A Generalized Model for Algorithmic Debugging

David Insa and Josep Silva

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

Abstract. Algorithmic debugging is a semi-automatic debugging technique that is present in practically all mature programming languages. In this paper we claim that the state of the practice in algorithmic debugging is a step forward compared with the state of the theory. In particular, we argue that novel techniques for algorithmic debugging cannot be supported by the standard internal data structures, such as the Execution Tree (ET), used in this technique, and hence a generalization of the standard definitions and algorithms is needed. We identify two specific problems of the standard formulation and implementations of algorithmic debugging, and we propose a reformulation to solve both problems. The reformulation has been done in a paradigm-independent manner to make it useful and reusable in different programming languages.

1 Introduction

One of the most important debugging techniques is *Algorithmic Debugging* (AD) [27]. This technique has experienced a significant advance in the last decade. Concretely, new techniques have been proposed to improve performance [9, 15], to improve scalability [11], to improve interaction with the user [6], and to improve GUIs [12, 13]. The maturity of these techniques has eventually led to the integration of algorithmic debuggers into sophisticated programming environments. Two interesting cases are [12] and [11], which combine AD with the standard debugging perspective of Eclipse [1]. The main advantage of AD is its high level of abstraction. It is even possible to debug a program without looking at the source code.

Example 1. Let us assume the existence of a buggy Java code composed of three methods: `int add(int x, int y)` sums its two arguments, `boolean isEven(int x)` returns `true` if its only argument is even, or `false` otherwise; and, `int sumNumbers(int[] array, String eo)` takes an array of integers and sums the elements that are even or odd depending on the value of the second argument. Therefore, with the following method invocation:

```
int[] array = { 1, 2, 3 };
int sum = sumNumbers(array, "odd");
```

the result should be `4`. Nevertheless, due to a bug in the code, the result is `3`.

Thanks to AD, with only this information (without knowing anything about the source code) we can identify the buggy method. For instance, if we debug this program

with the Hybrid Debugger for Java (HDJ),[1] we obtain the following debugging session (questions are generated by HDJ, and answers are provided by the user):

```
Starting Debugging Session:
      (1) sumNumbers({1,2,3},"odd")=3? No
      (2) isEven(2)=true? Yes
      (3) isEven(3)=false? Yes
      (4) add(0,3)=3? Yes
Bug found in method "sumNumbers" with the call "sumNumbers({1,2,3},"odd")".
```

Hence, an AD session is just a dialogue where the debugger asks questions and the user answers them. The Java code associated with this debugging session is depicted in Figure 1.

```
(1) int add(int x, int y) {
(2)     return x + y;
(3) }

(4) boolean isEven(int x) {
(5)     return (x % 2) == 0;
(6) }
```

```
(7)  int sumNumbers(int[] array, String eo) {
(8)      int sum = 0;
(9)      for (int i = 1; i < array.length; i++) {
(10)         if (eo.equals("even") && isEven(array[i]))
(11)             sum = add(sum, array[i]);
(12)         if (eo.equals("odd") && !isEven(array[i]))
(13)             sum = add(sum, array[i]);
(14)     }
(15)     return sum;
(16) }
```

Fig. 1. Java program to sum the even or odd numbers of an array

What the debugger internally does is to generate a data structure that represents the execution of the program. This data structure, often called Execution Tree (ET), is depicted in Figure 2. The ET has a node for each method invocation.[2] Each node normally contains a reference to the method that is being executed, the value of its arguments, the old and new values of the variables that may be changed within the execution, and its returned value. The debugger just traverses the ET asking the user about the validity of the nodes (i.e., nodes are marked as correct or wrong) until a buggy node is found. A node is buggy when it is wrong, and all of its children (if any) are correct.

The main properties of AD are the following:

Theorem 1 (Correctness of AD [23]). *Given an ET with a buggy node n, the method associated with n contains a bug.*

Theorem 2 (Completeness of AD [27]). *Given an ET with a bug symptom (i.e., the root is wrong), provided that all the questions generated by the debugger are correctly answered, then, a bug will eventually be found.*

---

[1] http://www.dsic.upv.es/~jsilva/HDJ/

[2] In the ET, nodes represent computations. Hence, depending on the underlying paradigm, they can represent methods, functions, procedures, clauses, etc. Our discussions in this paper can be applied to both the imperative and the declarative paradigms, but, for the sake of concreteness, we will focuss the discussion on the imperative paradigm and our examples on Java.

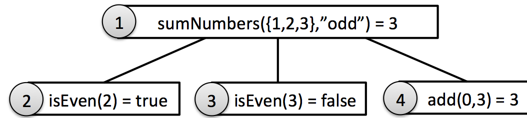

Fig. 2. ET generated for the program in Figure 1

1.1 Contributions of this work

In this work, we propose a new redefinition of AD in such a way that: (i) It is paradigm- and language-independent, and thus it is reusable by other researchers. (ii) It is a conservative generalization of the traditional formulation of AD, in such a way that many previous AD techniques are a particular case of this new formulation. (iii) It is formulated in a way that definitions of the data structures, properties, strategies, and algorithms are specified separately, so that they can be reused and/or concretized in a particular case. (iv) It states that the output of an algorithmic debugger should contain dynamic information (i.e., it should not include non-executed code). And, (v) it allows the debugger to ask questions about a code inside a method (and not only about the whole method).

2 Some problems identified in current algorithmic debuggers

We have been actively working in the area of AD for the last 10 years. This paper somehow summarizes and criticizes our own work to make a step forward. We claim that almost all current algorithmic debuggers—at least all that we know, including the most extended, which we compared in [7], and including our own implementations—have fundamental problems that were somehow inherited from the original formulation of AD [27].

In particular, we claim that the original formulation of AD, and most of the later definitions and implementations are obsolete with respect to the last advances on the practical side of AD. For instance, two important problems of the standard definitions of AD are the granularity and the static nature of the found errors (AD reports a whole routine as buggy). We can illustrate these problems observing again the debugging session of Example 1: The whole method `sumNumbers` is pointed out as buggy. This is very imprecise specially if `sumNumbers` were a method with a lot of code. However, AD researchers and developers are used to this behavior, and they would argue that this is the normal output of any algorithmic debugger. However, from an engineering perspective, this is quite surprising because the analysis performed by the debugger is by definition dynamic (in fact, the whole program is actually executed). Hence, the debugger should know that line 11 of Figure 1 is never executed, and thus it should not be reported as buggy. This leads us to our first proposition: The information reported by an algorithmic debugger should be dynamic instead of static. That is, the output of the algorithmic debugger should be the part of the method that has been actually executed to produce the bug, instead of the whole method.

We think that this problem comes from the first implementations of AD and it has been inherited in latter theoretical and practical developments. In fact, if we execute

this program with the debuggers: Buddha [26], DDT [4], Freja [21], Hat-Delta (and its predecessor Hat-Detect) [8], B.i.O. [3], Mercury's Algorithmic Debugger [19], Münster Curry Debugger [18], Nude [20], DDJ [13], and HDJ [11], they all would output the whole `sumNumbers` as buggy together with a counterexample that produces the bug (the found buggy node). Unfortunately, none of these debuggers make further use of the counterexample. An option would be that the debuggers use dynamic program slicing (to be precise, dynamic chopping) [30] to minimize the code shown as buggy.

Traditionally, AD reports a whole method as buggy. To reduce the granularity of the reported errors, new techniques have appeared (see, e.g., [16, 5]) that allow for debugging inside a method. Unfortunately, the standard definition of ET is not prepared for that. In fact, some of the recent transformations defined for AD do not fit in the traditional definition of the data structures used in this discipline. For instance, the *Tree Balancing* technique presented in [15], or the *zooming* technique presented in [5] cannot be represented with standard AD data structures such as the *Evaluation Dependence Tree* [24].

This lack of a common theoretical framework with standard data structures that are powerful enough as to represent recent developments makes researchers to reinvent the wheel once and again. In particular, we have observed that researchers (including ourselves) have produced local and partial formalizations to define their debuggers for a particular language and/or implementation (see, e.g., [6, 16, 15]). These theoretical developments are hardly reusable in other languages, and thus, they only serve as a formal description of their system, or as a means to prove results.

3 Related Work

Algorithmic debugging has been applied to all mature languages. All current implementations use a sort of ET to represent computations. Even in those lazy implementations of AD where the execution of the front-end and the back-end is interleaved (see, e.g., [22]), the construction of the ET is needed before the program can be debugged. Along the years each paradigm has adopted a well-defined and studied data structure to represent the ET.

3.1 A Little Bit of History

Algorithmic debugging started in the seminal work by Shapiro with the notion of contradiction backtracking using "crucial experiments" within Popper's philosophical dictum of conjectures and refutations [28]. Hence, the first notion of ET appeared in the context of the logic paradigm. Shapiro used refutation trees as ETs. Later implementations of AD in the logic paradigm such as NU-Prolog [31, 2] also used refutation trees.

In the context of the functional paradigm, the data structure used was proposed by Henrik Nilsson and Jan Sparud: The Evaluation Dependence Tree (EDT). They first proposed this data structure as a record of the execution [24], and then, as an appropriate ET for AD [25]. The EDT is particularly useful to represent lazy computations by hiding non-computed terms. In fact, the own EDT can be computed lazily as in [22]. The most

successful implementations of AD for the functional paradigm are based on the EDT. Notable cases are the Freja [21], Hat-Delta [8], and Buddha [26] debuggers.

In multi-paradigm languages such as Mercury, TOY, or Curry, the ET is also represented with either a proof tree or an EDT. Examples of debuggers for these languages are the Mercury Debugger [19], the Münster Curry Debugger [18], DDT [4], and B.i.O [3].

In the imperative paradigm, a redefinition of the EDT was used. It has been often called *Execution Tree* [10, 13], but, conceptually, it is equivalent to the EDT, and it can be seen as a dynamic version of the *Call Graph* where every single call generates a different node in the graph, and thus no cycles are possible (i.e., it is a tree).

3.2 Modern Implementations

All the debuggers mentioned in the previous sections are somehow "standard" in the sense that they are based on the standard definition of the ET (either the refutation trees or the EDT). However, in the last 5 years, there has been a new trend in AD tools: Researchers have implemented new techniques that go beyond the standard definition of the ET. Contrarily to the previously described tools, modern algorithmic debuggers are not standalone tools. They are plugins that can be integrated as part of an IDE. Examples of these debuggers are JHyde [12] and HDJ [11], being both of them part of Eclipse. Precisely because they are integrated into a development environment, they have direct access to dynamic information—they can even manipulate the JVM at runtime—that can be used to enhance the debugging sessions. In particular, the following techniques go beyond the standard ET: (i) *Tree compression* hides nodes of the ET (it breaks the standard parent-child relation in the ET). (ii) *Tree balancing* introduces new artificial nodes in the ET (it breaks the standard definition of ET node). (iii) *Loop expansion* and (iv) *ET zooming* decompose ET nodes (they break the standard definition of ET node). We are not aware of any definition of ET able to represent the previous four techniques.

4 Paradigm-Independent Redefinition of Algorithmic Debugging

Some of our last developments for AD cannot be formalized with the standard AD formulation. In a few cases, we just skipped the formalization of our technique and provided an implementation. In other cases we wanted to prove some properties, and thus we formalized (for one specific language, e.g., Java) the part of the system affected by those properties. Other developments were done for other paradigms, e.g., the functional paradigm, and we also formalized a different part of the system with different data structures. We have observed the same behavior in other researchers, and clearly, this is due to the lack of a standard solution.

We want to provide a definition of AD that is paradigm-independent (i.e., it can be used by either imperative- or declarative- languages). From the best of our knowledge, there does not exist such a formal definition of AD. Hence, in this section we formulate AD in an abstract way. The main generalization of our new formulation is to consider that ET nodes are not necessarily routines as in previous definitions (see, e.g., [24]). Contrarily, we allow ET nodes to contain any piece of code. This permits AD to report

any code as buggy, and not only routines, thus potentially reducing the granularity of the reported errors to single expressions.

In the following, we will only call *Execution Tree* to our new definition, and we will call *Routine Tree* (RT) to the traditional definition (that we also formalize in the next sections). Because our new definition is a conservative generalization, the RT is a particular case of the ET as it can be observed in the UML model of Figure 3.

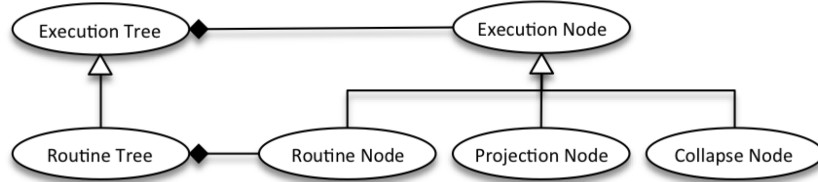

Fig. 3. UML model representing the structure of the execution tree

Observe that an execution node can be specialized depending on the piece of code it represents. In particular, we specialize three kinds of execution nodes named *Routine Node*, *Projection Node*, and *Collapse Node*. They correspond to definitions that already exist in the literature (see [10, 15]), but other kinds of nodes could appear in the future.

4.1 The Execution Tree

In this section we introduce some notation and formalize the notion of Execution Tree used in the rest of the paper. We want to keep the discussion and definitions in this section paradigm-independent. Hence, we consider programs as state transition systems.

Definition 1 (Program). *A* program $P = \{W, I, R, C\}$ *consists of:*

- *W: A set of* states.
- *I: A set of* starting states*, such that $I \subseteq W$.*
- *R: A* transition relation*, such that $R \subseteq W \times W$.*
- *C: A* source code*, composed of a set of statements.*

Definition 2 (Computation). *A* computation *is a maximal sequence of states $s_1, s_2, \ldots$ such that:*

- *s_1 is a starting state, i.e., $s_1 \in I$.*
- *$(s_i, s_{i+1}) \in R$ for all $i \geq 1$ (and $i \leq n-1$, if the sequence is of the finite length n).*

A finite segment $s_i, s_{i+1}, \ldots, s_j$ where $1 \leq i < j \leq n$ is called a subcomputation.

In the source code of a program, we consider statements[3] as the basic execution unit. Therefore, in the following, the source code of a program P is a set of statements

[3] Note the careful use of the word “statement" to refer to either imperative instructions, declarative expressions, etc.

$st_1, st_2, \ldots, st_n$ that produces the computation $s_0, s_1, \ldots, s_m$ for a given starting state s_0. We cannot provide a specific model of computation if we want to be paradigm-independent, thus we do not define the relation between statements and the transition relation R. This is possible (and convenient) thanks to the abstract nature of algorithmic debugging. In particular, algorithmic debugging only needs an initial state, a code, and a final state to identify bugs. No matter how the code makes the transition from the initial state to the final state. The user will decide whether this transition is correct or not.

Because the considered execution unit is the statement, it is possible to identify a bug in a single statement. This contrasts with traditional algorithmic debugging where routines are the execution units, and thus a whole routine is always reported as buggy.

We also use the notion of *code fragment* of a program P, which refers to any subset of statements in the source code C of P that produces a subcomputation $s_i, \ldots, s_j$ with $0 \leq i < j \leq m$. Code fragments often represent functions or loops in a program, but they can also represent blocks, single statements, or even function calls together with the whole called function.

Intuitively, not all the statements in a given code c that produces a computation $\mathcal{C}$ are actually executed. Some parts of the code are not needed to produce the computation (e.g., because they are dead code, because some condition does not hold, etc.). The projection of c modulo $\mathcal{C}$ is a subset of c where the unneeded code in c to produce $\mathcal{C}$ has been removed. Projections are often computed with dynamic slicing [30].

Definition 3 (Code Projection). *Given a code fragment c and a computation $\mathcal{C}_c = s_0, \ldots, s_n$ produced by c from a given initial state s_0, a* projection of c modulo $\mathcal{C}_c$ *is a code fragment that contains the minimum subset of c needed to produce the computation $\mathcal{C}_c$.*

We assume that each state in W is composed of pairs variable-value. The initial and final states, s_i and s_j, describe the effects of a given code fragment c. All three together form a *code behavior*.

Definition 4 (Code Behavior). *Given a code fragment c and a computation $\mathcal{C}_c = s_0, \ldots, s_n$ produced by c from a given initial state s_0, the* code behavior *of $\mathcal{C}_c$ is a triple $(s_0, \mathcal{P}_{\mathcal{C}_c}(c), s_n)$, where $\mathcal{P}_{\mathcal{C}_c}(c)$ is the projection of c modulo $\mathcal{C}_c$.*

Code behavior corresponds to the questions asked by the debugger. These questions are along the lines of *Should the code c with the initial state s_0 produce the final state s_n?*, or *Code c produced s_n from s_0, is that correct?* Many previous definitions of AD (see, e.g., [22, 5]) define the code behavior as the triple (s_0, c, s_n), which corresponds to the execution of a routine c, and usually the debugger only needs to show the call to c instead of showing both the call to c and the own routine c. Definition 4, however, introduces two important novelties:

- It allows c to be any code fragment, and not only a routine.
- It substitutes c by a projection of c modulo $\mathcal{C}_c$, thus the code associated with a code behavior only contains the code actually needed to produce that behavior.

This dynamic notion is much more precise than the usual static notion that considers (the complete code of) a routine.

Definition 5 (Intended Model). *Given a program $P = \{W, I, R, C\}$, an* intended model *$\mathcal{M}$ for P is a set of tuples $(s_i, \mathcal{P}(c), s_j)$ where $s_i, s_j \in W$ and $\mathcal{P}(c)$ is a projection of a code fragment $c \subseteq C$.*

Each tuple of the form $(s_i, \mathcal{P}(c), s_j)$ specifies that the execution of code $\mathcal{P}(c)$ from state s_i leads to state s_j. Intuitively, an intended model of a program contains the set of code behaviors that are correct with respect to what the programmer had in mind when he programmed these codes. It is used as a reference point against which one can compare computations to determine whether they are correct or wrong.
We are now in a position to define the nodes of an execution tree.

Definition 6 (Execution Node). *Let $P = \{W, I, R, C\}$ be a program. Let $\mathcal{C}_c$ be a computation produced by a code fragment $c \subseteq C$. Let $\mathcal{M}$ be an intended model for P. The* execution node *induced by $\mathcal{C}_c$ is a pair ($\mathcal{B}$, $\mathcal{S}$) where:*

1. *$\mathcal{B}$ is the code behavior of $\mathcal{C}_c$, and*
2. *$\mathcal{S}$ is the state of the node, which can be either:*
   - undefined*, or*
   - *the correctness of $\mathcal{B}$ with respect to $\mathcal{M}$:* $\begin{cases} \text{correct} & \text{if } \mathcal{B} \in \mathcal{M} \\ \text{wrong} & \text{if } \mathcal{B} \notin \mathcal{M} \end{cases}$

Observe that an execution node contains (inside $\mathcal{B}$) the source code $\mathcal{P}_{\mathcal{C}_c}(c)$ responsible of the computation it represents. Hence, if this node is eventually declared as buggy, its associated code is uniquely identified. This definition of execution node is general enough as to represent previous nodes that are used in different techniques. For instance, if the code of the node is a function, it can be represented as a *routine node*. Similarly, *projection nodes* and *collapse nodes*, introduced in [15], are special nodes that agglutinate the code of several other nodes. Clearly, they are also particular cases of our general definition.

In order to properly define execution trees, we need to define first a relation between execution nodes that specifies the parent-child relation.

Definition 7 (Execution Nodes Dependency). *Let N be a set of execution nodes. Given an execution node $n_c \in N$ induced by a computation $\mathcal{C}_c$, and an execution node $n_{c'} \in N$ induced by a subcomputation $\mathcal{C}_{c'}$ of $\mathcal{C}_c$, we say that n_c directly depends on $n_{c'}$ (expressed as $n_c \xrightarrow{N} n_{c'}$) if and only if there does not exist an execution node $n_{c''} \in N$ induced by subcomputation $\mathcal{C}_{c''}$ of $\mathcal{C}_c$, such that $\mathcal{C}_{c'}$ is a subcomputation of $\mathcal{C}_{c''}$.*

Observe that this dependency relation is intransitive, which is needed to define the parent-child relation in a tree. Hence, provided that we have three execution nodes, n_1, n_2, n_3, if $n_1 \xrightarrow{N} n_2 \xrightarrow{N} n_3$ then $n_1 \overset{N}{\not\to} n_3$.

Example 2. Given the following program:

```
CODE:
x++; y++; x=x+y;
```

and the initial state `(x=1,y=2)` we can generate the following execution nodes (among others):

```
ET NODES:
          (initial state)          code                  (end state)
---------------------------------------------------------------------
   node 1:    (x=1,y=2)       x++; y++; x=x+y;        (x=5,y=3)
   node 2:    (x=1,y=2)       x++; y++;               (x=2,y=3)
   node 3:    (x=2,y=3)       x=x+y;                  (x=5,y=3)
   node 4:    (x=2,y=2)       y++;                    (x=2,y=3)
```

with N={node 1, node 2, node 3, node 4}
we have $\quad node\ 1 \overset{N}{\to} node\ 2 \overset{N}{\to} node\ 4 \quad$ and $\quad node\ 1 \overset{N}{\to} node\ 3$

Finally, we define an *execution tree*. It essentially represents the execution of a code in a structured way where each node represents a sub-execution of its parent. Formally,

Definition 8 (Execution Tree). *Let $\mathcal{C}_c$ be a computation produced by a code fragment c. An* Execution Tree *(ET) of $\mathcal{C}_c$ is a tree $T = (N, E)$ where:*

- *$\forall n \in N$, n is the execution node induced by a subcomputation of $\mathcal{C}_c$,*
- *The root of the ET is the execution node induced by $\mathcal{C}_c$,*
- *$\forall (n_1, n_2) \in E$. $n_1 \overset{N}{\to} n_2$.*

This definition is a generalization of the usual call tree (CT), which in turn comes from the refutation trees initially defined for AD in [28, 27]. One important difference between them is that, given a computation $\mathcal{C}_c$ produced by a code fragment c, the CT associated with $\mathcal{C}_c$ is unique because it is only formed of routine nodes. In contrast, there exist different valid ETs associated with $\mathcal{C}_c$ due to the flexibility introduced by the execution nodes (i.e., with routine nodes only one set N is possible, while with execution nodes different sets N are possible). This flexibility of having several possible valid ETs to represent one computation is interesting because it leaves room for transforming the ET and still being an ET. Contrarily, the CT cannot be transformed because it would not be a CT anymore.

Once the ET is built, the debugger traverses the ET asking the oracle about the correctness of the information stored in each node. Using the answers, the debugger identifies a *buggy node* that is associated with a *buggy code* of the program. We can now formally define the notion of buggy node.

Definition 9 (Buggy Node). *Let $T = (N, E)$ be an execution tree. A* buggy node *of T is an execution node $n = (\mathcal{B}, \mathcal{S}) \in N$ where:*
(i) $\mathcal{S} =$ wrong, *and*
(ii) $\forall n' = (\mathcal{B}', \mathcal{S}') \in N$, $(n, n') \in E$. $\mathcal{S}' =$ correct.
Moreover, we say that a buggy node n is traceable *if and only if:*
(iii) $\forall n' = (\mathcal{B}', \mathcal{S}') \in N$, $(n', n) \in E^*$. $\mathcal{S}' =$ wrong.

We use E^* to refer to the symmetric and transitive closure of E. This is the usual definition of buggy node (see, e.g., [23]): a wrong node with all its children correct. We also introduce the notion of *traceable*. Roughly, traceable buggy nodes are those buggy nodes that may be directly responsible of the wrong behavior of the program (their effects are visible in the root of the tree). This property makes them debuggable by all AD strategies that are variants of Top-Down (see [29]).

Lemma 1 (Buggy Code). *Let T be an ET with a buggy node $((s,d,s'),\mathcal{S})$ whose children are $((s_1,d_1,s'_1),\mathcal{S}_1)$, $((s_2,d_2,s'_2),\mathcal{S}_2)\ldots((s_n,d_n,s'_n),\mathcal{S}_n)$. Then, $d \setminus \bigcup_{1\leq i\leq n} d_i$ contains a bug.*

Note that we use (s,d,s') meaning $(s,\mathcal{P}_{C_c}(c),s')$ for some c, and $\setminus$ is the set difference operator.

Proof (Buggy Code). Trivial adaptation from the proof by Lloyd [17] for Prolog.

Lemma 1 illustrates what (buggy) code should be shown to the user. When a buggy node is detected, the (buggy) code shown to the user is the code of the buggy node minus the code of its children.

4.2 Routine Tree

In this section we formalize the notion of RT used in most AD literature as a particular case of the ET. We call *routine tree* to this specialization of the ET to make explicit its multi-paradigm nature, because routines can refer to functions, procedures, methods, predicates, etc. We first define a *routine node*, which is a specialization of an execution node.

Definition 10 (Routine Node). *A* routine node *is an execution node $((s_0,\mathcal{P}_{C_c}(c),s_n),\mathcal{S})$ where code fragment c only contains:*

- *a routine call r, together with*
- *all the code of the routines directly or indirectly called from r.*

Therefore, in a routine node, s_0 and s_n are, respectively, the states just before and after the execution of the called routine. Almost all implementations reduce c to the routine call, and they skip the code of the own routine.

Definition 11 (Routine Tree). *A* routine tree *is an execution tree where all nodes are routine nodes.*

4.3 Search Strategies for AD

Once the ET is built, AD uses a search strategy to select one node. During many years, the main goal of most AD researchers has been the definition of better strategies to reduce the search space after every answer, and to reduce the complexity of the questions. A survey of search strategies for AD can be found in [29]. In our formalization, a search strategy is just a function that analyzes the ET and returns an execution node (either the next node to ask, or a buggy node).

Definition 12 (Strategy). *A* search strategy *is a function whose input is an execution tree $T=(N,E)$ and whose output is an execution node $n=(\mathcal{B},\mathcal{S})\in N$ such that:*

1. *$\mathcal{S}=$ undefined, or*
2. *n is a buggy node.*

4.4 AD Transformations

Some of the last research developments in AD have focussed on the definition of transformations of the ET. The goal of these transformations is to improve the structure of the ET before the debugging session starts, so that search strategies become more efficient. Some of these transformations cannot be applied to a routine tree. For this reason, we include this section to classify the kinds of transformations that have been defined so far, and establish a hierarchy so that future transformations can be also classified in.

There exist three essential elements in the front-end of an algorithmic debugger. The modification of any of them can lead to a different final output of the front-end (i.e., a different ET). Therefore, we classify the transformations in three different levels:

- Transformations of the *source code*: Transformations of the source code such as *inlining* are used to reduce the size of the ET by hiding routines. Contrarily, transformations such as *loops to recursion* [14] are used to augment the size of the ET to reduce the granularity of the reported buggy code (a loop instead of a routine). In general, users should not be aware of the internal transformations applied by the debugger, thus the code fragment shown to the user should be the original code.
- Transformations of the *execution*: Transforming the way in which the source code is executed can change the generated ET. One example is changing eager evaluation by lazy evaluation. Another example is passing arguments by value instead of passing them by reference. We are not aware of any implementation that includes this kind of transformations.
- Transformations of the *ET*: Transforming the ET can significantly reduce the number of questions generated. In general, the ET is transformed with the aim of making search strategies to behave as a dichotomic search. Hence, they try to produce balanced ETs [15], or also deep trees that can be cut in the middle. Other transformations such as *Tree compression* [9] try to avoid the repetition of questions about the same routine, or try to improve the understandability of questions. This is the case of the *Node simplification* transformation, which reduces all terms to normal form [6].

In Figure 4 we classify four AD transformations already available in the state of the art. Two of them, *tree balancing* and *loop expansion* produce ETs that are not routine trees.

4.5 An AD Scheme

Finally, we describe Algorithm 1, a general schema of an algorithmic debugger that includes all phases, from the generation of the ET to the reported bug. This algorithm gives an idea of how and when, the ET, the transformations, the oracle, and the search strategies participate in the whole debugging process.

The main function performs the two phases of AD (Lines 1-2) and then returns a buggy code of the program (Line 3). In the first phase (*getExecutionTree* function) the ET is created performing all possible transformations in the source code (Line 1), in the execution (Line 3) and in the ET (Line 5). Once the ET is created, the second phase

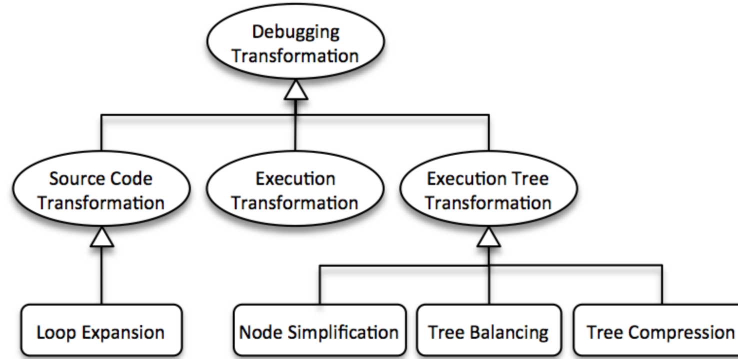

Fig. 4. AD transformations hierarchy

(*debugTree* function) starts. During this phase, the debugger traverses the ET selecting nodes with a search strategy (Line 2). The *selectNode* function is an implementation of one of the search strategies in the literature. There has been a lot of research for more than a decade concerning which should be the node to ask. A survey can be found in [29]. No matter what strategy is used, *selectNode* returns a node to ask (the state of the node is *undefined*), or a buggy node (the state of the node is *wrong* (Line 3)). Once a node has been selected, the debugger asks the oracle about its correctness (Line 5). The oracle provides the intended interpretation to the algorithm. With the answer of the oracle, the debugger updates the state of the nodes of the ET (Lines 5-7). Note that the answer of the oracle can affect the state of several nodes. This effectively changes the information of the ET, and thus, at this moment, a new ET transformation could be used to optimize the ET (Line 8). Then, the process is repeated selecting more nodes. When the strategy finds a buggy node (Lines 3-4) or it cannot select more nodes (Line 1) the second phase finishes and the debugger returns (see *getCode* function) the buggy code associated with the found buggy node (see Lemma 1), or it returns a message indicating that there does not exist a bug (it is indicated with $\perp$ in Line 3), respectively. The last case happens, e.g., when all nodes are reported as *correct*.

5 Conclusion

In this paper we report about some of the problems identified in the current state of the art of AD. One of the problems identified is that much of the recent work in the area does not fit into the standard notions and definitions of AD. In particular, we claim that practically all current definitions of the ET are obsolete with respect to the new proposed techniques.

To solve this situation we propose a generalization of AD able to represent all existent AD transformations. We make this abstraction considering theoretical developments done for a particular language or technique that are generalized, but also considering novel implementations of AD that include techniques that have not been formalized.

Algorithm 1 Main algorithm of an Algorithmic Debugger

Input: A program P and its input i.
Output: A buggy code c in P, or $\perp$ if no bug is detected in P.
Initializations: $\mathcal{A} = \emptyset$ // Set of answers provided by the oracle

begin
1) $T = getExecutionTree(P,i)$
2) $n = debugTree(T)$
3) **return** $getCode(n,T)$
end

function $debugTree(T = (N,E))$
begin
1) **while** $(\exists(\mathcal{B}',\mathcal{S}') \in N, \mathcal{S}' = undef \vee wrong)$
2) $(\mathcal{B},\mathcal{S}) = selectNode(T)$ // Strategy
3) **if** $(\mathcal{S} = wrong)$ **then**
4) **return** $(\mathcal{B},\mathcal{S})$
5) $answer = askOracle(\mathcal{B})$
6) $\mathcal{A} = \mathcal{A} \cup (\mathcal{B}, answer)$
7) $updateStates(\mathcal{A},N)$
8) $T = executionTreeTransformations(T)$
9) **return** $\perp$
end

function $getExecutionTree(P,i)$
begin
1) $P' = sourceCodeTransformations(P)$
2) $\mathcal{E}_{P'} = executeProgram(P',i)$
3) $\mathcal{E}'_{P'} = executionTransformations(\mathcal{E}_{P'})$
4) $T = generateExecutionTree(\mathcal{E}'_{P'})$
5) $T' = executionTreeTransformations(T)$
6) **return** T'
end

function $getCode(n, T = (N,E))$
begin
1) **if** $(n = ((s_0,d,s_n),\mathcal{S}))$ **then**
2) **return** $d \setminus \bigcup_{(n,((s'_0,d_i,s'_n),\mathcal{S}'))\in E} d_i$
3) **return** $\perp$
end

The main objectives of this work are two: First, putting together different ideas that have appeared in many works of AD. Putting these ideas together provides a wide perspective that allows us to make a step forward in the abstraction and generalization of the theoretical side of AD. In addition, it allows for classifications and taxonomies to help understanding the state of the art. Second, our new formulation of AD tries to save time. Many researchers have defined once and again similar concepts used in different languages and tools. We provide a paradigm-independent definition that is general enough to represent all current techniques, and it can be easily instantiated to any particular language.

Our plan for the immediate future work is to extend the model to also consider concurrency. Our ETs can represent concurrency, but Algorithm 1 completely ignores it. We want to study how concurrency should be represented, asked about, answered, and presented to the user when a bug is found. We will initially implement a debugger for concurrent programs. Then, we will try to generalize the model.

6 Acknowledgements

This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under Grant TIN2013-44742-C4-1-R and by the *Generalitat Valenciana* under Grant PROMETEOII/2015/013. David Insa was partially supported by the Spanish Ministerio de Educación under FPU Grant AP2010-4415. The authors thank the

anonymous referees of LOPSTR for their constructive feedback that has contributed to improve this work. They also thank Ehud Shapiro for providing historical information about algorithmic debugging.

References

1. Eclipse. Available from URL: `http://www.eclipse.org/`, 2003.
2. T. Barbour and L. Naish. Declarative debugging of a logical-functional language. Technical report, University of Melbourne, 1994.
3. B. Brael and H. Siegel. Debugging lazy functional programs by asking the oracle. In O. Chitil, Z. Horváth, and V. Zsók, editors, *Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes in Computer Science*, pages 183–200. Springer Berlin Heidelberg, 2007.
4. R. Caballero. A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In *Proceedings of the 2005 ACM-SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05)*, pages 8–13, New York, USA, 2005. ACM Press.
5. R. Caballero, E. Martin-Martin, A. Riesco, and S. Tamarit. EDD: A declarative debugger for sequential erlang programs. In E. Ábrahám and K. Havelund, editors, *20th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*, volume 8413 of *Lecture Notes in Computer Science*, pages 581–586. Springer, apr 2014.
6. R. Caballero, A. Riesco, A. Verdejo, and N. Martí-Oliet. Simplifying questions in maude declarative debugger by transforming proof trees. In G. Vidal, editor, *21st International Symposium Logic-Based Program Synthesis and Transformation (LOPSTR 2011)*, volume 7225 of *Lecture Notes in Computer Science*, pages 73–89. Springer, jul 2011.
7. D. Cheda and J. Silva. State of the practice in algorithmic debugging. *Electronic Notes in Theoretical Computer Science*, 246:55–70, 2009.
8. T. Davie and O. Chitil. Hat-delta: One Right Does Make a Wrong. In A. Butterfield, editor, *Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages (IFL'05)*, page 11, sep 2005.
9. T. Davie and O. Chitil. Hat-delta: One Right Does Make a Wrong. In *Proceedings of the 7th Symposium on Trends in Functional Programming (TFP'06)*, apr 2006.
10. P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimóthy. Generalized Algorithmic Debugging and Testing. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):303–322, dec 1992.
11. J. González, D. Insa, and J. Silva. A New Hybrid Debugging Architecture for Eclipse. In *Proceedings of the 23th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2013)*, volume 8901 of *Lecture Notes in Computer Science (LNCS)*, pages 183–201. Springer, 2014.
12. C. Hermanns and H. Kuchen. Hybrid Debugging of Java Programs. In M. J. Escalona, J. Cordeiro, and B. Shishkov, editors, *Software and Data Technologies*, volume 303 of *Communications in Computer and Information Science*, pages 91–107. Springer Berlin Heidelberg, 2013.
13. D. Insa and J. Silva. An Algorithmic Debugger for Java. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–6, 2010.
14. D. Insa and J. Silva. Automatic transformation of iterative loops into recursive methods. *Information and Software Technology*, 58:95–109, 2015.
15. D. Insa, J. Silva, and A. Riesco. Speeding Up Algorithmic Debugging Using Balanced Execution Trees. In M. Veanes and L. Viganò, editors, *Proceedings of the 7th International*

Conference Tests and Proofs (TAP 2013), volume 7942 of *Lecture Notes in Computer Science (LNCS)*, pages 133–151. Springer, jun 2013.

16. D. Insa, J. Silva, and C. Tomás. Enhancing Declarative Debugging with Loop Expansion and Tree Compression. In E. Albert, editor, *Proceedings of the 22th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2012)*, volume 7844 of *Lecture Notes in Computer Science (LNCS)*, pages 71–88. Springer, sep 2012.
17. J. Lloyd. Declarative Error Diagnosis. *New Generation Computing*, 5(2):133–154, 1987.
18. W. Lux. Münster Curry User's Guide. Available at URL: `http://danae.uni-muenster.de/~lux/curry/user.pdf`, may 2006.
19. I. D. MacLarty. *Practical Declarative Debugging of Mercury Programs*. PhD thesis, University of Melbourne, 2005.
20. L. Naish, P. W. Dart, and J. Zobel. The NU-Prolog Debugging Environment. In A. Porto, editor, *Proceedings of the 6th International Conference on Logic Programming (ICLP'89)*, pages 521–536, Lisboa, Portugal, 1989.
21. H. Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping, Sweden, may 1998.
22. H. Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.
23. H. Nilsson and P. Fritzson. Algorithmic Debugging for Lazy Functional Languages. *Journal of Functional Programming*, 4(3):337–370, 1994.
24. H. Nilsson and J. Sparud. The evaluation dependence tree: an execution record for lazy functional debugging. Technical report, Department of Computer and Information Science, Linköping, 1996.
25. H. Nilsson and J. Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4(2):121–150, 1997.
26. B. Pope. *A Declarative Debugger for Haskell*. PhD thesis, The University of Melbourne, Australia, 2006.
27. E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.
28. E. Y. Shapiro. Inductive inference of theories from facts. Technical Report RR 192, Yale University (New Haven, CT US), 1981.
29. J. Silva. A Survey on Algorithmic Debugging Strategies. *Advances in Engineering Software*, 42(11):976–991, nov 2011.
30. J. Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 44(3), 2012.
31. B. Thompson and L. Naish. A guide to the nu-prolog debugging environment. Technical report, University of Melbourne, 1997.