Balancing Execution Trees*

D. Insa, J. Silva
Universitat Politècnica de València
Valencia, Spain
{dinsa,jsilva}@dsic.upv.es

A. Riesco
Universidad Complutense de Madrid
Madrid, Spain
ariesco@fdi.ucm.es

Abstract

Algorithmic debugging is a debugging technique that uses a data structure representing all computations of a program. This data structure is the so-called *Execution Tree* and it strongly influences on the performance of the technique. In this work we present a transformation that allows us to automatically balance execution trees by collapsing and projecting some strategic nodes. We prove that the transformation is sound in the sense that all the bugs found after the transformation are real bugs; and if at least one bug is detectable before the transformation, then at least one bug will be also detectable after the transformation. We have implemented the technique and performed several experiments with real applications. The experimental results confirm the usefulness of the technique.

1 Introduction

Algorithmic debugging [11] is a semi-automatic debugging technique which produces a dialogue between the debugger and the programmer to find the bugs. This technique relies on the programmer having an *intended interpretation* of the program. In other words, some computations of the program are correct and others are wrong with respect to the programmer's intended semantics. Therefore, algorithmic debuggers compare the results of subcomputations with what the programmer intended. By asking questions to the programmer or using a formal specification the system can identify precisely the location of a bug.

Essentially, algorithmic debugging is a two-phase process: During the first phase, a data structure that represents the execution of the program is built. This data structure is called *Execution Tree* (ET). The ET contains nodes that represent subcomputations of the program. Therefore, the information of the ET's nodes is different in each paradigm (e.g., functions, methods, procedures, etc.), but the construction of the ET is very similar in all of them. Without loss of generalization, in the following we will base our examples in the language Java. But our ET transformations and the balancing technique is applicable to any ET, independently of the language it represents. In the object-oriented paradigm, the ET is constructed as follows: Each node of the ET is associated with a method invocation, and it contains all the information needed to decide whether the method invocation produced a correct result. This information includes the call to the method with its parameters and the result, and the values of all the attributes that are in the scope of this method, before and after the invocation. This information allows the programmer to know whether all the effects of the method invocation correspond to her intended semantics. The root node of the ET is the *main* method of the program. For each node n with associated method m, and for each method invocation m' done

*This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant ACOMP/2009/017, and by the *Universidad Politécnica de Valencia* (Program PAID-06-08).

by m, a new node associated with m' is recursively added to the ET as a child of n.

Example 1 *Consider the Java program in Figure 1. This program wrongly simulates—it has a bug—some movements on a chessboard. The portion of the ET that is associated with the call* `p.castling(tower,king)` *is depicted in Figure 2.*

```
public class Chess {
  public static void main(String[] args) {
    Chess p = new Chess();
    Position tower = new Position();
    Position king = new Position();

    king.locate(5,1);
    tower.locate(8,1);
    p.castling(tower,king);
  }

  void castling(Position t,Position k) {
    if (t.x!=8)     {
      for(int i=1; i<=2; i++) {t.left();}
      for(int i=1; i<=2; i++) {k.right();}
    }
    else
    {
      for(int i=1; i<=3; i++) {t.right();}
      for(int i=1; i<=2; i++) {k.left();}
    }
  }
}

class Position {
    int x;
    int y;

    void locate(int a, int b) {x=a; y=b;}
    void up() {y=y+1;}
    void down() {y=y-1;}
    void right() {x=x+1;}
    void left() {x=x-1;}
 }
```

Figure 1: Example program

For the purpose of this work, we can consider ETs as labeled trees. We need to formally define the notions of context and method invocation before we provide a definition of ET.

Definition 1 (Context) *Let $\mathcal{P}$ be a program, and m a method in $\mathcal{P}$. Then, the* context *of m is $\{(a, v) \mid a$ is an attribute in the scope of m and v is the value of $a\}$.*

Roughly, the context of a method is composed of all the variables of the program that can be affected by the execution of this method. Clearly, these variables can be other objects that in turn contain other variables. In a realistic program, each node contains several data structures that could change during the invocation. All this information (before and after the invocation) should be visualized together with the call to the method so that the programmer can decide whether it is correct.

Definition 2 (Method Invocation) *Let $\mathcal{P}$ be a program and $\mathcal{E}$ an execution of $\mathcal{P}$. Then, each* method invocation *of $\mathcal{E}$ is represented with a triple $I = (b, m, a)$ where m is a string representing the call to the method with its parameters and the returned value, b is the context of the method in m before its execution and a is the context of the method in m after its execution.*

Definition 3 (Execution Tree) *Given a program $\mathcal{P}$ with a set of method definitions $M = \{m_i \mid 1 \leq i \leq n\}$, and a call c to $m \in M$, the* execution tree *(ET) of $\mathcal{P}$ w.r.t. c is a tree $t = (V, E)$ where the label of a node $v \in V$ is denoted with $l(v)$. $\forall v \in V, l(v)$ is a method invocation. And*

- *The root of the ET is the method invocation associated with c.*
- *For each node associated with a call c' to $m_j, 1 \leq j \leq n$, we have a child node associated with a call c'' to $m_k, 1 \leq k \leq n$, iff*
  1. *during the execution of c', c'' is invoked, and*
  2. *the call c'' is done from the definition of m_j.*

Once the ET is built, in the second phase, the debugger uses a strategy to traverse the ET asking the programmer to answer about the correctness of the information stored in each node. If the method invocation of a node is wrong according to the intended semantics, the answer is NO. Otherwise, the answer is YES. Using the answers, the debugger tries to find a buggy node (a buggy node is associated with the buggy source code of the program).

Definition 4 (Buggy node) *Given an ET $t = (V, E)$ being $r \in V$ the root of t, a buggy node of t is a node $v \in V$ such that:*

1. *The method invocation of v is wrong.*
2. *$\nexists v' \in V$, $(v \to v') \in E$ and v' is wrong.*
3. *$\forall v' \in V$, $r \to^* v' \to^* v$, v' is wrong.*

Therefore, when all the children of a node with a wrong computation (if any) are correct, the node becomes buggy and the debugger locates the bug in the part of the program associated with this node [10]. If a bug symptom is detected then algorithmic debugging is complete [11]; hence, if all the questions are answered, the bug will eventually be found.

Due to the fact that questions are asked in a logical order, *top-down search* [1] is the strategy that has been traditionally used (see, e.g., [3, 4, 7]) to measure the performance of different debugging tools and methods. It basically consists of a top-down, left-to-right traversal of the ET. When a node is answered NO, one of its children is asked. When the node is answered YES, the next node asked is one of its siblings. Therefore, the node asked is always a child or a sibling of the previous question node. Hence, the idea is to follow the path of wrong computations from the root of the tree to the buggy node.

However, selecting always the leftmost child does not take into account the size of the subtrees that can be explored. Binks proposed in [2] a variant of top-down search in order to consider this information when selecting a child. This variant is called *heaviest first* because it always selects the child with the biggest subtree. The objective is to avoid selecting small subtrees that have a lower probability of containing a bug.

Another important strategy is *divide and query* (D&Q) [11], which always selects the node whose subtree's size is the closest one to half the size of the whole tree. If the answer is YES, this node (and its subtree) is pruned. If the answer is NO the search continues in the subtree rooted at this node. This strategy asks, in general, less questions than top-down search because it prunes near half of the tree with every question. However, its performance is strongly dependent on the structure of the ET. If the ET is balanced, this strategy is query-optimal.

There are many other strategies: variants of top-down search [9, 14], variants of D&Q [5], and others [8, 12]. A comparison of strategies can be found in [13]. In general, all of them are strongly influenced by the structure of the ET.

Example 2 *An algorithmic debugging session for the ET in Figure 2 is the following (`YES` and `NO` answers are provided by the programmer):*

```
Starting Debugging Session...

(1)         p.castling(tower,king)       ? NO
     king.x=5                 king.x=3
     king.y=1                 king.y=1
     tower.x=8                tower.x=11
     tower.y=1                tower.y=1

(2)  t.x=8      t.right()   t.x=9       ? YES
     t.y=1                  t.y=1

(3)  t.x=9      t.right()   t.x=10      ? YES
     t.y=1                  t.y=1

(4)  t.x=10     t.right()   t.x=11      ? YES
     t.y=1                  t.y=1

(5)  k.x=5      k.left()    k.x=4       ? YES
     k.y=1                  k.y=1

(6)  k.x=4      k.left()    k.x=3       ? YES
     k.y=1                  k.y=1

Bug found in method:
castling(Position t, Position k)
of class Chess
```

The debugger points out the part of the code *which contains the bug. In this case,* `t.x!=8` *should be* `t.x==8`*. Note that, to debug the program, the programmer only has to answer questions. It is not even necessary to see the code.*

2 Collapsing and projecting nodes

Even though the strategy heaviest first significantly improves top-down search, its performance strongly depends on the structure of the ET. The more balanced the ET is, the better. Clearly, when the ET is balanced, heaviest first is much more efficient because it prunes more nodes after every question. If the ET is completely balanced, heaviest first is equivalent to divide and query and both are query-optimal.

2.1 Advantages of collapsing and projecting nodes

Our technique is based on a series of transformations that allows us to collapse/project some nodes of the ET. A collapsed node is a new node that replaces some nodes (they are removed from the ET). In contrast, a projected node is a new node that is placed as the parent of a set of nodes (they remain in the ET). This section describes with an example the main advantages of collapsing/projecting nodes in the ET:

- *Balancing execution trees.* If we augment an ET with projected nodes, we can strategically place the new projected nodes in such a way that the ET becomes balanced. With a balanced ET, the debugger can speed up the debugging session by reducing the number of asked questions.

  Example 3 *Consider again the program in Figure 1. The portion of the ET associated with the method invocation* `p.castling(tower,king)` *is shown in Figure 2. We can add projected nodes to this ET as depicted in Figure 3.*

  Note that now the ET became balanced, and hence, many strategies perform less questions. For instance, in the worst case, using the ET of Figure 2 the debugger would ask all the nodes before the bug is found. This is due to the broad nature of this ET that prevents strategies from pruning any node. In contrast, using the ET of Figure 3 the debugger prunes almost half of the tree with every question. In this example, with the standard ET, D&Q produces the following debugging session (numbers refer to the codes of the nodes in the figure):

  ```
  Starting Debugging Session...
  (1) NO,  (2) YES,  (3) YES,  (4) YES,
  (5) YES,  (6) YES
  Bug found in method:
  castling(Position t, Position k)
  of class Chess
  ```

  In contrast, with the ET of Figure 3, D&Q produces the following debugging session:

  ```
  Starting Debugging Session...
  (1) NO,  (2) YES,  (3) YES
  Bug found in method:
  castling(Position t, Position k)
  of class Chess
  ```

- *Skipping repetitive questions.* Algorithmic debuggers tend to repeat the same (or very similar) question several times when this question is associated with a method invocation which is inside a loop. In our example, this happens, for instance, in sentence `for(int i=1; i<=3; i++) {t.right();}` which is used to move the tower three positions to the right. In this case, the three questions

  ```
  {tower.x=1, tower.y=1} tower.right()
  {tower.x=2, tower.y=1}
  {tower.x=2, tower.y=1} tower.right()
  {tower.x=3, tower.y=1}
  {tower.x=3, tower.y=1} tower.right()
  ```

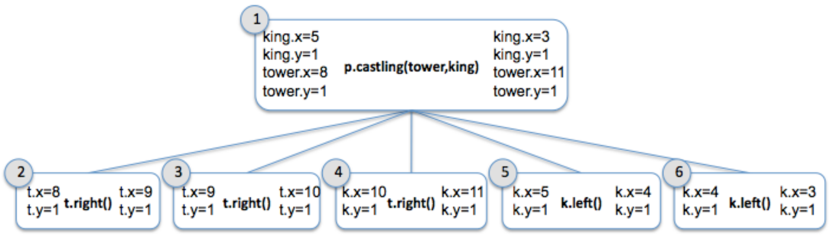
1
king.x=5
king.y=1
tower.x=8
tower.y=1
p.castling(tower,king)
king.x=3
king.y=1
tower.x=11
tower.y=1
2
t.x=8 t.y=1 t.right() t.x=9 t.y=1
3
t.x=9 t.y=1 t.right() t.x=10 t.y=1
4
k.x=10 k.y=1 t.right() k.x=11 k.y=1
5
k.x=5 k.y=1 k.left() k.x=4 k.y=1
6
k.x=4 k.y=1 k.left() k.x=3 k.y=1

Figure 2: ET associated with the call `p.castling(tower,king)` of the program in Fig. 1

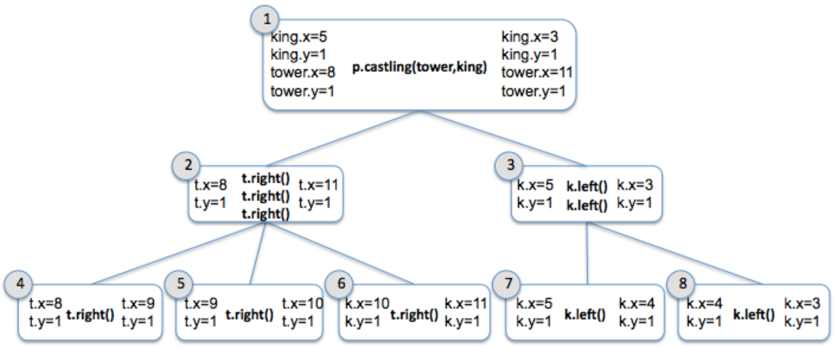
1
king.x=5
king.y=1
tower.x=8
tower.y=1
p.castling(tower,king)
king.x=3
king.y=1
tower.x=11
tower.y=1
2
t.x=8 t.y=1 t.right() t.right() t.right() t.x=11 t.y=1
3
k.x=5 k.y=1 k.left() k.left() k.x=3 k.y=1
4
t.x=8 t.y=1 t.right() t.x=9 t.y=1
5
t.x=9 t.y=1 t.right() t.x=10 t.y=1
6
k.x=10 k.y=1 t.right() k.x=11 k.y=1
7
k.x=5 k.y=1 k.left() k.x=4 k.y=1
8
k.x=4 k.y=1 k.left() k.x=3 k.y=1

Figure 3: Balanced ET associated with the call `p.castling(tower,king)` of the program in Fig. 1

```
{tower.x=4, tower.y=1}
```

could be projected to the question

```
{tower.x=1, tower.y=1}
tower.right(); tower.right();
tower.right()
{tower.x=4, tower.y=1}
```

This kind of projection, where all the projecting nodes refer to the same method, has an interesting property: If the projecting nodes are leaves, then they can be deleted from the ET. The reason is that the new projected node and the projecting nodes refer to the same method. Therefore, it does not matter what computation produced the bug, because the bug will necessarily be in the same method. Hence, if the projected node is wrong, then the bug is in the method pointed by this node. When the children of the projected node are removed, we call it *collapsed node*.

Note that, in this case, the idea is not to add nodes to the ET as in the previous case, but deleting them. Because the input and output of all the questions affect to the same attributes (i.e., x and y) the user can answer them all together, since they are, in fact, a sequence of operations whose output is the input of the next question (i.e., they are enchained). Therefore, the collapse allows us to treat a set of questions as a whole. This is particularly interesting because it approximates the real behavior intended by the programmer. For instance, in this example, the intended meaning of the loop was to move the tower three positions to the right. The intermediate positions are not interesting for the programmer, only the initial and final positions are meaningful for the intended meaning.

Example 4 *Consider again the ET of Figure 3. Observe that, if the projected nodes are wrong, then the bug must be in the unique method that appears in the projected node. Therefore, we could collapse the node instead of projecting it. Hence, nodes 4, 5, 6, 7 and 8 could be removed; and thus, with only three questions we could discover any bug in any node.*

- *Speeding up algorithmic debugging.* One important problem of algorithmic debugging strategies is that they must use a given ET without any possibility of changing it. This often prevents algorithmic debuggers from asking questions that prune a big part of the ET, or from asking questions that concentrate on the regions with a higher probability of containing the bug. The use of collapsed nodes can help to solve these drawbacks.

  The initial idea of this section was to use collapsed nodes to balance the ET. This idea is very interesting in combination with D&Q, because it can cause the debugging session to be optimal in the worst case (the query complexity of a balanced tree is $O(b \cdot log\ n)$ being b the branching factor and n the number of nodes in the ET). However, this idea could be further extended in order to force the strategies to ask questions related to parts of the computations with a higher probability of containing the bug. Concretely, we can replace parts of the ET with a collapsed node in order to avoid questions related to this part. If the debugging session determines that the collapsed node is wrong, we can expand it again to continue the debugging session inside this node. Therefore, with this idea, the original ET is transformed into a tree of ETs that can be explored when it is required. Let us illustrate this idea with an example.

  Example 5 *Consider the ET shown at the top of Figure 4. This ET has a root that started two subcomputations. The computation on the left performed eight method invocations, while the computation on the right performed only three. Therefore, in this ET, all the existing algorithmic debugging strategies would explore first the left subtree.*[1] *If we balance the left branch by inserting collapsed nodes we get the new ET shown below the previous one. This balanced ET requires (on average) less questions than the previous one; but the strategies will still explore first the left branch of the root.*

  Now, let us assume that the debugger identified the right branch as more likely to be buggy (e.g., because it contains recursive calls, because it is non-deterministic, because it contains calls with more arguments involved or with complex data structures...). We can change the structure of the ET in order to make algorithmic debugging strategies to explore the desired branch. In this example we can skip from the ET the nodes that were projected. The new ET is shown on the right of Figure 4. With this ET the debugger will explore first the right branch of the root. Observe that it is not necessary that the nodes that were projected refer to the same method. They can be completely different and independent computations. However, if the debugger determines that they are probably correct, they can be omitted to direct the search to other parts of the ET. Of course, they can be expanded again if required by the strategies.

- *Reducing the size of the ET.* One important problem of modern algorithmic debuggers is scalability. With realistic programs, the size of the ET can be huge (indeed gigabytes) and, thus, it does not fit in main memory. The same scalability problem affects graphical user interfaces (GUI). Loading the whole ET in the GUI is often too slow as to be use-

[1]Current strategies assume that all nodes have the same probability of being buggy, therefore, heavy branches are explored first.

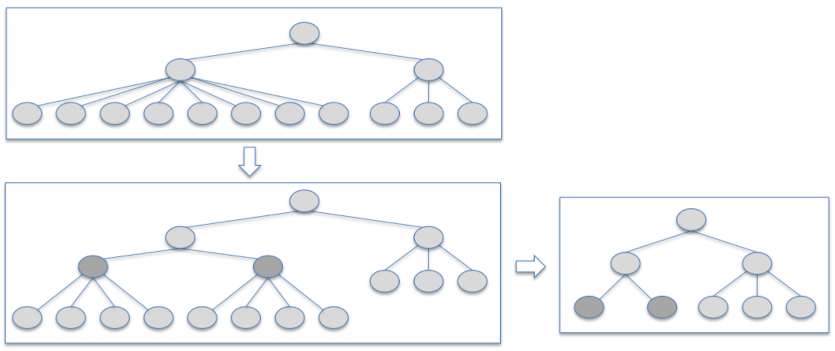

Figure 4: Transformation of ETs

ful, and it is often impossible because, again, all graphical objects do not fit in the graphical memory. Modern algorithmic debuggers such as DDJ [6] incorporate a clustering mechanism to avoid the load of the whole ET in main memory. Our experiments with DDJ and the balancing technique of this paper show that the collapsing of nodes allows us to increase the amount of ET levels shown to the user. For instance, some programs only allow the debugger to load 4 levels of the ET in the GUI (because the next level would produce a memory overflow). With the collapsing technique, we could load 5 levels due to the reduction of nodes. In particular, some loops contain hundreds of nodes that are collapsed into a single node. This saved memory is used to show nodes of another level in the balanced ET.

2.2 Collapsing and projecting algorithms

In this section we define a technique that allows us to balance an ET while keeping the completeness property of algorithmic debugging. The technique is based on two basic transformations for ETs (namely "*collapse chain*" and "*project chain*") and on a new data structure called *execution forest* (EF) that is a generalization of the ET.

Definition 5 (Execution Forest) *An* execution forest *is a tree $t = (V, E)$ whose internal vertices V are method invocations and whose leafs are either method invocations or execution forests.*

Roughly speaking, an EF is an ET where some subtrees have been replaced (i.e., collapsed) by a single node. Observe that this recursive definition of EF is more general than the definition of ET because an ET is an instance of an EF where no collapsed nodes exist. We can now introduce the two basic transformations of our technique. Both transformations are based on the notion of *chain*. Informally, a chain is formed by an ordered set of method invocations in which the final context produced by a method of the chain is the initial context of the next method. Chains often represent a set of method invocations performed one after the other during an execution. Formally,

Definition 6 (Chain) *Given an execution forest $t = (V, E)$ and a set of nodes $C \in V$ with associated method invocations $i_1, i_2, \ldots, i_n$ we say that C is a* chain *iff*

- $\exists v \in V \mid \forall c \in C \ . \ (v \to c) \in E$*, and*
- $\forall j, 1 \leq j \leq n-1$*, if* $i_j = (a_1, m_1, a_2)$ *then* $i_{j+1} = (a_2, m_2, a_3)$

It is common to find chains when one or more method invocations are performed inside a loop. When the chain is formed by a single method without children that is repeated, all the nodes that form the chain can be collapsed into a single node and the chain is deleted from the EF. This new collapsed node only needs to show to the user the initial and the final context of the chain.

Algorithm 1 Collapse Chain

```
Input: An EF $t = (V, E)$ and a set of nodes $C \subset V$
Output: An EF $t' = (V', E')$
Preconditions: $C$ is a chain with nodes $(a_1, m_1, a_2)$, $(a_2, m_2, a_3)$, ..., $(a_n, m_n, a_{n+1})$ and $\nexists\ v \in V\ .\ (c \rightarrow v) \in E$, with $c \in C$

begin
$parent = u \in V \mid (u \rightarrow c) \in E \wedge c \in C$
$colnode = (a_1, m, a_{n+1})$
          with $m = m_1; m_2; \ldots; m_n$
$V' = (V \setminus C) \cup \{colnode\}$
$E' = ((E \setminus \{(v_1 \rightarrow v_2) \in E \mid v_2 \in C\})$
        $\cup \{(parent \rightarrow colnode)\}$
end

return $t' = (V', E')$
```

The other transformation for chains is based on the projection of a chain producing a new (parent) node whose question represents the whole chain. In this case, the chain remains in the EF. This transformation is very convenient to balance the EF.

We showed in the previous section that collapsing nodes can be very useful. However, collapsing nodes is not always a good idea, because it can introduce difficult questions that delay the debugging session. Our method for balancing EFs is implemented by Algorithm 4. This algorithm first uses Algorithm 3 to shrink the EF by collapsing all chains formed with leaves; and then it balances the EF by projecting some nodes. Algorithm 4 is able to balance the EF while it is being computed. Concretely, the algorithm should be executed for each node of the EF that is completed (i.e., the result of the method invocation is already calculated, thus all the children of this node are also completed).

After several experiments, we found a situation where collapsing nodes often produces

Algorithm 2 Project Chain

```
Input: An EF $t = (V, E)$ and a set of nodes $C \subset V$
Output: An EF $t' = (V', E')$
Preconditions: $C$ is a chain with nodes $(a_1, m_1, a_2)$, $(a_2, m_2, a_3)$, ..., $(a_n, m_n, a_{n+1})$

begin
$parent = u \in V \mid (u \rightarrow c) \in E \wedge c \in C$
$projnode = (a_1, m, a_{n+1})$
          with $m = m_1; m_2; \ldots; m_n$
$V' = V \cup \{projnode\}$
$E' = ((E \setminus \{(v_1 \rightarrow v_2) \in E \mid v_2 \in C\})$
        $\cup \{(parent \rightarrow projnode)\}$
        $\cup \{(projnode \rightarrow c) \mid c \in C\}$
end

return $t' = (V', E')$
```

good results; this situation happens when the collapsed nodes form a chain. Therefore, the algorithm only projects and collapses nodes that belong to a chain. When the chain is composed by a single function and all the nodes of the chain are leafs, the whole chain can be replaced by a single collapsed node. All the collapsed nodes are computed first, and then the projected nodes are calculated. If the chain is very long, it can be cut in several subchains to be projected and thus balance the EF. In order to cut chains we use function *cutChain*:

```
function $cutChain$(chain $\{c_1, ..., c_n\}$, int $i, j$)
  if card($\{c_1, \ldots, c_{i-1}\}$)<=1
  then $s_{ini} = \emptyset$
  else $s_{ini} = \{c_0, \ldots, c_{i-1}\}$ end if
  if card($\{c_{j+1}, \ldots, c_n\}$)<=1
  then $s_{fin} = \emptyset$
  else $s_{fin} = \{c_{j+1}, \ldots, c_n\}$ end if
return $(s_{ini}, s_{fin})$
```

Another conclusion of our experiments is that all these transformations must be only done when the produced collapsed node is not very hard to answer. A good measure to achieve this is counting the number of changes in the state produced by the chain. In our implementation, we took as a design decision that no collapsed nor projected node contains more than five different changes in the state. In particular, we only collapse chains where the number of *different* attributes changed is not higher than five. Note that, in object-oriented languages, attributes can be objects. Therefore, in our implementation, any change in the state of the object-valued attribute is taken into account as a change in the chain. However, if the same attribute is changed later, it is not considered as a new change in the state.

Algorithm 3 Shrink EF

```
  **Input:** An EF $t = (V, E)$
  **Output:** An EF $t' = (V', E')$
  **Preconditions:** Given a node $v$, $v.method$ is the name of the method in $l(v)$
  **Initialization:** $t' = t$,
  set $\mathcal{S}$ contains all the chains of $t$ such that for all chain $C = \{c_1, \ldots, c_n\}$ of $\mathcal{S}$, $\forall x,\ 1 \leq x \leq n-1,\ c_x.method == c_{x+1}.method$, and $\nexists\ v \in V\ .\ (c \to v) \in E$, with $c \in C$

  begin
  while ($\mathcal{S} \neq \emptyset$)
    take a chain $C = \{c_1, \ldots, c_n\}$ of $\mathcal{S}$
    $\mathcal{S} = \mathcal{S} \backslash \{C\}$
    $t' = collapseChain(t', C)$
  end while
  end

  return $t'$
```

3 Correctness

Our technique for balancing EF is based on the transformations presented in the previous section. We now prove that these transformations do not prevent the debugger from finding the bug.

Theorem 1 (Chain Projection Correctness) *Let $t = (V, E)$ and $t' = (V', E')$ be two EF, and let C be a chain of V such that* `projectChain`$(t, C) = t'$.

1. *If t contains a buggy node, then t' also contains a buggy node.*

Algorithm 4 Shrink & Balance EF

```
  **Input:** An EF $t = (V, E)$ whose root is $root \in V$
  **Output:** An EF $t' = (V', E')$
  **Preconditions:** Given a node $v$, $v.weight$ is the size of the subtree rooted at $v$

  begin
  $t' = shrink(t)$
  $childs = \{v \in V' \mid (root \to v) \in E'\}$
  $\mathcal{S} = \{c \mid c$ is a chain in $childs\}$
  $rootweight = root.weight$
  $weight = rootweight/2$
  while ($\mathcal{S} \neq \emptyset$)
    $child = c \in childs \mid \nexists c' \in childs, c \neq c'$
                                  $\wedge c'.weight > c.weight$
    $distance = |weight - child.weight|$
    if ($child.weight >= weight$
     or $\nexists s, i, j$ such that $s = \{c_1, \ldots, c_n\} \in S$
      and ($|W - weight| < distance$)
      with $W = \sum_{x=i}^{j} c_x.weight$)
    then $childs = childs \backslash \{child\}$
      $rootweight = rootweight - child.weight$
      $weight = rootweight/2$
      if ($\exists s \in \mathcal{S} \mid s = \{c_1, \ldots, c_n\}$
                  and $child = c_i, 1 \leq i \leq n$)
      then $(s_{ini}, s_{fin}) = cutChain(s, i, i)$
           $\mathcal{S} = (\mathcal{S} \backslash \{s\}) \cup s_{ini} \cup s_{fin}$
      end if
    else
     find an $s, i, j$ such that $s = \{c_1, \ldots, c_n\} \in S$ and $\sum_{x=i}^{j} c_x.weight$ has a weight as close as possible to $weight$
      $s' = \{c_i, \ldots, c_j\}$
      $(s_{ini}, s_{fin}) = cutChain(s, i, j)$
      $\mathcal{S} = (\mathcal{S} \backslash \{s\}) \cup s_{ini} \cup s_{fin}$
      $t' = projectChain(t', s')$
      for each $c \in s'$
        $rootweight = rootweight - c.weight$
      end for each
      $childs = (childs \backslash s')$
      $weight = rootweight/2$
    end if
  end

  return $t' = (V', E')$
```

2. *If t' contains a buggy node n associated with method m, then m contains a bug.*

Proof. Let $v \in V$ be the parent node of the chain C, and let $w \in V'$ be the projected node of C. Let us start with the proof of the first item: If t contains a buggy node, then t' also contains a buggy node. We consider three cases:

- $v \in V$ is buggy. This means, by Definition 4, that $v \in V'$ is wrong and $\forall c \in C$, c is correct. In addition, we know that if $\forall c \in C$, c is correct, then w is correct because w is the composition of the results produced in C. Therefore, we have that $w \in V'$ is also correct and, hence, $v \in V'$ is a buggy node.

- $c \in C \subset V$ is buggy. This means that both $v \in V$ and $v \in V'$ are wrong. We have two possibilities: (i) w is wrong. In such a case, $c \in C \subset V'$ is also buggy and the claim follows. (ii) w is not wrong. In such a case, $v \in V'$ is buggy according to Definition 4.

- $u \in V, v \neq u \neq c \in C$ is buggy. The claim follows trivially because the projection only affects v and C. Therefore, $u \in V'$ is equal to $u \in V$ and they have the same parent and children.

Now, we prove that if t' contains a buggy node n associated with method m, then m contains a bug. We consider four cases:

- $v \in V'$ is buggy. This means that $v \in t$ is wrong and $w \in V'$ is correct. If $\forall c \in C$, c is correct, then $v \in V$ is a buggy node, and the claim follows because both $v \in V$ and $v \in V'$ are associated with the same method.

  Otherwise, $\exists c \in C$, c is wrong. Nevertheless, the result and the final context of C are correct because w is correct. This means that chain C is not the cause of the wrong equation associated with v.

  In addition, by Definition 4, we know that all the children of $v \in V'$ are correct, but v is wrong. Thus, the method associated with v must contain a bug and the claim follows.

  This particular case of the proof is interesting because it reveals that (at least) two bugs belong to t; one is associated with v and the other is associated with c.

- $w \in V'$ is buggy. This case is not possible because it implies that either the result or the final context of w are wrong. Hence, because both the result and the final context are produced by the nodes in C, we know that at least one node is also wrong. But this is a contradiction because in such a case w cannot be buggy according to Definition 4.

- $c \in C \subset V'$ is buggy. This means that $w \in V'$ and $v \in V'$ are wrong because both of them are ancestors of c. Therefore, $v \in V$ is also wrong. And, of course, $c \in C \subset V$ is also wrong. Moreover, because the children of c are the same in V and V', we know that $c \in C \subset V$ is a buggy node, and the claim follows because both $c \in V$ and $c \in V'$ are associated with the same method.

- $u \in V', u \neq v, u \neq w$ and $u \neq c \in C$ is buggy. In this case the projection does not influence the buggy node u and thus the claim follows trivially because the method associated to u is wrong and all the method invocations associated with the children of u are correct.

□

Theorem 2 (Chain Collapse Correctness) *Let $t = (V, E)$ and $t' = (V', E')$ be two EF, and let C be a chain of V such that all the nodes in the chain are leafs and have the same associated method. Given $t' =$ `collapseChain`(t, C),*

1. *If t contains a buggy node, then t' also contains a buggy node.*

2. *If t' contains a buggy node n associated with method m, then m contains a bug.*

Proof. This proof is completely analogous to the proof of Theorem 1 except for the fact that the chain is removed from the EF.

Let $v \in V$ be the parent node of the chain C, and let $w \in V'$ be the collapsed node of C. Let us start with the proof of the first item: If t contains a buggy node, then t' also contains a buggy node. We consider three cases:

- $v \in V$ is buggy. By Definition 4, we know that $v \in V'$ is wrong and all the children of v including the nodes of in C are correct. In addition, we know that if $\forall c \in C$, c is correct, then w is correct because w is the composition of the results produced in C. Therefore, we have that $w \in V'$ is also correct and, hence, $v \in V'$ is a buggy node because all its children are correct.

- $c \in C \subset V$ is buggy. This means that both $v \in V$ and $v \in V'$ are wrong because they are ancestors of c. We have two possibilities: (i) w is wrong. In such a case, w is buggy because it has no children and because its parent $v \in V'$ is wrong. (ii) w is not wrong. In such a case, $v \in V'$ is buggy according to Definition 4.

- $u \in V, v \neq u \neq c \in C$ is buggy. The claim follows trivially because the collapse only affects v and C. Therefore, $u \in V'$ is equal to $u \in V$ and they have the same parent and children.

Now, we prove that if t' contains a buggy node n associated with method m, then m contains a bug. We consider three cases:

- $v \in V'$ is buggy. This means that $v \in t$ is wrong and $w \in V'$ is correct. If $\forall c \in C$, c is correct, then $v \in V$ is a buggy node, and the claim follows because both $v \in V$ and $v \in V'$ are associated with the same method.

  Otherwise, $\exists c \in C$, c is wrong. Nevertheless, the result and the final context of C are correct because w is correct. This means that chain C is not the cause of the wrong equation associated with v.

  In addition, by Definition 4, we know that all the children of $v \in V'$ are correct, but v is wrong. Thus, the method associated with v must contain a bug and the claim follows.

  Here again, this particular case of the proof reveals that at least two bugs belong to t; one is associated with v and the other is associated with c.

- $w \in V'$ is buggy. Then, either the result or the final context of w are wrong. Hence, because both the result and the final context are produced by the nodes in C, we know that at least one node is also wrong. Because the ancestors of w are wrong (by Definition 4), then at least one node in C is buggy. Moreover, because all the nodes in the chain have the same associated method, the bug must be in this method that is the only that appears in w.

- $u \in V', v \neq u \neq w$ is buggy. In this case the collapse does not influence the buggy node u and thus the claim follows trivially because the method associated to u is wrong and all the method invocations associated with the children of u are correct.

□

We provide in this section an interesting result related to the projection and collapse of chains. This result is related to the incompleteness of the technique when it is used intra-session. In particular, one could expect the following result:

> *An EF contains a buggy node associated with method m if and only if its balanced version also contains a buggy node associated with method m.*

but it is not true.

In general, our technique ensures that all the bugs that caused the wrong behavior of the root node (i.e., the wrong final state of the whole program) can be found in the balanced EF. This means that all the buggy nodes that are responsible of the wrong behavior are present in the balanced EF.

However, algorithmic debugging can find bugs by fluke. Those nodes that are buggy nodes in the EF, but did not cause the wrong behavior of the root node can be undetectable with some strategies in the balanced version of the EF. And also, the opposite is true: It is possible to find bugs in the balanced EF that were undetectable in the original EF. Let us explain it with an example.

Example 6 *Consider the following EFs:*

The EF on the right is the same as the one on the left but a new projected node has been added. If we assume the following intended semantics:

$x = 1$	$f()$	$x = 2$	$x = 1$	$g()$	$x = 4$
$x = 3$	$h()$	$x = 3$	$x = 4$	$g()$	$x = 4$

then grey nodes are wrong and white nodes are right.

Observe that in the EF on the left, only nodes 2 and 3 are buggy. Therefore, all the strategies will report these nodes as buggy, but never node 1. However, node 1 contains a bug but it is undetectable by the debugger until nodes 2 and 3 have been corrected. Nevertheless, observe that nodes 2 and 3 did not produce the wrong behavior of node 1. They simply produced two errors that, in combination, produced by fluke a global correct behavior.

Now, observe in the EF on the right that node 1 is buggy and thus detectable by the strategies. In contrast, nodes 2 and 3 are now undetectable by some strategies such as top-down search (they could be detected by D&Q). Thanks to the balancing process, it has been made explicit that three different bugs are in the EF.

The above properties of the collapsing and projecting techniques are the reason why we introduced in Definition 4 the third item. The standard definition of buggy node only includes the first two items and hence, a buggy node is any node associated with a wrong method. Contrarily, we force the buggy node to be the last node of a path from the root made of wrong equations. This restriction is enough to formulate our correctness results.

4 Implementation

We have implemented the technique presented in this paper and integrated it into the Declarative Debugger for Java DDJ 2.4. The implementation allows the programmer to activate the transformations of the technique and to parameterize them in order to adjust the number and size of the projected/collapsed nodes. It has been tested with a collection of real applications (e.g., an interpreter, a compiler, an XSLT processor, etc) producing good results.

Table 1 summarizes the results of the performed experiments. The first column contains the names of the benchmarks. The source code and other information about all the benchmarks can be found at

`http://www.dsic.upv.es/~jsilva/DDJ/examples`

Each benchmark has been evaluated assuming that the bug could be in any node. This means that each row of the table is the average of a number of experiments. For instance, `kxml2` was tested 1445 times (i.e., the experiment was repeated choosing a different node as buggy, and all nodes were tried). For each benchmark, column `ET nodes` shows the size of the ET evaluated for this benchmark; column `Col./Proj. nodes` shows the number of nodes that were projected and collapsed by the debugger. Observe that the opportunities of collapsing are very few compared to the number of projections done; column `Questions` shows the average number of questions done by the debugger before finding the bug in the original ET; column `Questions Bal.` shows the average number of questions done by the debugger before finding the bug in the balanced ET; Finally, column `(%)` shows the improvement achieved with the collapsing technique. Clearly, the collapsing technique has an important impact in the reduction of questions with a mean reduction of 30-40% using top-down.

The implementation takes advantage of one property of projected nodes: If they are wrong, then at least one of their children is wrong. This allows us to avoid unnecessary questions. For instance, with top-down search, we can skip the question to the last child of a wrong projected node.

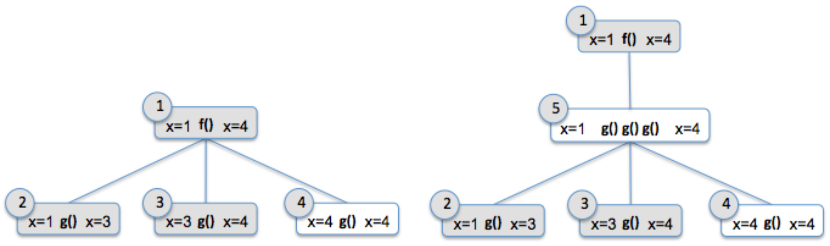

Figure 5: Counterexample

Benchmark	ET nodes	Col./Proj. nodes	Questions	Questions Bal.	%
argparser	192 nodes	0/63 nodes	22.78	15.7	68.92 %
cglib	1463 nodes	0/247 nodes	82.41	49.73	60.34 %
kxml2	1445 nodes	2/277 nodes	81.61	50.90	62.37 %
javassist	1499 nodes	5/148 nodes	83.84	61	72.76 %

Table 1: Benchmark results

Essentially, our implementation [6] produces the EF and transforms it by collapsing and projecting nodes with an implementation of Algorithm 4. Finally, it is explored with standard strategies starting the debugging session at any node selected by the user.

Our algorithm is very conservative because it only collapses or projects nodes that belong to a chain. Hence, the transformation is only applied trying to ensure that the question produced is not complicated. This has produced good results, but sometimes the question of a collapsed node is hard to answer. Even in this case, our implementation ensures that if the programmer is able to find the bug with the standard ET, she will also be able with the balanced EF. That is, the introduction of projected nodes cannot produce the debugging session to stop. This is due to the possibility of answering “I don’t know.” Our debugger allows the programmer to answer “I don’t know,” skipping the current question and continuing the debugging process with the other questions (e.g., with the children). Therefore, even if the programmer cannot answer a projected question, she can continue the debugging session, thus projected nodes can delay debugging, but not stop it.

All the information related to the experiments, the source code of the benchmarks, the bugs, the source code of the tool and other material can be found at

`http://www.dsic.upv.es/~jsilva/DDJ`

5 Conclusions

This work presentes a new technique that allows us to automatically balance standard ETs. This technique has been implemented and experiments with real applications confirm that the balancing process has a positive impact on the performance of algorithmic debugging.

From a theoretical point of view, two important results have been proved. The projection and the collapse of nodes do not prevent from finding bugs, and the bugs found after the transformations are always real bugs. Another interesting and surprising result is the fact that balancing ETs can change the order in which bugs are found by the debugger.

In our current experiments, we are now taking advantage of the Execution Forests. This data structure allows us to apply more drastic balancing transformations. For instance, it allows us to collapse a whole subtree of the EF. This permits to avoid questions related to some parts of the EF and direct the search in other direction. In this respect, we do not plan to apply this transformation to chains, but to subtrees; based on approximations of the probability of a subtree to be buggy.

Execution forests provide a new dimension in the search that allows the debugger to go into a collapsed region and explore it ignoring the rest of the EF; and also to collapse regions in order to be ignored by the search strategies.

References

[1] E. Av-Ron. *Top-Down Diagnosis of Prolog Programs.* PhD thesis, Weizmanm Institute, 1984.

[2] D. Binks. *Declarative Debugging in Gödel.* PhD thesis, University of Bristol, 1995.

[3] 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.

[4] R. Caballero. Algorithmic Debugging of Java Programs. In *Proc. of the 2006 Workshop on Functional Logic Programming (WFLP'06).* Electronic Notes in Theoretical Computer Science, 63–76, 2006.

[5] 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).* Springer LNCS 749, 153–170, 1993.

[6] D. Insa and J. Silva. Debugging with Incomplete and Dynamically Generated Exution Trees. In *Proc. of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'10).* Hagenberg, Austria. July 23-25, 2010.

[7] G. Kokai, J. Nilson, and C. Niss. GIDTS: A Graphical Programming Environment for Prolog. In *Workshop on Program Analysis For Software Tools and Engineering (PASTE'99)*, pages 95–104. ACM Press, 1999.

[8] I. MacLarty. Practical Declarative Debugging of Mercury Programs. Ph.D. thesis, Department of Computer Science and Software Engineering, The University of Melbourne. 2005.

[9] M. Maeji and T. Kanamori. Top-Down Zooming Diagnosis of Logic Programs. Tech. Rep. TR-290, ICOT, Japan. 1987.

[10] H. Nilsson and P. Fritzson. Algorithmic Debugging for Lazy Functional Languages. *Journal of Functional Programming*, 4(3):337–370, 1994.

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

[12] J. Silva. Three New Algorithmic Debugging Strategies. In *Proc. of VI Jornadas de Programación y Lenguajes (PROLE'06).* 243–252, 2006.

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

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