains true, and another iteration is performed to look for the buggy node in this subtree. Otherwise, variable *thru* becomes false and method *equation* is used to determine the optimal node. If it returns true, attribute *node* remains pointing to the father, otherwise it is updated to point to the child. Finally, the node pointed by attribute *node* is returned as the optimal node.

Note that the behavior of the strategy should be the same when variable *thru* remains true and when the *equation* method returns false. In both cases attribute *node* should be updated to point to the child. Algorithm 2 directly finishes the loop at this point and then it checks which node is the optimal one. But in our code this procedure would repeat Lines (30) and (31) after the loop. This can be easily avoided extending the *if* condition, because during the execution of Line (31) the value of *thru* is not modified and then the loop will terminate.

Another important part of the code is implemented by method *equation* that implements Line (9) of Algorithm 2. It is used to decide what node is better between any pair of nodes. If it returns true the first node is better, if it returns false the second node is better.

The following subsections discuss specific changes in the algorithms that are used for each of the architectures discussed.

3.1 Standard Architecture

The specific code for this architecture is shown in Frame 2. In this setting the nodes that cannot be buggy are pruned from the MET. This means that the root node can be marked as *Undefined* or *Wrong*, and the others can only be marked as *Undefined*. This property strongly simplifies the technique and the implementation of the methods.

Traditionally, the weight of a node represents the number of nodes of the subtree. In our approach the weight of a node represents the probability that the subtree of this node contains the bug. This probability is computed with method *calculateWeight* shown in Frame 2. This method does not take into account the wrong root when computing probabilities because it is not necessary to answer it again to determine that it is wrong.

In this architecture, Lines (24) to (26) of Frame 1 could be moved to Line (9) using the condition *numChildren==0* instead of *indexChild==-1* and the condition *this.node.getState()==Undefined* in line (16) can be removed. These changes make the algorithm to avoid unnecessary checks.

```
private int getNodeChildren()
begin
(1)  return this.node.children.size();
end

private void moveNodeToChild(int index)
begin
(1)  this.node = this.node.getChild(index);
end

private void moveNodeToFather()
begin
(1)  this.node = this.node.getFather();
end

private double calculateWeight()
begin
(1)  double weight = this.node.getWeight();
(2)  if (this.node.getState() == Wrong)
(3)      weight -= this.node.getIndividualWeight();
(4)  return weight;
end
```

Frame 2: class OptimalDivideQuery (code for the standard architecture)

3.2 Fix MET Architecture

As stated before, in the standard architecture nodes are pruned from the MET. While this simplifies the technique, it also removes the possibility of reusing the same MET in other sessions (e.g., in order to find more than one bug). For this reason many debuggers avoid the pruning of the MET with some labeling mechanism that labels nodes as "possibly-buggy" or "no-buggy". In order to make this labeling mechanism compatible with the code in Frame 1, we assign a *weight* of 0 to Right and Trusted nodes, Unknown nodes are assigned an *individual weight* of 0, and Normal and Wrong nodes maintain their usual weights.

In an architecture where no nodes are pruned and different debugging sessions can use the same MET, a debugging session can end up with multiple Wrong nodes. This means that, in order to maintain the standard behavior, the last node marked as wrong should be the initial (root) node of the strategy (Line (1) of *selectNode* method in Frame 1). However, some debuggers [9,1,3] allow the user to make manual debugging sessions. In these cases the user can select a node whose subtree already has a wrong node. The algorithm of the standard architecture is not prepared for METs with multiple wrong nodes, but this situation can be easily handled adding Lines (15) to (19) in Frame 3 between Lines (14) and (15) of method *selectNode* in Frame 1. These lines make the algorithm to restart the search in the subtree whose root is already marked as wrong. In order to do this, we use the wrong node as the new root of the algorithm (this is done implicitly), we update both *weight* and *weightFather* to the weight of the new root, and we re-execute the external

```
private Node father = null;
private ArrayList<Node> children = new ArrayList<Node>();

public Node selectNode()
begin
(1)   ...
(15)        if (this.node.getState() == Wrong) {
(16)            weight = this.calculateWeight();
(17)            weightFather = weight;
(18)            continue mainLoop;
(19)        }
(20)  ...
end

private int getNodeChildren()
begin
(1) this.children.clear();
(2) this.father = this.node;
(3) int numChildren = this.getNodeChildren(0);
(4) this.node = this.father;
(5) return numChildren;
end

private int getNodeChildren(int numChildren)
begin
(1) for (Node child : this.node.children)
(2)     if (child.getState() == Undefined || child.getState() == Wrong)
(3)         this.children.add(numChildren++, child);
(4)     else if (child.getState() == Unknown) {
(5)         this.node = child;
(6)         numChildren = this.getNodeChildren(numChildren);
(7)     }
(8) return numChildren;
end

private void moveNodeToChild(int index)
begin
(1) this.node = this.children.get(index);
end

private void moveNodeToFather()
begin
(1) this.node = this.father;
end

private double calculateWeight()
begin
(1) double weight = this.node.getWeight();
(2) if (this.node.getState() == Wrong)
(3)     weight -= this.node.getIndividualWeight();
(4) return weight;
end
```

Frame 3: class OptimalDivideQuery (code for the fix MET architecture)

loop.

In addition, when the user marks a node as unknown, this node should be removed from the MET. But, because we do not prune these nodes, they remain in the middle of the tree. This means that the *getNodeChildren* method should be modified in order to exclude Unknown nodes. Moreover, it should also be modified to exclude Right and Trusted nodes that also remain in the

MET. This is performed by the new version of *getNodeChildren* shown in Frame 3, that stores in attribute *children* the children of the current node. If one of its children is an Unknown node, it is ignored, but the children of this Unknown node are also added. After collecting the children of the current node, we can use *moveNodeToChild* and *moveNodeToFather* methods in Frame 3 to move from father to children and vice versa. Finally, having correctly updated the weights of Right, Trusted and Unknown nodes, the *calculateWeight* method can be implemented as the one in Frame 2.

3.3 Dynamic MET Architecture

In this section we show the changes that we should make in order to adapt the algorithm to debuggers where the MET is dynamically generated while the debugging session is performed. These debuggers allow the user to start the debugging session while the MET is being produced and thus it is uncompleted. Therefore, the MET contains:

(i) *Completed nodes.* Those nodes that have been invoked and their execution already finished,

(ii) *Not completed nodes.* Those nodes that have been invoked but their execution did not finish, and

(iii) *Not generated nodes.* Those nodes that are not present yet in the MET because their invocation has not been performed yet.

An algorithmic debugger can only ask questions to completed nodes, that contain not only arguments and initial context but also final context and return value. Therefore, method *getNodeChildren(int numChildren)* should exclude those nodes that are not completed yet. The implementation of this method is similar to the one from Frame 3 changing Lines (2) and (4) (see Frame 4). Note that both approaches can be used together using a && operator in Line (2) and changing Line (4) by **else if** *(!node.isCompleted() || node.getState() == Unknown)* {. Methods *getNodeChildren()*, *moveNodeToChild* and *moveNodeToFather* can be implemented as in Frame 3.

In algorithmic debugging the weight of a node is computed with the sum of the weights of its children adding its own individual weight. In order to ensure that we calculate the weights of the nodes in linear time, they are calculated only when the node is completed. Note that when a node is completed, its children are also completed and thus no more nodes can be added. Therefore, in trees where nodes are not completed, the weights of some root descendants have not been calculated yet and, hence, the weight of the root node cannot be determined. Consequently, method *calculateWeight* should also be modified. The new version of this method in Frame 4 uses the *getNodeChildren*,

```
private Node father = null;
private ArrayList<Node> children = new ArrayList<Node>();

private int getNodeChildren()
begin
    this.children.clear();
    this.father = this.node;
    int numChildren = this.getNodeChildren(0);
    this.node = this.father;
    return numChildren;
end

private int getNodeChildren(int numChildren)
begin
    for (Node child : this.node.children)
        if (child.isCompleted())
            this.children.add(numChildren++, child);
        else {
            this.node = child;
            numChildren = this.getNodeChildren(numChildren);
        }
    return numChildren;
end

private void moveNodeToChild(int index)
begin
    this.node = this.children.get(index);
end

private void moveNodeToFather()
begin
    this.node = this.father;
end

private double calculateWeight()
begin
    double weight = 0;
    int numChildren = this.getNodeChildren();
    for (int index = 0; index < numChildren; index++) {
        this.moveNodeToChild(index);
        weight += this.node.getWeight();
        this.moveNodeToFather();
    }
    if (this.node.isCompleted() && this.node.getState() != Wrong)
        weight += this.node.getIndividualWeight();
    return weight;
end
```

Frame 4: class OptimalDivideQuery (code for the dynamic MET architecture)

moveNodeToChild and *moveNodeToFather* methods to transform all those completed nodes in the MET whose parent is not completed into children of the root node. In this way, when the strategy is used again, the weight of the root will be updated as new nodes become completed.

In this architecture, as in the standard architecture, Lines (24) to (26) of Frame 1 could also be moved to Line (9) using the condition *numChildren==0* instead of *indexChild==-1* and removing the condition *this.node.getState()==Undefined* in line (16). Here again, these changes make the algorithm to avoid unnecessary checks.

4 Conclusions

In this work we present an implementation of Optimal D&Q for object-oriented languages. We have shown our code for Java that has been integrated into a real debugger (DDJ) but the ideas discussed for the implementation could also be applied in other languages.

Our implementation has been presented in a parameterized way so that it can work in three different architectures, including those architectures that allow the dynamic generation of the MET. For each architecture, it has been discussed how to adapt the algorithms to the particular restrictions that they impose, and how to change the code to increase performance in this particular architecture.

References

[1] R. Caballero. A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In *Proc. of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05)*, pages 8–13, New York, USA, 2005. ACM Press.

[2] D. Cheda and J. Silva. State of the practice in algorithmic debugging. *Electron. Notes Theor. Comput. Sci.*, 246:55–70, August 2009.

[3] T. Davie and O. Chitil. Hat-delta: One Right Does Make a Wrong. In *Seventh Symposium on Trends in Functional Programming, TFP 06*, April 2006.

[4] V. Hirunkitti and C. J. Hogger. A Generalised Query Minimisation for Program Debugging. In *Proc. of International Workshop of Automated and Algorithmic Debugging (AADEBUG'93)*, pages 153–170. Springer LNCS 749, 1993.

[5] D. Insa and J. Silva. An Algorithmic Debugger for Java. *In Proc. of the 26th IEEE International Conference on Software Maintenance*, 0:1–6, 2010.

[6] D. Insa and J. Silva. Optimal Divide and Query (extended version). *Available in the Computing Research Repository (http://arxiv.org/abs/1107.0350)*, July 2011.

[7] Y. Luo and O. Chitil. Algorithmic debugging and trusted functions. Technical report 10-07, University of Kent, Computing Laboratory, UK, August 2007.

[8] H. Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Linköping, Sweden, May 1998.

[9] B. Pope. *A Declarative Debugger for Haskell.* PhD thesis, The University of Melbourne, Australia, 2006.

[10] E. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1982.

[11] J. Silva. A Comparative Study of Algorithmic Debugging Strategies. In *Proc. of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06)*, pages 143–159. Springer LNCS 4407, 2007.