Speeding Up Algorithmic Debugging Using Balanced Execution Trees*

David Insa, Josep Silva, and Adrián Riesco

Departamento de Sistemas Informáticos y Computación,
Universitat Politècnica de València, Valencia, Spain
{dinsa,jsilva}@dsic.upv.es
Departamento de Sistemas Informáticos y Computación,
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 performed during the execution of a program. This data structure is the so-called *Execution Tree* and it strongly influences the performance of the technique. In this work we present a transformation that automatically improves the structure of the execution trees by collapsing and projecting some strategic nodes. This improvement in the structure implies a better behavior and performance of the standard algorithms that traverse it. 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 also be 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

Debugging is one of the most difficult and less automated tasks in object-oriented programming. The most extended technique is still based on the use of breakpoints. Breakpoints allow the programmer to manually inspect a computation. She must decide where to place the breakpoints and use them to inspect the values of strategic variables. Between the point where the effect of a bug is observed, and the point where the bug is located, there can be hundreds or thousands of lines of code, thus, the programmer must navigate the computation in order to find the bug. Ideally, breakpoints should be placed in a way that the amount of code inspected is reduced as much as possible.

There exists a debugging technique called *algorithmic debugging* (AD) [14] that tries to automate the problem of inspecting a computation in order to find a bug. AD is a semi-automatic debugging technique that produces a dialogue between the debugger and the programmer to locate bugs. This technique relies on the programmer having an *intended interpretation* of the program. In other words, some computations of the

* This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grants TIN2008-06622-C03-02 and TIN2012-39391-C04-04, by the *Generalitat Valenciana* under grant ACOMP/2009/017, and by the *Comunidad de Madrid* under grant S2009/TIC–1465. David Insa has been partially supported by the spanish *Ministerio de Educación* under grant AP2010-4415.

program are correct and others are wrong with respect to the semantics intended by the programmer. Therefore, algorithmic debuggers compare the actual results of subcomputations with those expected by the programmer. By asking questions to the programmer or using a formal specification the system can identify precisely the location of a bug.

The idea behind AD is that the own debugger selects automatically the places that should be inspected. The advantage is that the debugger knows a priori the length of the computation, and thus it can perform a dichotomic search inspecting the subcomputation located in the middle of the remaining suspicious area. This allows the debugger to efficiently explore the computation. Moreover, the places inspected are always method invocations and thus, it is easy for the programmer to decide if the result produced by an invocation with given arguments is correct or not.

Conceptually, AD is a two-phase process: During the first phase, a data structure that represents the execution of the program, the *Execution Tree* (ET), is built; while in the second phase a *navigation strategy* is used to iteratively select nodes in the ET. The debugger asks a question related to each selected node to an external oracle (typically the user). With every answer the debugger prunes a part of the ET until a single node remains. The bug is located in the method associated with this node.

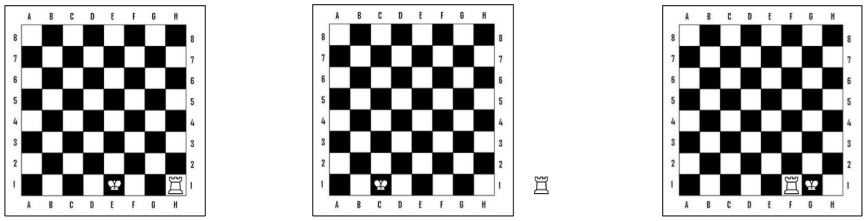
In this work we introduce a technique that changes the structure of the ET in such a way that the navigation strategies present an almost optimal behavior. The objective of the technique is to balance[1] the ET in such a way that navigation strategies prune half of the ET at every step, because they can always find a node that divides the search area in half. Our experiments with real programs show that the technique reduces (as an average) the number of questions to the oracle by around 30%.

In order to show the general idea of the technique with an example, we need to explain first how the ET is constructed. The ET contains nodes that represent subcomputations of the program. Therefore, the information of the ET's nodes refer to method executions. Without loss of generality, we will base our examples on programs implemented using the Java language, although our ET transformations and the balancing technique are conceptually applicable to any ET, independently of the language it represents.

Our technique is particularly useful for object-oriented programs because (i) it allows the programmer to ignore the operational details of the code to concentrate on the validity of the effects produced by method executions. In general, the programmer does not need to look at the source code to interact with the debugger: she only has to answer questions related to the effects of the computations. (ii) The technique is very powerful in presence of iterative loops which are inexistent in other paradigms (e.g., functional).

In the object-oriented paradigm, an ET is constructed as follows: Each node of the ET is associated with a method execution. It contains all the information needed to decide whether the method execution produced a correct result. This information includes the call to the method with its arguments and the result, and the values of all the attributes that are in the scope of this method, at the beginning and at the end of the execution (observe, e.g., that exception objects are also in the scope). This information allows the programmer to know whether all the effects of the method execution correspond to her intended semantics. The root node of the ET is the initial method execution of the program (e.g., *main*). For each node n with associated method m, and for each

[1] Note that throughout this paper we use *balanced* to indicate that the tree becomes *binomial*.



Initial position Final position Expected position

```
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();}
    }
  }
}
public class Position {
    int x, 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;}
 }
```

Fig. 1. Example program

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

Example 1. Consider the Java program in Figure 1. This program has a bug, and thus it wrongly simulates a movement on a chessboard. The call `p.castling(tower,king)` produces the (wrong) movement shown in the chessboards of the figure. Figure 2 depicts the portion of the ET associated with the method call `p.castling(tower,king)`. With this ET, all current navigation strategies need to ask about the six nodes. In contrast, if we balance this ET with our transformation, the bug is found with at most three questions.

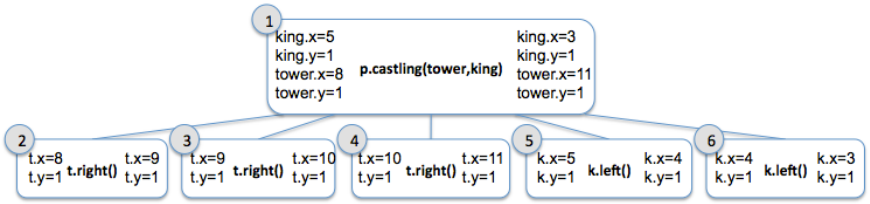


Fig. 2. ET associated with the call `p.castling(tower,king)` of the program in Figure 1

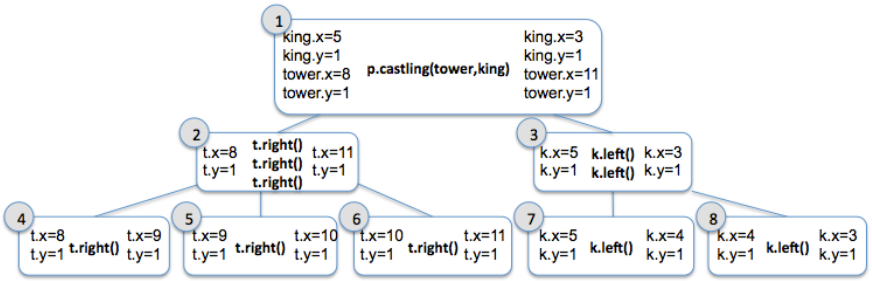


Fig. 3. Balanced ET associated with the call `p.castling(tower,king)` for Figure 1

Our technique presents three important advantages that make it useful for AD. First, it can be easily adapted to other programming languages. We have implemented it for Java and it can be directly used in other object-oriented languages, but it could be easily adapted to other languages such as C or Haskell using the analogy between methods and functions. Second, the technique is quite simple to implement and can be integrated into any existing algorithmic debugger with small changes. And third, the technique is conservative. If the questions triggered by the new nodes are difficult to answer, the user can answer "I don't know" and continue the debugging session as in the standard ETs. Moreover, the user can naturally get back to the original ET in case it is needed.

The rest of the paper has been organized as follows. In Section 2 we introduce some preliminary definitions that will be used in the rest of the paper. In Section 3 we explain our technique and its main applications, and we introduce the algorithms used to balance ETs. The correctness of the technique is proved in Section 4. Then, in Section 5, we present our implementation and some experiments carried out with real Java programs. Section 6, discusses the related work. Finally, Section 7 concludes.

2 Algorithmic Debugging

In this section we introduce some notation and formalize the notion of execution tree used in the rest of the paper. For the purpose of this work, we consider ETs as labeled trees. We need to formally define the notions of context and method execution before we provide a definition of ET.

Definition 1 (Context). *Let $\mathcal{P}$ be a program, and X the execution of a method in $\mathcal{P}$. The* context *of X at a particular instant t is $\{(a,v) \mid a$ is an attribute in the scope of X at instant t and v is the value of $a\}$.*

Roughly, the context of a method at a particular instant of its execution is composed of all the variables of the program that are visible at this moment. 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 execution. All this information (the context at the beginning and at the end of the execution of the method) should be visualized together with the call to the method so that the programmer can decide whether it is correct.[2]

Definition 2 (Method Execution). *Let $\mathcal{P}$ be a program and X an execution of $\mathcal{P}$. Then, each* method execution *done in X is represented with a triple $\mathcal{E} = (b,m,e)$ where m represents the call to the method with its arguments and the returned value, b is the context of the method in m at the beginning of its execution, and e is the context of the method in m at the end of its execution. A* composite method execution *is a non empty sequence of method executions $\langle (b_1,m_1,e_1),(b_2,m_2,e_2),\ldots,(b_n,m_n,e_n)\rangle$ that we represent as $(b_1,m_1,m_2,\ldots,m_n,e_n)$.*

Thanks to the declarative properties of AD, we can ignore the operational details of an execution. From the point of view of the debugger an execution is a finite tree of method executions. This can be modeled with the following grammar:

$$T = (b, m[L], e) \qquad L = \varepsilon \qquad L = TL$$

where the terminal m is a method of the program and b and e represent the context at the beginning and at the end of the execution of the method. For instance, `p.castling(tower,king)` in Example 1 can be represented with the tree:

$$\begin{array}{l}(b_1, \texttt{p.castling}(\texttt{tower},\texttt{king}) \\ \quad [(b_2, \texttt{t.right}()[], e_1), (e_1, \texttt{t.right}()[], e_2), (e_2, \texttt{t.right}()[], e_3), \\ \quad (b_3, \texttt{k.left}[], e_4), (e_4, \texttt{k.left}[], e_5)], e_6)\end{array}$$

With this tree, we can construct the ET in Figure 2. Roughly speaking, an ET is a tree whose nodes represent method executions and the parent-child relation is defined by the tree produced by the grammar. Formally,

Definition 3 (Execution Tree). *Given a program $\mathcal{P}$ and a method execution $\mathcal{E}$, the* execution tree *(ET) of $\mathcal{P}$ with respect to $\mathcal{E}$ is a tree $t = (V,E)$ where $\forall v \in V$, v is a composite method execution, and*

- *The root of the ET is $\mathcal{E}$.*
- *For each pair of method executions $\mathcal{E}_1$, $\mathcal{E}_2 \in V$, we have that $(\mathcal{E}_1 \rightarrow \mathcal{E}_2) \in E$ iff*
  1. *during the execution of the method associated with $\mathcal{E}_1$, the method associated with $\mathcal{E}_2$ is invoked, and*
  2. *from the instant when $\mathcal{E}_1$ starts until the instant when $\mathcal{E}_2$ starts, there does not exist a method execution $\mathcal{E}_3$ such that $\mathcal{E}_3$ has started but not ended.*

[2] We refer the reader to Appendix B for a brief explanation about how this information is displayed in our implementation.

Note that we use $(v \to v')$ to denote a directed edge from v to v'. From now on, we will assume that there exists an intended semantics I of the program being debugged. It corresponds to the model the programmer had in mind while writing the program, and it contains, for each method m and each context b of m at the beginning of its execution, the expected context e at the end of its execution, that is, $(b,m,e) \in I$. Moreover, given this atomic information, we are able to deduce judgments of the form $(b,m_1;\ldots;m_n,e)$ with the inference rule Tr, that defines the transitivity for the composition of methods

$$\frac{\overline{(b,m_1,e')} \quad \overline{(e',m_2;\ldots;m_n,e)}}{(b,m_1;\ldots;m_n,e)} \ \mathsf{Tr} \text{ if } n > 1$$

and we say that $I \models (b,m_1;\ldots;m_n,e)$. Using this intended semantics we can formally define the correctness of method executions.

Definition 4 (Correctness of method executions). *Given a method execution $\mathcal{E}$ and the intended semantics of the program I, we say that $\mathcal{E}$ is* correct *if $\mathcal{E} \in I$ or $I \models \mathcal{E}$ and* wrong *otherwise.*

Once the ET is built, in the second phase the debugger uses a strategy to traverse the ET asking the oracle about the correctness of the information stored in each node. If the method execution of a node is wrong, it answers NO. Otherwise, it answers YES. Using the answers, the debugger identifies a buggy node that is associated with a buggy source code of the program.

Definition 5 (Buggy node). *Given an ET $t = (V,E)$, a buggy node of t is a node $v \in V$ such that (i) the method execution of v is wrong and (ii) $\forall v' \in V$, $(v \to v') \in E$, v' is correct.*

According to Definition 5, when all the children of a node with a wrong computation (if any) are correct, the node becomes buggy and the debugger locates a bug in the part of the program associated with this node [12]. A buggy node detects a *buggy method*, which informally stands for methods that return an incorrect context even though all the methods executions performed by them are correct.

Lemma 1 (Buggy method). *Given an ET $t = (V,E)$, and a buggy node $v \in V$ in t with $v = (b,m,e)$, then m contains a bug.*

Proofs of all technical results have been included in Appendix A.

Due to the fact that questions are asked in a logical order (i.e., consecutive questions refer to related parts of the computation), *top-down search* is the strategy that has been traditionally used (see, e.g., [2, 3, 8]) to measure the performance of different debugging tools and methods. It basically consists of a top-down (assuming that the root is on top), left-to-right traversal of the ET. When the answer to the question of a node is NO, then the next question is associated with one of its children. When the answer is YES, the next question is associated with one of its siblings. Therefore, the node asked is always a child or a sibling of the previous asked 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 [1] 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) [13], that always selects the node whose subtree's size is the closest one to half the size of the 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, fewer 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 [10, 5], variants of D&Q [6], and others [9, 14]. A comparison of strategies can be found in [14]. In general, all of them are strongly influenced by the structure of the ET.

Example 2. An AD session for the ET in Figure 2 using D&Q follows (`YES` and `NO` answers are provided by the programmer):

```
Starting Debugging Session...
(2)  t.x=8,  t.y=1  >>>  t.right()  >>>  t.x=9,  t.y=1    ? YES
(3)  t.x=9,  t.y=1  >>>  t.right()  >>>  t.x=10, t.y=1    ? YES
(4)  t.x=10, t.y=1  >>>  t.right()  >>>  t.x=11, t.y=1    ? YES
(5)  k.x=5,  k.y=1  >>>  k.left()   >>>  k.x=4,  k.y=1    ? YES
(6)  k.x=4,  k.y=1  >>>  k.left()   >>>  k.x=3,  k.y=1    ? YES
(1)  king.x=5,                                   king.x=3,
     king.y=1,  >>>  p.castling(tower,king)  >>>  king.y=1,   ? NO
     tower.x=8,                                  tower.x=11,
     tower.y=1                                   tower.y=1
Bug found in method: castling(Position t, Position k) of class Chess.
```

The debugger points out the buggy method, 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.

3 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.

3.1 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 that are then removed from the ET. In contrast, a projected node is a new node that is placed as the parent of a set of nodes that remain in the ET. This section describes the main advantages of collapsing/projecting nodes:

Balancing execution trees. If we augment an ET with projected nodes, we can strategically place the new nodes in such a way that the ET becomes balanced. In this way, the debugger speeds 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 `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 becomes balanced, and hence, many strategies ask fewer questions. For instance, in the worst case, using the ET of Figure 2 the debugger would ask about all the nodes before the bug is found. This is due to the broad nature of this ET that prevents strategies from pruning any nodes. 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 of Figure 2, D&Q produces the following debugging session (numbers refer to the codes of the nodes in the figure):

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

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

```
Starting Debugging Session...
(2) YES  (3) YES  (1) NO
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 it is associated with a method execution that is inside a loop. In our example, this happens in `for(int i=1; i<=3; i++) {t.right();}`, which is used to move the tower three positions to the right. Here, the nodes

```
{t.x=8,  t.y=1}  t.right()  {t.x=9,  t.y=1}
{t.x=9,  t.y=1}  t.right()  {t.x=10, t.y=1}
{t.x=10, t.y=1}  t.right()  {t.x=11, t.y=1}
```

could be projected to the node

```
{t.x=8, t.y=1}  t.right(); t.right(); t.right()  {t.x=11, t.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 this method. Hence, if the projected node is wrong, then the bug is in the method pointed to 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 to delete them. Because the input and output of all the questions relate to the same attributes (i.e., x and y), then 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 chained). Therefore, this technique 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; only the initial and final ones are meaningful for the intended meaning.

Example 4. Consider the ET of Figure 3. Observe that, if the projected nodes are wrong, then the bug must be in the unique method appearing in the projected node. Thus, 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.

Fig. 4. Transformation of ETs.

Enhancing the search of algorithmic debugging. One important problem of AD strategies is that they must use a given ET without any possibility of changing it. This often prevents strategies from selecting nodes that prune a big part of the ET, or from selecting nodes that concentrate on the regions with a higher probability of containing the bug. Collapsing some subtrees into a single node can help to solve these drawbacks.

The initial idea of this section was to use projected 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 (its query complexity is $O(b \cdot log\ n)$, where b is the branching factor and n is 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 leftmost ET in Figure 4. This ET has a root that started two subcomputations. The computation on the left performed ten method executions, while the computation on the right performed only three. Hence, in this ET, all the existing strategies would explore first the left subtree.[3] If we balance the left branch by inserting projected nodes we get the new ET shown on the right of the previous one. This balanced ET requires (on average) fewer questions than the previous one; but the strategies will still explore the left branch of the root first.

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 AD strategies start by exploring the desired branch. In this example we can remove from the ET the nodes that were projected. The new ET is shown on the right of Figure 4. With this ET the tool explores 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 strategy (e.g., if the debugger cannot find the bug in the other nodes).

Disadvantages Given these benefits, we must talk of a potential drawback: the difficulty of the questions related to the new nodes. The new questions may be more difficult to

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

answer than the previous ones, but the user can avoid these difficult questions as we will discuss later. However, the nodes encapsulated in the new projected/collapsed are intimately related, as we explain in the next section, and thus in many situations the question is more related to the behavior the user had in mind when writing the code than the original questions (i.e., the behavior of the whole loop vs. the behavior of each single call inside the loop).

3.2 Collapsing and projecting algorithms

In this section we define a technique that allows us to balance an ET while keeping the soundness and completeness of AD. The technique is based on two basic transformations for ETs (namely *collapse leaf chain* and *project chain*, described respectively by Algorithms 1 and 2), and on a new data structure called an *Execution Forest* (EF) that is a generalization of an ET.

Definition 6 (Execution Forest). *An* execution forest *is a tree $t = (V,E)$ whose internal vertices V are method executions and whose leaves are either method executions or execution forests.*

Roughly speaking, an EF is an ET where some subtrees have been replaced (i.e., collapsed) by a single node. Note that this recursive definition of EF is more general than the one for ET because an ET is an instance of an EF where no collapsed nodes exist. We can now define 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 sibling nodes in which the final context produced by a node of the chain is the initial context of the next node. Chains often represent a sequence of method executions performed one after the other during an execution. Formally,

Definition 7 (Chain). *Given an EF $t = (V,E)$ and a set $C \subset V$ of n nodes with associated method executions $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$ we say that C is a* chain *iff*

- $\exists v \in V$ *such that* $\forall c \in C \,.\, (v \to c) \in E$,
- $\forall i, j, 1 \leq i < j \leq n$, *the method associated with $\mathcal{E}_i$ is executed before the method associated with $\mathcal{E}_j$, and*
- $\forall j, 1 \leq j \leq n-1$, *if* $\mathcal{E}_j = (e_j, m_j, e_{j+1})$
  then $\mathcal{E}_{j+1} = (e_{j+1}, m_{j+1}, e_{j+2})$

The first condition ensures that all the elements in the chain are siblings. The second condition ensures that the elements are executed one after the other. The third condition ensures that for all nodes in the chain the final context of a node is the initial context of the next chained node. Note that, by the definition of context, only those attributes that can be affected by the execution of the methods are taken into account. It is common to find chains when one or more methods are executed inside a loop.

The basic transformations of chains are described by Algorithms 1 and 2. Algorithm 1 is in charge of collapsing chains, that consists in creating a new node *colnode* with initial context the initial context of the first node of the chain, final context the final context of the last node of the chain, and with the composition of the methods in the chain as associated method. Then, the nodes in the chain (and thus their edges) are

Algorithm 1 Collapse Leaf 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
1) $parent = u \in V$ such that $\forall c \in C, (u \rightarrow c) \in E$
2) $colnode = (a_1, m, a_{n+1})$ with $m = m_1; m_2; \ldots; m_n$
3) $V' = (V \setminus C) \cup \{colnode\}$
4) $E' = ((E \setminus \{(parent \rightarrow v) \in E \mid v \in C\}) \cup \{(parent \rightarrow colnode)\}$
end
return $t' = (V', E')$

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
1) $parent = u \in V$ such that $\forall c \in C, (u \rightarrow c) \in E$
2) $prjnode = (a_1, m, a_{n+1})$ with $m = m_1; m_2; \ldots; m_n$
3) $V' = V \cup \{prjnode\}$
4) $E' = ((E \setminus \{(parent \rightarrow v) \in E \mid v \in C\}) \cup \{(parent \rightarrow prjnode)\} \cup \{(prjnode \rightarrow c) \mid c \in C\}$
end
return $t' = (V', E')$

removed from the tree and the new node is linked to the parent of the nodes in C, thus reducing the size of the EF. Algorithm 2 is in charge of projecting chains, and works in a similar way to the previous algorithm. Given a tree t and a chain C in the tree, it removes from t the edges between each $c \in C$ and its parent *parent*, and then introduces a new node *prjnode* built as explained before, that is linked to each c as their new parent, and to *parent* as its new child.

Algorithm 3 is in charge of removing the chains of leaves that can be collapsed. It first computes in the initialization all the maximal chains (i.e., chains that are not subchains of other chains) of nodes that are leaves and are related to the same methods. Then, for each of these chains, it applies Algorithm 1 to collapse them by removing the chain from the tree and adding the corresponding collapsed node.[4]

Our method for balancing EFs is implemented by Algorithm 4. This algorithm first uses Algorithm 3 to shrink the EF (line 1) by collapsing as many nodes as possible; and then it balances this shrunken EF by projecting some nodes. The objective is to divide the tree into two parts with the same weight (i.e., number of nodes). Therefore, we first compute the half of the size of the EF (lines 4 and 5). If a child of the root is already heavier than half the size of the tree, then, the weight of this node is not taken into account in the balancing process because the question associated to this node will be the first question asked (lines 9-15). Otherwise, it projects the part of a chain whose weight is as close as possible to half the weight of the root (lines 16-24). This allows us to prune half of the subtree when asking a question associated with a projected node. In the case that the heavier node (lines 13-15) or the projected chain (lines 18-19) belongs to a bigger chain, it must be cut with function *cutChain* producing new (smaller) chains

[4] Note that, since these chains are usually found when a loop or a recursive call is used, our approach generates nodes whose questions are very close to the intended meaning the programmer had in mind while developing the program and thus, although the new questions comprise a bigger context, they may be even easier to answer than the "atomic" ones.

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 v
Initialization: $t' = t$, set $\mathcal{S}$ contains all the maximal chains of t s.t. for each chain $s = \{c_1, \ldots, c_n\}$ of $\mathcal{S}$,
$\forall x,\ 1 \leq x \leq n-1,\ c_x.method == c_{x+1}.method$, and $\nexists\ v \in V\ .\ (c \rightarrow v) \in E$, with $c \in s$
begin
1) **while** $(\mathcal{S} \neq \emptyset)$
2) take a chain $s \in \mathcal{S}$
3) $\mathcal{S} = \mathcal{S} \backslash \{s\}$
4) $t' = collapseChain(t', s)$
end while
end
return t'

that are also processed. Of course, the size of the chains already processed is not taken into account when dividing the successive (sub)chains (because they will be already pruned during a debugging session).

If a chain is very long, it can be cut in several subchains to be projected and thus better balance the EF. In order to cut chains we use the function *cutChain*:

function *cutChain*(chain $\{c_1, ..., c_n\}$, int i, int j)
if $i > 2$ **then** $s_{ini} = \{c_1, \ldots, c_{i-1}\}$ **else** $s_{ini} = \emptyset$ **end if**
if $n - j > 1$ **then** $s_{end} = \{c_{j+1}, \ldots, c_n\}$ **else** $s_{end} = \emptyset$ **end if**
return (s_{ini}, s_{end})

This function removes from a chain a subchain delimited by indices i and j. As a result, depending on the indices, it can produce two subchains that are located before and after the subchain. Note that when the initial index i is 2, there is only one node remaining before the subchain, and thus, because it is not a chain, $\emptyset$ is returned. The same happens on the right.

The algorithm finishes when no more chains can be projected. Trivially, because the number of chains and their length are finite, termination is ensured. In addition, 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 final context of the method execution is already calculated, thus all the children of this node are also completed). Note that this means that the algorithm is applied bottom-up to the nodes of the EF. Hence, when balancing a node, all the descendants of the node have been already balanced. This also means that modern debuggers that are able to debug programs with uncompleted ETs [7] can also use the technique, because the ET can be balanced while being computed.

4 Correctness

Our technique for balancing EFs is based on the transformations presented in the previous section. We present in this section the theoretical results about soundness and completeness, whose proofs are available in Appendix A.

Theorem 1 (Completeness and soundness of EFs). *Given an EF with a wrong root, it contains a buggy node which is associated with a buggy method.*

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
1) $t' = shrink(t)$
2) $children = \{v \in V' \mid (root \rightarrow v) \in E'\}$
3) $\mathcal{S} = \{s \mid s$ is a maximal chain in $children\}$
4) $rootweight = root.weight$
5) $weight = rootweight/2$
6) **while** $(\mathcal{S} \neq \emptyset)$
7) $child = c \in children$ such that $\nexists c' \in children, c \neq c' \wedge c'.weight > c.weight$
8) $distance = |weight - child.weight|$
9) **if** $(child.weight \geq weight$ or $\nexists s, i, j$ s.t. $s = \{c_1, \ldots, c_n\} \in S$ and $(|W - weight| < distance)$ with $W = \sum_{x=i}^{j} c_x.weight)$
10) **then** $children = children \backslash \{child\}$
11) $rootweight = rootweight - child.weight$
12) $weight = rootweight/2$
13) **if** $(\exists s \in \mathcal{S}$ such that $s = \{c_1, \ldots, c_n\}$ and $child = c_i, 1 \leq i \leq n)$
14) **then** $(s_{ini}, s_{end}) = cutChain(s, i, i)$
15) $\mathcal{S} = (\mathcal{S} \backslash \{s\}) \cup s_{ini} \cup s_{end}$
end if
else
16) find an s, i, j such that $s = \{c_1, \ldots, c_n\} \in S$ and $\sum_{x=i}^{j} c_x.weight$ is as close as possible to $weight$
17) $s' = \{c_i, \ldots, c_j\}$
18) $(s_{ini}, s_{end}) = cutChain(s, i, j)$
19) $\mathcal{S} = (\mathcal{S} \backslash \{s\}) \cup s_{ini} \cup s_{end}$
20) $t' = projectChain(t', s')$
21) **for each** $c \in s'$
22) $rootweight = rootweight - c.weight$
end for each
23) $children = (children \backslash s')$
24) $weight = rootweight/2$
end if
end while
end
return $t' = (V', E')$

Completeness and soundness are kept after our transformations. In particular, an EF with a buggy node still has a buggy node after any number of collapses or projections.

Theorem 2 (Chain Collapse Correctness). *Let $t = (V, E)$ and $t' = (V', E')$ be two EFs, being the root of t wrong, and let $C \subset V$ be a chain such that all nodes in the chain are leaves and they have the same associated method. Given $t' =$* `collapseChain(t,C)`,

1. *t' contains a buggy node.*
2. *Every buggy node in t' is associated with a buggy method.*

Theorem 3 (Chain Projection Correctness). *Let $t = (V, E)$ and $t' = (V', E')$ be two EFs, and let $C \subset V$ be a chain such that $t' =$* `projectChain(t,C)`.

1. *All buggy nodes in t are also buggy nodes in t'.*
2. *Every buggy node in t' is associated with a buggy method.*

We also provide in this section an interesting result related to the projection of chains. This result is related to the incompleteness of the technique when it is used intra-session (i.e., in a single debugging session trying to find one particular bug). Concretely, the following result does not hold: *A buggy node can be found in an EF if and only if it can be found in its balanced version.*

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

However, AD can find bugs by a 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. The opposite is also 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 EFs in Figure 5. 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 (expressed with triples of the form: initial context, method, final context) then grey nodes are wrong and white nodes are right:

$$\begin{array}{lll} x=1\ g()\ x=4 & x=4\ g()\ x=4 & x=1\ f()\ x=2 \\ x=3\ h()\ x=3 & x=4\ h()\ x=4 & \end{array}$$

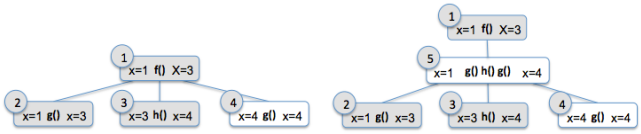


Fig. 5. New buggy nodes revealed.

Note 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 a 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 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.

5 Implementation

We have implemented the technique presented in this paper and integrated it into an algorithmic debugger for Java.The implementation allows the programmer to activate the transformations of the technique and to parameterize them in order to adjust the size of the projected/collapsed chains. It has been tested with a collection of small to large programs including real applications (e.g., an interpreter, a compiler, an XSLT processor, etc.) producing good results, as summarized in Table 1. All the information related

Benchmark	ET nodes	Prj./Col.	Prj./Col. nodes	Bal. time	Quest.	Quest. bal.	%
NumReader	12 nodes	0/0	0/0 nodes	0 msec	6,46	6,46	0,00 %
Orderings	72 nodes	2/14	5/45 nodes	0 msec	11,47	8,89	22,46 %
Factoricer	62 nodes	7/0	17/0 nodes	0 msec	13,89	7,90	43,09 %
Sedgewick	41 nodes	3/8	7/24 nodes	0 msec	18,79	7,52	59,95 %
Clasifier	30 nodes	4/7	10/20 nodes	0 msec	15,52	6,48	58,21 %
LegendGame	93 nodes	12/20	28/40 nodes	0 msec	16,00	9,70	39,36 %
Cues	19 nodes	3/1	8/2 nodes	0 msec	10,40	8,20	21,15 %
Romanic	123 nodes	20/0	40/0 nodes	0 msec	25,06	16,63	33,66 %
FibRecursive	6724 nodes	19/1290	70/2593 nodes	344 msec	38,29	21,47	43,92 %
Risk	70 nodes	7/8	19/43 nodes	0 msec	30,69	10,28	66,50 %
FactTrans	198 nodes	5/0	12/0 nodes	0 msec	18,96	14,25	24,88 %
RndQuicksort	88 nodes	3/3	9/0 nodes	0 msec	12,88	10,40	19,20 %
BinaryArrays	132 nodes	7/0	18/0 nodes	0 msec	15,56	10,58	32,03 %
FibFactAna	380 nodes	3/29	9/58 nodes	0 msec	30,13	29,15	3,27 %
NewtonPol	46 nodes	1/3	2/40 nodes	0 msec	23,09	4,77	79,35 %
RegresionTest	18 nodes	1/0	3/0 nodes	0 msec	6,84	6,26	8,46 %
BoubleFibArrays	214 nodes	0/40	0/83 nodes	0 msec	12,42	12,01	3,33 %
ComplexNumbers	68 nodes	17/9	37/18 nodes	16 msec	20,62	10,20	50,53 %
StatsMeanFib	104 nodes	3/20	6/56 nodes	0 msec	12,33	11,00	10,81 %
Integral	25 nodes	0/2	0/22 nodes	0 msec	8,38	3,38	59,63 %
TestMath	51 nodes	1/2	2/5 nodes	0 msec	12,77	11,65	8,73 %
TestMath2	267 nodes	7/13	16/52 nodes	31 msec	66,47	58,33	12,24 %
Figures	116 nodes	8/3	16/6 nodes	0 msec	13,78	12,17	11,66 %
FactCalc	105 nodes	3/11	8/32 nodes	0 msec	19,81	12,64	36,19 %
SpaceLimits	127 nodes	38/0	76/0 nodes	0 msec	40,85	29,16	28,61 %
Argparser	129 nodes	31/9	70/37 nodes	16 msec	20,78	12,71	38,85 %
Cglib	1216 nodes	67/39	166/84 nodes	620 msec	80,41	65,01	19,15 %
Javassist	1357 nodes	10/8	28/24 nodes	4.745 msec	79,52	77,50	2,54 %
Kxml2	1172 nodes	260/21	695/42 nodes	452 msec	79,61	28,21	64,56 %
HTMLcleaner	6047 nodes	394/90	1001/223 nodes	8.266 msec	169.49	138,85	18,08 %
Jtestcase	4151 nodes	299/27	776/54 nodes	1.328 msec	85,05	80,52	5,32 %

Table 1. Benchmark results.

to the experiments, the source code of the tool, the benchmarks, and other materials 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, `cglib` was tested 1.216 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; column `Prj./Col.` shows the number of projected/collapsed nodes inserted into the EF; column `Prj./Col. nodes` shows the number of nodes that were projected and collapsed by the debugger; column `Bal. time` shows the time needed by the debugger to balance the whole EF; column `Quest.` shows the average number of questions done by the debugger before finding the bug in the original ET; column `Quest. 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 balancing technique. Clearly, the balancing technique has an important impact in the reduction of questions with a mean reduction of 30% using top-down.

Essentially, our debugger produces the EF associated with any method execution specified by the user (by default `main`) and transforms it by collapsing and projecting nodes using Algorithm 4. Finally, it is explored with standard strategies to find a bug. If we observe again Algorithms 1, 2, and 3, a moment of thought should convince the reader that their cost is linear with the branching factor of the EF. In contrast, the

cost of Algorithm 4 is quadratic with the branching factor of the EF. On the practical side, our experiments reveal that the average cost of a single collapse (considering the 1.675 collapses) is 0,77 msec, and the average cost of a single projection (considering the 1.235 projections) is 17,32 msec. Finally, the average cost for balancing an EF is 2.818,25 msec.

Our algorithm is very conservative because it only collapses or projects nodes that belong to a chain. Our first experiments showed that if we do not apply any restriction in the use of chains, the technique produces EFs that are much more balanced. Repeating the experiments in this way (considering all 23.257 experiments) produced a query reduction of 42%. However, this reduction comes with a cost: the complexity of the questions may be increased. Therefore, we only apply the transformations when the question produced is not complicated (i.e., when it is generated by a chain). This has produced good results. In the case that the question of a collapsed/projected node was still hard to answer, our tool gives the possibility of answering "I don't know", thus skipping the current question and continuing the debugging process with the other questions (e.g., with the children). This means 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 is conservative and cannot cause the debugging session to stop.

6 Related Work

We are not aware of other approaches for balancing the structure of the ET. However, besides our approach, there exist other transformations devoted to reducing the size of the ET, and thus the number of questions performed. Our implementation allows us to balance an already generated ET, and it also allows us to automatically generate the balanced ET. This can be done by collapsing or projecting nodes during their generation. However, conceptually, our technique is a post-ET generation transformation.

The most similar approach is the tree compression technique introduced by Dave and Chitil [5]. This approach is also a conservative approach that transforms an ET into an equivalent (smaller) ET where the same bugs can be detected. The objective of this technique is essentially different: it tries to reduce the size of the ET by removing redundant nodes, and it is only applicable to recursive calls. A similar approach to tree compression is declarative source debugging [4], that instead of modifying the tree implements an algorithm to prevent the debugger from selecting questions related to nodes generated by recursive calls.

Another approach which is related to ours was presented in [11], where a transformation for list comprehensions of functional programs was introduced. In this case, it is a source code (rather than an ET) transformation to translate list comprehensions into equivalent functions that implement the iteration. The ET produced can be further transformed to remove the internal nodes of the ET reducing the size of the final ET as in the tree compression technique. Both techniques are orthogonal to the balancing of the ET, thus they both can be applied before balancing.

7 Conclusions

This work presents a new technique that allows us to automatically balance standard ETs. This technique has been implemented, and experiments with real applications confirm that it has a positive impact on the performance of AD.

From a theoretical point of view, two important results have been proved. The projection and the collapse of nodes do not prevent 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 discover bugs undetectable with the original ET and can also change the order in which bugs are found.

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 so that search strategies can ignore them.

References

1. D. Binks. *Declarative Debugging in Gödel*. PhD thesis, University of Bristol, 1995.
2. 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 2005*, pp. 8–13. ACM Press, 2005.
3. R. Caballero, C. Hermanns, and H. Kuchen. Algorithmic debugging of Java programs. In Francisco Javier López-Fraguas, editor, *Proc. of the 15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid, Spain*, *ENTCS* 177, pp. 75–89. Elsevier, 2007.
4. Miguel Calejo. *A Framework for Declarative Prolog Debugging*. PhD thesis, New University of Lisbon, 1992.
5. T. Davie and O. Chitil. Hat-delta: One Right Does Make a Wrong. In *Seventh Symposium on Trends in Functional Programming, TFP 2006*, April 2006.
6. V. Hirunkitti and C. J. Hogger. A Generalised Query Minimisation for Program Debugging. In *Proc. of International Workshop of Automated and Algorithmic Debugging, AADEBUG 1993*, *LNCS* 749, pp. 153–170. Springer, 1993.
7. D. Insa and J. Silva. Debugging with Incomplete and Dynamically Generated Execution Trees. In María Alpuente, editor, *Proc. of the 20th International Symposium on Logic-based Program Synthesis and Transformation, LOPSTR 2010*, *LNCS* 6564. Springer, 2011.
8. 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 1999*, pp. 95–104. ACM Press, 1999.
9. I. MacLarty. *Practical Declarative Debugging of Mercury Programs*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, 2005.
10. M. Maeji and T. Kanamori. Top-Down Zooming Diagnosis of Logic Programs. Technical Report TR-290, Japan, 1987.
11. H. Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping, Sweden, May 1998.

12. H. Nilsson and P. Fritzson. Algorithmic Debugging for Lazy Functional Languages. *Journal of Functional Programming*, 4(3):337–370, 1994.
13. E.Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.
14. Josep Silva. A Survey on Algorithmic Debugging Strategies. *Advances in Engineering Software*, 42(11):976–991, 2011.

Note for the reviewers: The following appendices have been only included to ease the reviewing process, and they will not be part of the final paper. In case of acceptance, these appendices will be published as a technical report so that the interested reader will have public access to them.

A Proofs of technical results

Lemma 1 (Buggy method). *Given an ET $t = (V,E)$, and a buggy node $v \in V$ in t with $v = (b,m,e)$, then m contains a bug.*

Proof. Because v is buggy, then the method execution (b,m,e) is wrong, thus $(b,m,e) \notin I$ and $I \not\models (b,m,e)$. Moreover, by Definition 3, we have a child of v for each call to a method done from the definition of m. But we know by Definition 5 that for all child v' of v, $v' \in I$ or $I \models v'$ Hence, m contains a bug.

Proposition 1 *Let t be a EF with a wrong root. Then t contains a buggy node.*

Proof. We prove the claim by induction on the size of t. (Base case) t only contains one node b. Then b is buggy, because it is wrong and it has no children. (Induction hypothesis) t contains i nodes and at least one of them is buggy. (Inductive case) t contains $i+1$ nodes. In this case we have a tree of i nodes that, by the induction hypothesis, does contain a buggy node b plus one extra node n. If n is not the child of b, then b is buggy. If n is the child of b, then either n is correct, and thus b is buggy; or n is wrong and hence, it is buggy because it has no children.

Lemma 2 (Soundness of projections and collapses). *Given a collapsed or projected node $v = (b,m_1;\ldots;m_n,e)$ in an EF. If v is buggy, then it contains a buggy method.*

Proof. We have two possibilities: (1) v is a collapsed node. In this case v has not children, and because v is wrong, $(b,m_1;\ldots;m_n,e) \notin I$; therefore, trivially, at least one method m_i, $1 \leq i \leq n$, is buggy. (2) v is a projected node. This case is impossible because a projected node cannot be buggy. The reason is that if all the children of v are correct, then v is correct by Definition 4 using the inference rule Tr. Otherwise, at least one child is wrong, but then, v cannot be buggy by Definition 5.

Theorem 1 (Completeness and soundness of EFs). *Given an EF with a wrong root, it contains a buggy node which is associated with a buggy method.*

Proof. The first point is proved by Proposition 1, while the second one is proved by Lemmas 1 and 2.

Theorem 2 (Chain Collapse Correctness). *Let $t = (V,E)$ and $t' = (V',E')$ be two EFs, being the root of t wrong, and let $C \subset V$ be a chain such that all nodes in the chain are leaves and they have the same associated method. Given $t' =$* `collapseChain(t,C)`,

1. *t' contains a buggy node.*
2. *Every buggy node in t' is associated with a buggy method.*

Proof. For the first item, only leaf nodes can be collapsed, therefore, the root node could only be collapsed if it is the only node of t. However, even in this case we have that $\nexists v \in V$ such that $(v \to r) \in E$ being r the root of t. Therefore, according to Definition 7, r is not a chain and thus it cannot be collapsed. Hence, the root of t' is the same as the root of t, and thus t' contains a buggy node by Proposition 1.

Now, we prove that any buggy node of t' is associated with a buggy method. Let $v \in V$ be the parent node of the chain C, and let $w \in V'$ be the collapsed node of C. We consider three cases:

- $u \in V', v \neq u \neq w$ is buggy. In this case the collapse does not influence the buggy node u and thus the claim follows by Lemma 1.
- v is buggy in t'. This case is trivial, because v is wrong and w is correct by Definition 5. Therefore, the new node w can be inferred with the rule Tr and thus the method in v is wrong according to Lemma 1.
  This case is particularly interesting because it reveals a phenomenon: node v is not changed by the transformation and thus it belongs to both trees t and t'. However, it could be possible that v is not buggy in t but it is buggy in t'. This happens because w has somehow hidden some error in the chain—some wrong intermediate result that was visible in the chain is now hidden because only the initial and final contexts are shown—, revealing a new bug located in v.
- w is buggy in t'. Then, either the result or the final context of w are wrong. Hence, since both the result and the final context are produced by the nodes in C, we know that at least one node $c \in C$ is also wrong. Because c is a leaf and it is wrong, then it is a buggy node in t and it is associated with a buggy method m by Lemma 1. According to Algorithm 1 w is associated with a method execution $(b, m_0 \ldots m \ldots m_n, e)$, and thus it is associated with a buggy method.

Theorem 3 (Chain Projection Correctness). *Let $t = (V, E)$ and $t' = (V', E')$ be two EFs, and let $C \subset V$ be a chain such that $t' =$* `projectChain(t,C)`.

1. *All buggy nodes in t are also buggy nodes in t'.*
2. *Every buggy node in t' is associated with a buggy method.*

Proof. Let $v \in V$ be the parent node of the chain C, and let $w \in V'$ be the projected node of C. We consider an arbitrary buggy node $u \in V$ and show that it is also buggy in t'. Five cases are possible:

- u is the root of t. It is easy to see that t' has the same root as t, since the node added by `projectChain` requires a parent node in the tree (i.e., the root cannot be projected). Thus, if the root of t is buggy, then the root of t' is also buggy.
- $u \neq v, u \neq w$ and $u \neq c \in C$. In this case the projection does not influence the buggy node u nor its children and thus u is also buggy in t'.
- $u = v$. This means that u is the parent of the chain C. If it is buggy, then by Definition 5 all nodes in C are correct. Then, as shown in the proof of Lemma 2, w is correct. Hence, u is also buggy in t'.
  In the general case, v will be buggy in t, and also in t' ($\forall c \in C$, c will be correct; and thus w is also correct). However, it could be possible that v is correct in t, two nodes $c_1, c_2 \in C$ were wrong, and their combined (wrong) effects produced a correct result. In that case, both errors would be hidden in the projection node (but of course they would remain in c_1 and c_2). As a result, a new buggy node (v) not present in t would appear in t'.

- $u = c \in C$. This means that c is wrong and all its children correct. Since the children of c have not been modified by `projectChain`, c was buggy in t and it is also buggy in t'.

In all cases, the buggy node is associated with a buggy method by Lemmas 1 and 2.

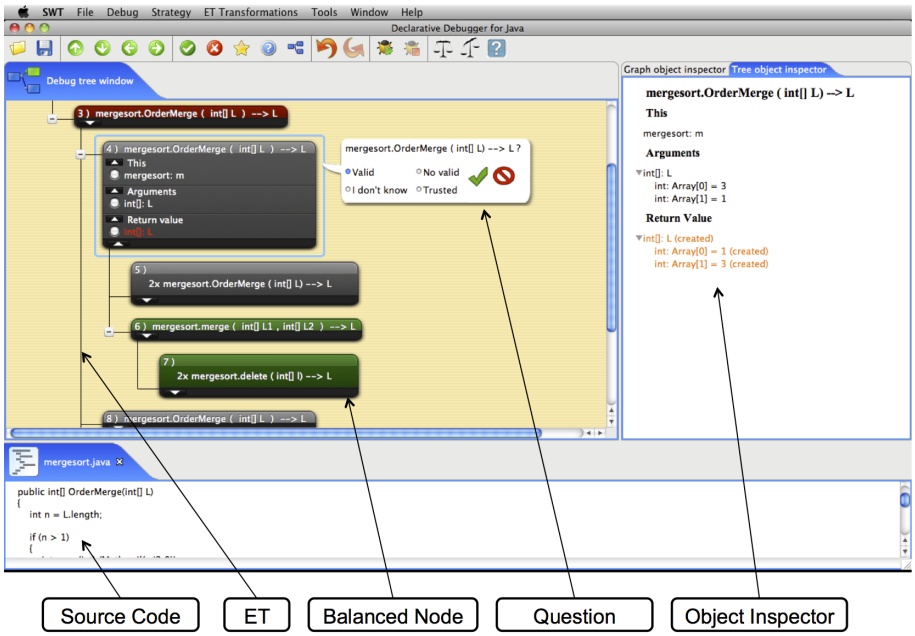


Fig. 6. Algorithmic Debugging session of Mergesort.

B Case of Study: Mergesort

In this appendix we show a real application of our algorithm using our implementation. In particular, we show the result of balancing a Mergesort's EF. The following version of Mergesort algorithm was initially extracted from Wikipedia (`http://es.wikipedia.org/wiki/Ordenamiento_por_mezcla#Java`) and then modified to include one bug in function `merge`: it does not update positions appropriately.

Method `main` declares an object of class `mergesort`, initializes the variable `x` with the list `{3,1,2}`, and sorts `x` with `OrderMerge`. Finally, the result is stored in variable `y`, and it is printed:

```
import java.util.Random;

public class mergesort{
  public static void main(String[] args) {
    mergesort m = new mergesort();
    int[] x = {3,1,2};
    int[] y;
    y=m.OrderMerge(x);
    for(int i = 0; i < y.length; i++){
      System.out.println(y[i]);
    }
  }
```

Method `OrderMerge` implements the mergesort algorithm: First, it splits the list given as argument; then, it sorts each fragment and merges them together to obtain the final result:

```
public int[] OrderMerge(int[] L) {
  int n = L.length;
  if (n > 1){
    int m = (int) (Math.ceil(n/2.0));
    int [] L1 = new int[m];
    int [] L2 = new int[n-m];
    for (int i = 0; i < m; i++){
      L1[i] = L[i];
    }
    for (int i = m; i < n; i++){
      L2[i-m] = L[i];
    }
    L = merge(OrderMerge(L1), OrderMerge(L2));
  }
  return L;
}
```

Method `merge` introduces the elements of the (sorted) list received as arguments in an ordered fashion in a new list, which is returned as the result. However, we have introduced an error: when the element in the first array is smaller than the one in the second array, it is introduced in the new array, *but the position is not updated*. Thus, the next element introduced overwrites its value:

```
public int[] merge(int[] L1, int[] L2){
  int[] L = new int[L1.length+L2.length];
  int i = 0;
  while ((L1.length != 0) && (L2.length != 0)) {
    if (L1[0] < L2[0]){
      L[i] = L1[0];
      // The previous line is erroneous,
      // it should be:
      //L[i++] = L1[0];
      L1 = delete(L1);
      if (L1.length == 0){
        while (L2.length != 0) {
          L[i++] = L2[0];
          L2 = delete(L2);
        }
      }
    }
    else{
      L[i++] = L2[0];
      L2 = delete(L2);
      if (L2.length == 0) {
```

```
          while (L1.length != 0) {
            L[i++] = L1[0];
            L1 = delete(L1);
          }
        }
      }
    }
    return L;
  }
}
```

Finally, method `delete` removes the first element of the list received as argument:

```
  public int[] delete(int [] l){
    int [] L = new int[l.length-1];
    for(int i = 1; i < l.length; i++){
      L[i-1] = l[i];
    }
    return L;
  }
```

If we execute the main method of the class, we observe that the initial list, `{3,1,2}`, is sorted to `{2,3,0}`. Since the expected result was `{1,2,3}`, there must be a bug in the code. At this point we can use a conventional debugger and place breakpoints at some parts of the code to try to figure out if the variables are updated correctly at those points. Then add new breakpoints and so on.

An alternative is to use our debugger. When we load *mergesort.java* in the debugger we see the information shown in Figure 6. In the main panel we can observe a part of the EF associated with Mergesort. The debugger uses colors to distinguish between those nodes that are wrong (in red), and those nodes that are correct (in green). The validity of grey nodes is unknown, and thus they are suspicious of being buggy. Of course, the programmer can inspect any node at any moment, and she can also direct the debugging session manually as it happens with breakpoints. But in general, the programmer allows the debugger to automatically direct the search for the bug, because it always selects the node that better divides the suspicious area.

In the figure, the debugger has automatically selected a node related with the method `OrderMerge`, and it has popped up the question associated with this node:

```
mergesort.OrderMerge({3,1})={1,3}?
```

Clearly the answer to the question is `Valid` because the elements of the array have been ordered correctly. Note also that it is not even necessary to look at the implementation of the method to answer the question; i.e., to answer we focus on the objective of the method (we only have to know that this method orders an array), and we do not need to know the operational details (it does not matter how it was implemented).

The information associated to each node is displayed in the panel at the right (the object inspector). It is similar to the "variable watch view" used in traditional debuggers: It allows us to inspect all objects in the scope of this method. Note that the information

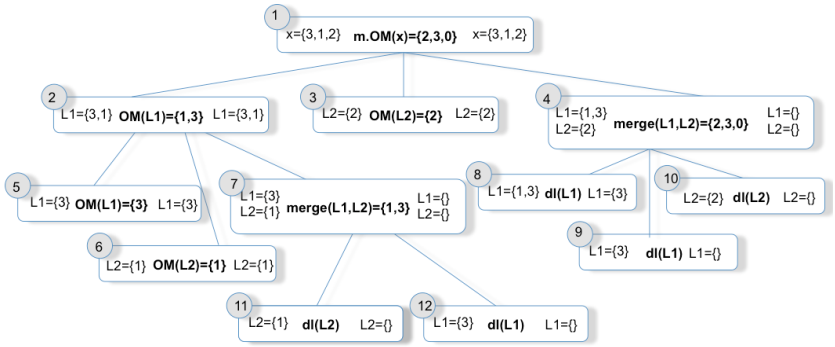


Fig. 7. EF associated with Mergesort.

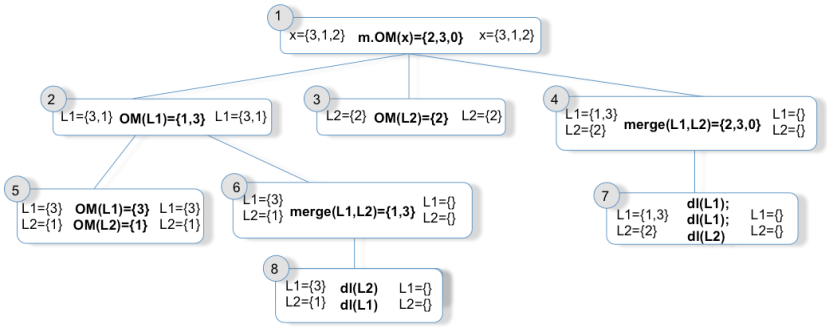


Fig. 8. Balanced EF associated with Mergesort.

is classified so that we can see the arguments and the result separately; and creation, deletion or changes in any object are highlighted with colors.

The EF associated with Mergesort is displayed in Figure 7, where `OM` abbreviates `OrderMerge`. An standard algorithmic debugger would traverse this tree as follows:

```
Starting Debugging Session...

(1) NO  (2) YES  (3) YES (4) NO (8) YES (9) YES (10) YES

Bug found in method:
merge(int[] L1, int[] L2) of class Mergesort
```

Fortunately, using our balancing algorithm, the debugger can transform the EF in Figure 7 to produce the EF of Figure 8. Nodes (5) and (6); (8), (9) and (10); and, (11) and (12), have been collapsed. The tool automatically detected that they form chains and collapsed them to form a single question. For instance, the new collapsed node (7) in Figure 8 contains the following question: *"Having the lists L1={1,3} and L2={2}, if we delete two elements from list L1, and one element from list L2; do we obtain the lists L1={} and L2={}?"*

Although larger trees would not fit in this paper, we can already see the pattern of collapse for this example: merging two lists of sizes $n1$ and $n2$, that in general produces $n1+n2+1$ nodes of the debugging tree (one node for `merge` and $n1+n2$ for `delete`), only produces two nodes in the collapsed tree. Note that this improvement not only reduces the number of questions asked to the user, it also enhances the computation of the debugging tree, since the number of nodes is much smaller.

With the balanced EF, the traversal is:

```
Starting Debugging Session...
(1) NO  (2) YES  (3) YES (4) NO (7) YES
Bug found in method:
merge(int[] L1, int[] L2) of class Mergesort
```