Algorithmic Debugging Generalized[☆]

David Insa[a], Josep Silva[a,*]

[a]*Departamento de Sistemas Informáticos y Computación*
Universitat Politècnica de València
Camino de Vera s/n, 46022 Valencia, Spain

Abstract

Algorithmic debugging is a semi-automatic debugging technique that abstracts the operational details of computations, allowing the programmers to debug their code from an abstract point of view. However, its use in practice is still marginal, and one of the reasons is the lack of precision of this technique when reporting errors (current algorithmic debuggers do not point an expression or line as buggy, but they point a whole procedure/function/method as containing the bug). In this paper, we make a step forward to overcome this problem. We identify two specific causes of that problem in the standard formulation and implementations of algorithmic debugging, and we present a reformulation to solve both problems. We show that the novel ideas included in the reformulation proposed cannot be supported by the standard internal data structures (such as the Execution Tree) used in this technique and, hence, a generalisation of the standard definitions and algorithms is needed. The reformulation has been done in a language-independent manner to make it useful and reusable in different programming languages.

Keywords: Algorithmic Debugging, Transformation, Generalization

> *"Everything is vague to a degree you do not realize till you have tried to make it precise"*. Bertrand Russell

1. Introduction

Algorithmic Debugging (AD) [28, 6] is a semi-automatic debugging technique with a high level of abstraction. It is composed of two independent phases: error diagnosis and error correction. 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],

[☆]This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad* under grant TIN2013-44742-C4-1-R and TIN2016-76843-C4-1-R, and by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic).

[*]Corresponding Author. Phone Number: (+34) 96 387 7007 (Ext. 73530)
Email addresses: dinsa@dsic.upv.es (David Insa), jsilva@dsic.upv.es (Josep Silva)
URL: http://www.dsic.upv.es/~dinsa/ (David Insa), http://www.dsic.upv.es/~jsilva/ (Josep Silva)

to improve interaction with the user [7], 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 [31]. Compared to other fault localization techniques [2, 33], the main advantage of AD is its high level of abstraction.

Example 1.1. *Let us assume the existence of a buggy Java code composed of three methods: the* `boolean isPrime(int x)` *method determines whether its argument x is a prime number, the* `boolean isDivisible(int x, int y)` *method returns* `true` *if x is divisible by y, or* `false` *otherwise; and, the* `double inflectionPoint(int x)` *method returns the inflection point y such that any divisor d_1 of x greater than y is* associated *with another divisor d_2 of x smaller than y (i.e., $\forall d_1 \mid y < d_1 \,.\, \exists d_2 \mid d_2 < y \wedge d_1 * d_2 = x$). Therefore, with the following method invocation:*

```
isPrime(9);
```

the result should be `false`*. Nevertheless, due to a bug in the code, the result is* `true`*.*

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) isPrime(9) = true?  No
     (2) inflectionPoint(9) = 3.0?  Yes
     (3) isDivisible(9, 2) = false?  Yes
Bug found in method "isPrime" with the call "isPrime(9)".
```

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

```
(1) double inflectionPoint(int n) {
(2)     return Math.sqrt(n);
(3) }
(4) boolean isDivisible(int n1, int n2) {
(5)     return n1 % n2 == 0;
(6) }
```

```
(7) boolean isPrime(int number) {
(8)     if (number <= 1)
(9)         return false;
(10)    double lim = inflectionPoint(number);
(11)    for (int i = 2; i < lim; i++)
(12)        if (isDivisible(number, i))
(13)            return false;
(14)    return true;
(15) }
```

Figure 1: Java program to check whether a number is prime

The debugger internally generates 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

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

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.*

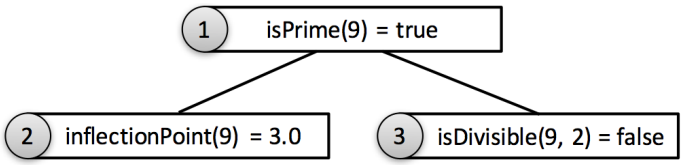

Figure 2: ET generated for the program in Figure 1

In this simple example, we can see one important weakness of the classical formulation of AD. Note that the bug in Figure 1 is that `i < lim` should be `i <= lim`. However, the debugger reports the whole `isPrime` method as buggy, which is very imprecise (it even includes non-executed code such as lines 9 and 13). In this work, we overcome this problem with a reformulation of AD.

1.1. Contributions of this work

In this work, we improve the classical error diagnosis phase of AD. In particular, we propose a new redefinition of this phase in such a way that: (i) It is 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 special 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 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).

We do not claim that our definitions cannot be further generalized, but we do claim (and show) that this generalization is appropriate and enough for the current state of the art (in contrast with the classic formulation). We also provide some UML models to clarify and categorize different ideas introduced in the recent bibliography, so that AD techniques can be classified with them. We also identify and formally define properties that researchers should report about their AD techniques, because these properties allow other researchers and developers to know whether a given technique is useful for a particular application. Finally, we propose a taxonomy of AD techniques, and we classify current techniques according to this taxonomy. For each technique and property, we prove whether the property holds in that technique.

[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 focus the discussion on the imperative paradigm and our examples on Java.

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 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 [8], and including our own implementations—have fundamental problems that were somehow inherited from the original formulation of AD [28].

In particular, the original formulation of AD and most of the later definitions and implementations are obsolete with respect to the recent 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 errors found. We can illustrate these problems observing again the debugging session of Example 1.1: The whole method `isPrime` is pointed out as buggy. This is very imprecise especially if `isPrime` were a method with a lot of code. However, AD researchers and developers are used to this behaviour, 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 lines 9 and 13 of Figure 1 are never executed, and thus they 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 subsequent 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) [9], 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 `isPrime` as buggy together with a counterexample that produces the bug (the buggy node found). Unfortunately, none of these debuggers makes 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.

To reduce the granularity of the errors reported, new techniques have appeared that allow for debugging inside a method (see, e.g., [16, 5]). 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., [7, 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 many mature languages (e.g., Java, Haskell, Prolog, Erlang, etc.). Most 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. Over the years, different paradigms have 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 [27]. 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 [1, 32] 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 EDT itself 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 [9], 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. In contrast 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] (both are Eclipse plugins). These tools use the presentation facilities of the development environment, and 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:

Tree compression deletes nodes of the ET, breaking the standard parent-child relation in the ET.

Tree balancing introduces new artificial nodes, breaking the standard definition of ET node.

Loop expansion and *ET zooming* decompose nodes, breaking the standard definition of ET node.

We are not aware of any definition of ET able to represent the previous four techniques. In Section 4.4, we show that the above techniques are valid ET transformations for the new (more general) notion of ET that we introduce in the next section, and we state the specific properties these techniques preserve when transforming ETs. Formal proofs are included in Appendix A.

4. A Generalized Reformulation of Algorithmic Debugging

Some of the last developments for AD cannot be formalized with the standard AD formulation. In a few cases, they just skipped the formalization of the technique and provided an implementation. In other cases, they wanted to prove some properties, and thus they 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 they also formalized a different part of the system with different data structures. We have observed this behaviour again and again and, clearly, this is due to the lack of a standard solution.

We want to provide a definition of AD that solves the problems described in Section 2 and that is general enough to be used in different languages. To 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]). In contrast, 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 errors reported to single expressions.

In the following, we only use *Execution Tree* to denote our new definition, and we use *Routine Tree* (RT) to denote the traditional definition (that we also formalize in the following 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.

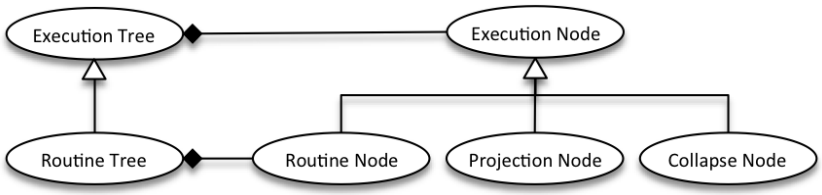

Figure 3: UML model representing the structure of the execution tree

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.

Example 4.1 (Running example). *From here on, we will use the following bug-free program that calculates the square root of variable x:*

```
(1) if (x < 0)
(2)     return Double.NaN;
(3) double b = x;
(4) while (Math.abs(b * b - x) > 1e-12)
(5)     b = ((x / b) + b) / 2;
(6) return b;
```

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 language-independent. Hence, we consider programs as state transition systems.

Definition 4.2 (Program). *A* program $P = \{W, I, R, C\}$ *consists of:*

- W*: A set of* environments*.*
- I*: A set of* initial environments*, 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 expressions.*

Definition 4.3 (Computation). *A* computation *is a maximal sequence of environments* $e_1, e_2, \ldots$ *such that:*

- e_1 *is an initial environment, i.e.,* $e_1 \in I$*.*
- $(e_i, e_{i+1}) \in R$ *for all* $i \geq 1$ *(and* $i \leq n-1$*, if the sequence is of the finite length* n*).*

A finite segment $e_i, e_{i+1}, \ldots, e_j$ *where* $1 \leq i < j \leq n$ *is called a* subcomputation*.*

Example 4.4. *The computation produced in the running example (Example 4.1) assuming that the initial environment is* `(x=9)` *is shown below:*

e_1 $(x=9)$	e_5 $(x=9, b=5)$	e_9 $(x=9, b=3.0235)$	e_{13} $(x=9, b=3.000000001)$
e_2 $(x=9)$	e_6 $(x=9, b=5)$	e_{10} $(x=9, b=3.0235)$	e_{14} $(x=9, b=3.000000001)$
e_3 $(x=9, b=9)$	e_7 $(x=9, b=3.4)$	e_{11} $(x=9, b=3.0001)$	e_{15} $(x=9, b=3)$
e_4 $(x=9, b=9)$	e_8 $(x=9, b=3.4)$	e_{12} $(x=9, b=3.0001)$	e_{16} $(x=9, b=3, r=3)$

In the final environment, we represent with r *the result of* `return` *(to represent this* r*, debuggers often use graphical notation or they use the routine call with completely evaluated arguments).*

In the source code of a program, we consider expressions[3] as the basic execution unit. Therefore, in the following, the source code of a program P is a set of expressions $ex_1, ex_2, \ldots, ex_n$ that produces the computation $e_1, e_2, \ldots, e_m$ for a given initial environment e_1. We cannot provide a specific

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

model of computation if we want to be language-independent, thus we do not define the relation between expressions 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 environment, a code, and a final environment to identify bugs, no matter how the code makes the transition from the initial environment to the final environment. The user will decide whether this transition should have occurred in the execution.

We also left undefined the notion of environment. In the following, we will refer to environments in an abstract way, and they should not be only considered as a mapping from declared variables to values (i.e., states, as in the imperative programming paradigm). Hence, the environments shown in Example 4.4 are just a particular case. In their general form, environments can include all the information needed to identify a point in a computation. Each debugger implementor will decide what information in the environment should be shown to allow programmers to answer the debugger questions. For instance, the environment before and after a function call in the functional paradigm could be $(x = 42)\ f\ x\ (R = 0)$ where R represents the result of the function call $f\ 42$, and it could be shown to the programmer, e.g., with $f\ 42 \Rightarrow 0$. In the original formulation by Shapiro (for the logic paradigm) [28], the environment was defined as "*a vector over some domain D... that should be interpreted as the input/output of a procedure*".

Because the considered execution unit is the expression, it is possible to identify a bug in a single expression. 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 expressions in the source code C of P that produces a subcomputation $e_i, \ldots, e_j$ 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 expressions such as function calls together with the whole called function.

Not all the expressions 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 4.5 (Code Projection). *Given a code fragment c and a computation $\mathcal{C}_c = e_1, \ldots, e_n$ produced by c from a given initial environment e_1, a* projection of c modulo $\mathcal{C}_c$*, written $\mathcal{P}_{\mathcal{C}_c}(c)$, is a code fragment that contains the minimum subset of c needed to produce the computation $\mathcal{C}_c$.*

The initial and final environments, e_i and e_j, describe the effects of a given code fragment c. All three together form a *code behaviour*.

Definition 4.6 (Code Behaviour). *Given a code fragment c and a computation $\mathcal{C}_c = e_1, \ldots, e_n$ produced by c from a given initial environment e_1, the* code behaviour *of $\mathcal{C}_c$ is a triple $\langle e_1, p, e_n \rangle$, where $p = \mathcal{P}_{\mathcal{C}_c}(c)$ is the projection of c modulo $\mathcal{C}_c$.*

Example 4.7. *Considering from our running example the code fragment c that corresponds with lines (1), (2), and (3); and the computation $\mathcal{C}_c = e_1, e_2, e_3$ produced by c from the initial environment* `(x=9)` *(see Example 4.4), the associated code behaviour is $\langle e_1, (1)(3), e_3 \rangle$. Here, (1)(3) is the projection of c modulo $\mathcal{C}_c$ because it is the only code actually executed.*

The code behaviour corresponds to the questions asked by the debugger. These questions are along the lines of: *At this point of the computation, the code p with the initial environment e_1 produced the final environment e_n. Is this what you intended?* Many previous definitions of AD (see, e.g., [22, 5]) define the code behaviour as the triple $\langle e_1, c, e_n \rangle$, 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 routine itself c. Definition 4.6, 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 behaviour only contains the code actually needed to produce that behaviour.

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

Definition 4.8 (Intended Model). *Given a program $P = \{W, I, R, C\}$, an* intended model $\mathcal{M}$ *for P is a set of tuples $\langle e_i, IC(\mathcal{P}(c)), e_j \rangle$ where $e_i, e_j \in W$, $\mathcal{P}(c)$ is a projection of a code fragment $c \subseteq C$ and $IC(p)$ is the intended code that p should implement.*

Each tuple of the form $\langle e_i, p, e_j \rangle$ specifies that the execution of code p from environment e_i leads to environment e_j. Intuitively, an intended model of a program contains the set of code behaviours that the programmer had in mind when they programmed these codes. It is used as a reference point against which one can compare computations to determine whether they are the expected ones or not. Note that $IC(\mathcal{P}(c))$ uniquely identifies c (for instance, it may specify archive, start line, start column, end line, end column), and thus it is possible that different codes c_1, c_2 that point to different lines contain the same characters (e.g., $x = y$). Therefore, it is perfectly possible that they have different intended codes that produce different final environments even with the same initial environment (i.e., the first $x = y$ is correct while the second $x = y$ should be $x = y + 1$).
We are now in a position to define the nodes of an execution tree.

Definition 4.9 (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 $\langle \mathcal{B}, \mathcal{S} \rangle$ where:*

1. *$\mathcal{B} = \langle e_i, \mathcal{P}_{\mathcal{C}_c}(c), e_j \rangle$ is the code behaviour 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 $\mathcal{M}$:* $\begin{cases} \text{correct} & \text{if } e_j = e_f \wedge \langle e_i, IC(\mathcal{P}(c)), e_f \rangle \in \mathcal{M} \\ \text{wrong} & \text{if } e_j \neq e_f \wedge \langle e_i, IC(\mathcal{P}(c)), e_f \rangle \in \mathcal{M} \end{cases}$

Observe that an execution node contains (inside $\mathcal{B}$) the source code $\mathcal{P}_{\mathcal{C}_c}(c)$ responsible for 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 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 4.10 (Execution Nodes Dependency). *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 a 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. Therefore, 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 $n_1 \not\xrightarrow{N} n_3$.

Example 4.11. *Consider our running example with the computation obtained in Example 4.4. We can generate the following execution nodes (among others):*

	(initial environment)	code	(end environment)
node 1:	(x=9)	(1)(3−6)	(x=9,b=3)
node 2:	(x=9)	(1)	(x=9)
node 3:	(x=9,b=9)	(4−5)	(x=9,b=3)
node 4:	(x=9,b=9)	(4−5)	(x=9,b=5)
node 5:	(x=9,b=9)	(5)	(x=9,b=5)
node 6:	(x=9,b=3.4)	(4−5)	(x=9,b=3.0235)
node 7:	(x=9,b=3)	(6)	(x=9,b=3,r=3)

with N=$\{$node 1, node 2, node 3, node 4, node 5, node 6, node 7$\}$ we have node $1 \xrightarrow{N}$ *node* 2, *node* $1 \xrightarrow{N}$ *node* 3, *node* $1 \xrightarrow{N}$ *node* 7, *node* $3 \xrightarrow{N}$ *node* 4, *node* $3 \xrightarrow{N}$ *node* 6, *node* $4 \xrightarrow{N}$ *node* 5 *(see Figure 4). In these execution nodes, node 1 refers to the execution of the whole code. Node 2 refers to the execution of the* `if` *expression at the beginning. Node 3 refers to the execution of the whole loop. Node 4 refers to the execution of the first iteration of the loop. Node 5 refers to the execution of the body of the loop during the first iteration. Node 6 refers to the execution of the third iteration of the loop. Finally, node 7 refers to the execution of the* `return` *statement. Note that during the execution of the code, the loop performs 6 iterations. However, only two iterations are explicitly represented, the other ones are contained in 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 4.12 (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 $\mathcal{C}_c$,*

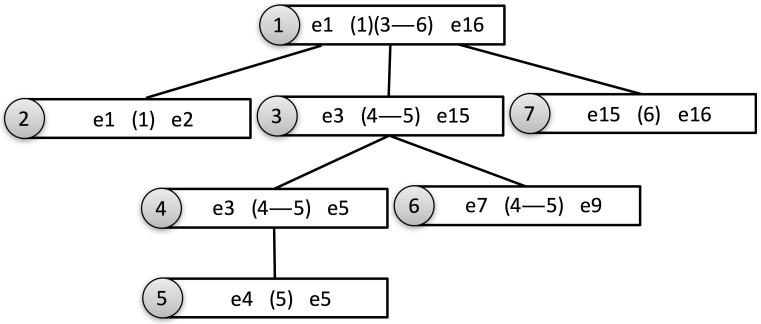

Figure 4: ET associated with the nodes in Example 4.11

- $\forall (n_1, n_2) \in E \;.\; n_1 \overset{N}{\rightarrow} n_2$.

This definition is a generalization of the usual call tree (CT), which in turn comes from the refutation trees initially defined for AD in [27, 28]. 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 execution nodes (i.e., with routine nodes only one set N is possible, whereas 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. In contrast, 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 4.13 (Buggy Node). *Let $T = (N, E)$ be an execution tree. A* buggy node *of T is an execution node $n = \langle \mathcal{B}, \mathcal{S} \rangle \in N$ where:*
(i) $\mathcal{S} =$ wrong*, and*
(ii) $\forall n' = \langle \mathcal{B}', \mathcal{S}' \rangle \in N$, $(n, n') \in E \;.\; \mathcal{S}' =$ correct.
Moreover, we say that a buggy node n is traceable *if and only if:*
(iii) $\forall n' = \langle \mathcal{B}', \mathcal{S}' \rangle \in N$, $(n', n) \in E^* \;.\; \mathcal{S}' =$ 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 for the wrong behaviour 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 $\langle \langle e, \mathcal{P}_{\mathcal{C}_c}(c), e' \rangle, \mathcal{S} \rangle$ whose children are $\langle \langle e_1, \mathcal{P}_{\mathcal{C}_{c_1}}(c_1), e'_1 \rangle, \mathcal{S}_1 \rangle$, $\langle \langle e_2, \mathcal{P}_{\mathcal{C}_{c_2}}(c_2), e'_2 \rangle, \mathcal{S}_2 \rangle \ldots \langle \langle e_n, \mathcal{P}_{\mathcal{C}_{c_n}}(c_n), e'_n \rangle, \mathcal{S}_n \rangle$, let $B = \{e\} \cup \{e_x \mid 1 \leq$*

$x \leq n\} \cup \{e'_y \mid 1 \leq y \leq n\} \cup \{e'\}$ *be a set that contains the initial and final environments of the (sub)computations, and let* $Z = \{e_i \ldots e_j \mid i < j \wedge e_i, e_j \in B \wedge \nexists x, y \,.\, 1 \leq x \leq n \wedge i \leq y < j \wedge e_y, e_{y+1} \in \mathcal{C}_{c_x}\}$ *be the set of subcomputations performed in the parent node but not performed in its children. Then, the set of expressions* $EX = \bigcup_{\mathcal{C}_z \in Z} \mathcal{P}_{\mathcal{C}_z}(c)$ *contains a bug.*

Proof. [Buggy Code] The proof is analogous to 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 that produces all the subcomputation of the buggy node that are not produced in its children. This is illustrated in Example 4.14.

Example 4.14. *Consider again the ET in Figure 4. The program in our running example is correct, but assume that line (3) contains a bug, e.g., the code should obtain the square root of the integer value of the argument (i.e.,* `double b = (int) x;`*). In order to obtain the buggy code associated with the wrong node (i.e.,* n_3*), we need to have in mind the following information:*

$n_1 = \langle\langle e_1, (1)(3-6), e_{16}\rangle, \text{wrong}\rangle$*, where lines (1)(3-6) are the projection of* $e_1 \ldots e_{16}$
$n_2 = \langle\langle e_1, (1), e_2\rangle, \text{correct}\rangle$*, where line (1) is the projection of* $e_1 e_2$
$n_3 = \langle\langle e_3, (4-5), e_{15}\rangle, \text{correct}\rangle$*, where lines (4-5) are the projection of* $e_3 \ldots e_{15}$
$n_7 = \langle\langle e_{15}, (6), e_{16}\rangle, \text{correct}\rangle$*, where line (6) is the projection of* $e_{15} e_{16}$
$B = \{e_1, e_2, e_3, e_{15}, e_{16}\}$*, which is an auxiliary set for computing the Z set*
$Z = \{e_2 e_3\}$*, which corresponds to the environments produced in* n_1 *but not in* n_2*,* n_3*, or* n_7

Now, it is straightforward to obtain the code that contains the bug: The information stored in Z shows that the bug corresponds to the statement that converted e_2 *into* e_3*. Hence, it corresponds with line (3). In this example, we have seen that the buggy code corresponds with a code that has been executed by the buggy node but not by its children. However, it is also possible that a code executed by a child does not manifest a bug but it is actually buggy. Let us see an example.*

Assume now that we only have one bug that is manifested in the fourth iteration of the loop. In such a case, node 3 (n_3*) would be wrong and nodes 4 (*n_4*) and 6 (*n_6*) correct. In order to obtain the buggy code, we need to have in mind the following information:*

$n_3 = \langle\langle e_3, (4-5), e_{15}\rangle, \text{wrong}\rangle$*, where lines (4-5) are the projection of* $e_3 \ldots e_{15}$
$n_4 = \langle\langle e_3, (4-5), e_5\rangle, \text{correct}\rangle$*, where lines (4-5) are the projection of* $e_3 e_4 e_5$
$n_6 = \langle\langle e_7, (4-5), e_9\rangle, \text{correct}\rangle$*, where lines (4-5) are the projection of* $e_7 e_8 e_9$
$B = \{e_3, e_5, e_7, e_9, e_{15}\}$*, which is an auxiliary set for computing the Z set*
$Z = \{e_5 e_6 e_7, e_9 \ldots e_{15}\}$*, which corresponds to the environments produced in* n_3 *but not in* n_4 *or* n_6

Now, we can obtain the code that contains the bug. In this case, it corresponds to the statements that converted e_5 *into* e_6*,* e_6 *into* e_7*,* e_9 *into* e_{10}*,* e_{10} *into* e_{11}*, . . . , or* e_{14} *into* e_{15}*. Hence, the bug must be in lines (4) and (5).*

Therefore, we can observe that a code that does not manifest a bug can be buggy. For instance, in the example, we can see that nodes n_4 *and* n_6*, which contain the (buggy) code of the loop, are*

correct whereas their parent node n_3, which contains exactly the same code, is buggy. Lemma 1 ensures that the code reported as buggy is the code of the parent, but not the one of the children, because the bug is only observable in the code behaviour of node n_3.

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 4.15 (Routine Node). *A* routine node *is an execution node* $\langle\langle e_1, \mathcal{P}_{\mathcal{C}_c}(c), e_n\rangle, \mathcal{S}\rangle$ *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, e_1 and e_n are, respectively, the environments 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 routine definition.

Definition 4.16 (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 about, or a buggy node).

Definition 4.17 (Strategy). *A* search strategy *is a function whose input is an execution tree* $T = (N, E)$ *and whose output is an execution node* $n = \langle \mathcal{B}, \mathcal{S}\rangle \in N$ *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 included in the classification.

There exist three essential elements in the back-end of an algorithmic debugger. The modification of any of them can lead to a different final output of the back-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. In contrast, transformations such as *loops to recursion* [16, 14] are used to augment the size of the ET to reduce the granularity of the buggy code reported (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 dichotomical 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 [7].

Given a computation $\mathcal{C}_c$ of a code fragment c, with $T = (N,E)$ being the execution tree of $\mathcal{C}_c$, we represent with $\mathcal{T}(T) = (N',E')$ either: (i) a transformation T' of T, (ii) the ET of a computation $\mathcal{C}'_c$ transformed from $\mathcal{C}_c$, or (iii) the ET of a computation $\mathcal{C}_{c'}$ of a code fragment c' transformed from c. Hence, $\mathcal{T}(T)$ is the ET produced after applying one or more of the previously described transformations. Note that we use T to refer to the ET that would have been generated without applying any transformation.

From here on, we assume the existence of two mappings M_I and M_S that can be directly provided by or inferred from the transformations. $M_I : N' \to N$ is an identity mapping that relates a node $n' \in N'$ with its equivalent $n \in N$ (i.e., the transformation did not affect the generation of n). $M_S : N' \to \{N\}$ is a source mapping that relates a node $n' \in N'$ with the set of nodes $Z \subseteq N$ whose information is used to generate it.

In this section, we also propose a set of interesting properties, so with a quick view we can find out how a transformation behaves. We propose five different properties to see how buggy nodes on a given ET are modified by a transformation: *Buggy Node Completeness*, *Buggy Code Completeness*, *Buggy Code Reduction*, *Buggy Code Semi-reduction*, and *Bug Existence*. All of them are formally presented below.

Property 1 (Buggy Node Completeness). *Let $T = (N,E)$ be an ET, $\mathcal{T}(T) = (N',E')$ its transformed version, and M_I the identity mapping of the transformation. $\forall n \in N$, if n is a buggy node in T then n' is a buggy node in $\mathcal{T}(T)$ and $(n',n) \in M_I$.*

Because all buggy nodes belong to both ETs, then, by Definition 4.12, the same buggy code behaviours are detectable. Hence, all bugs that can be localized in the ET generated before applying

the transformation are still detectable after applying the transformation. Preserving all buggy nodes is often too restrictive. A relaxed version of this property is:

Property 2 (Buggy Code Completeness). *Let $T = (N,E)$ be an ET, and $\mathcal{T}(T) = (N',E')$ its transformed version. $\forall n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N$, n is a buggy node in T . $\exists n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N'$, n' is a buggy node in $\mathcal{T}(T)$ and $p = p'$.*

This property ensures that all the code that can be reported as buggy in the original ET, can also be reported as buggy in the transformed ET. However, the specific context that produced the bug is not necessarily the same in the original and in the transformed version. There exist transformations that violate the last condition ($p = p'$). However, this is not a problem provided that p' is a subset of p, because this means that the code reported as buggy is smaller in the transformed code (i.e, precision is increased by reducing the granularity of the bug found). This is captured by the next property:

Property 3 (Buggy Code Reduction). *Let $T = (N,E)$ be an ET, $\mathcal{T}(T) = (N',E')$ its transformed version, and M_S the source mapping of the transformation. $\forall n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N$, n is a buggy node in T . $\exists n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N'$, n' is a buggy node in $\mathcal{T}(T)$, and either $p = p'$ or $p \supset p' \wedge (n', Z) \in M_S \wedge n \in Z$.*

Roughly, Property Buggy Code Reduction characterizes those transformations where a buggy node is either maintained or used to generate another buggy node where the associated code has been reduced. Hence, precision is increased. Some transformations cannot satisfy Property Buggy Code Reduction because they introduce new source code into the buggy node and thus $p \not\supseteq p'$. However, the new code introduced by the transformation is often correct by construction and thus, precision is still augmented. This is formalized in the following property:

Property 4 (Buggy Code Semi-reduction). *Let $T = (N,E)$ be an ET, $\mathcal{T}(T) = (N',E')$ its transformed version, and M_S the source mapping of the transformation. $\forall n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N$, n is a buggy node in T . $\exists n' = \langle\langle e'_1, p' \cup p'', e'_n\rangle, \mathcal{S}'\rangle \in N'$, n' is a buggy node in $\mathcal{T}(T)$, p'' is correct, and either $p = p'$ or $p \supset p' \wedge (n', Z) \in M_S \wedge n \in Z$.*

Sometimes, these properties are further relaxed to just a bug existence property.

Property 5 (Bug Existence). *Let $T = (N,E)$ be an ET, and $\mathcal{T}(T) = (N',E')$ its transformed version. $\forall n \in N$, n is a buggy node in T . $\exists n' \in N'$, n' is a buggy node in $\mathcal{T}(T)$.*

Property Bug Existence is the minimum exigible requirement to ensure that the transformation cannot hide all bugs.

We also propose six additional properties to show how the ET is modified by the transformation: *Keep Traceability*, *Zoom in Nodes*, *Zoom out Nodes*, *May Hide Bugs*, *May Reveal Bugs*, *ET size*. They are formally presented in the following properties.

Property 6 (Keep Traceability). *Let $T = (N,E)$ be an ET, $\mathcal{T}(T)$ its transformed version, and M_I the identity mapping of the transformation. $\forall n \in N$, n is traceable in T . n' is traceable in $\mathcal{T}(T)$ and $(n', n) \in M_I$.*

Keep Traceability determines whether all traceable nodes are still traceable after the transformation.

Property 7 (Zoom in Nodes). *$\exists \mathcal{T}(T) = (N', E')$ that is the transformed ET obtained from an ET $T = (N, E)$ with the source mapping M_S such that $n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N' \wedge n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N \wedge (n', Z) \in M_S \wedge n \in Z$ where either $p \supset p'$ or $p' = p'' \cup p''' \wedge p \supset p'' \wedge p'''$ is correct.*

Zoom in Nodes determines whether a node can be decomposed in such a way that a new node is created to represent a subcomputation of the original node.

Property 8 (Zoom out Nodes). *$\exists \mathcal{T}(T) = (N', E')$ that is the transformed ET obtained from an ET $T = (N, E)$ with the source mapping M_S such that $n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N' \wedge n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N \wedge (n', Z) \in M_S \wedge n \in Z$ where either $p \subset p'$ or $p' = p'' \cup p''' \wedge p \subset p'' \wedge p'''$ is correct.*

Zoom out Nodes determines whether a node can be created in such a way that an original node represents a subcomputation of the new node.

Property 9 (May Hide Bugs). *$\exists \mathcal{T}(T) = (N', E')$ that is the transformed ET obtained from an ET $T = (N, E)$ such that $n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N$ is a buggy node in T with a bug $b \in p$ and $\nexists n' \in N'$, $n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle$ is a buggy node in $\mathcal{T}(T)$ and $b \in p'$.*

May Hide Bugs determines whether a bug that could have been found before the transformation cannot be found after the transformation.

Property 10 (May Reveal Bugs). *$\exists \mathcal{T}(T) = (N', E')$ that is the transformed ET obtained from an ET $T = (N, E)$ such that $n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N'$ is a buggy node in $\mathcal{T}(T)$ with a bug $b \in p'$ and $\nexists n \in N$, $n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle$ is a buggy node in T and $b \in p$.*

May Reveal Bugs determines whether a bug that could not have been found before the transformation can be found after the transformation.

Property 11 (ET size). *Let $T = (N, E)$ be an ET, and $\mathcal{T}(T) = (N', E')$ its transformed version where $T \neq \mathcal{T}(T)$. We say that the ET size is bigger (i.e., $\uparrow$) when $|N| > |N'|$, equal (i.e., $=$) when $|N| = |N'|$ and smaller (i.e., $\downarrow$) when $|N| < |N'|$.*

ET size shows how the amount of nodes is modified due to the transformation.

In Figure 5, 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. These transformations are classified in Table 1 with respect to the properties they fulfil. In order to present this table, we have proven for each ET transformation and for each property whether the property holds or not for this technique. This amounts a total of 44 proofs, which we have delegated to Appendix A.

In the table, we see that Properties Buggy Node Completeness, Buggy Code Completeness, and Buggy Code Reduction do not hold in any of the four transformations, and only Loop Expansion satisfies Property Buggy Code Semi-reduction. In fact, the main goal of Loop Expansion is to

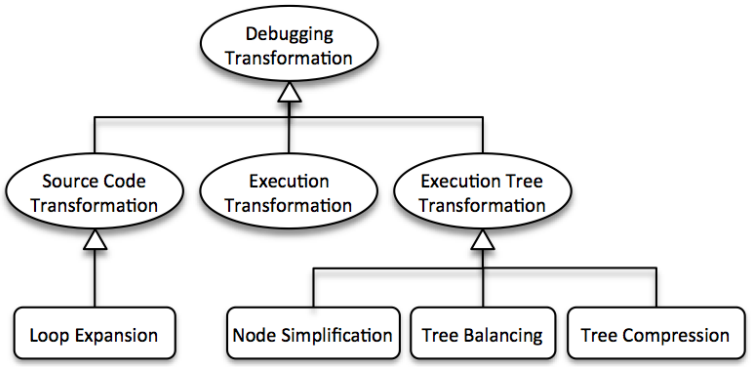

Figure 5: AD transformations hierarchy

	Buggy Node Completeness	Buggy Code Completeness	Buggy Code Reduction	Buggy Code Semi-reduction	Bug Existence	Keep Traceability	Zoom in Nodes	Zoom out Nodes	May Hide Bugs	May Reveal Bugs	ET size
Loop Expansion	×	×	×	✓	✓	×	✓	×	✓	✓	$\uparrow =$
Node Simplification	×	×	×	×	✓	×	✓	✓	✓	✓	$=$
Tree Balancing	×	×	×	×	×	×	×	✓	✓	✓	$\uparrow = \downarrow$
Tree Compression	×	×	×	×	×	×	×	×	✓	✓	$\downarrow$

Table 1: Classification of the transformations

reduce the granularity of the bugs found. Moreover, only Loop Expansion and Node Simplification can ensure the existence of a bug after the transformation (Property Bug Existence).

None of the transformations can ensure the traceability of the original buggy nodes (Property Keep Traceability). Only two of the techniques use decomposition (Property Zoom in Nodes) and composition (Property Zoom out Nodes) of nodes. But all the transformations can hide (Property May Hide Bugs) and reveal (Property May Reveal Bugs) bugs. With respect to the size of the transformed ET (Property ET size), Tree Compression is the only transformation that always reduces the size. Node Simplification always maintains the same size. Loop Expansion can either maintain or increase the size. And, finally, Tree Balancing can reduce, maintain, or increase the size.

4.5. An AD Scheme

Finally, we describe Algorithm 1, a general scheme of an algorithmic debugger that includes all phases, from the generation of the ET to the bug reported. 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.

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) **if** $(n = \perp)$ **then**
4) **return** $\perp$
5) **return** $getCode(n,T)$
end

function $debugTree(T = (N,E))$
begin
1) **while** $(\exists \langle \mathcal{B}', \mathcal{S}' \rangle \in N, \mathcal{S}' = undef \vee wrong)$
2) $\langle \mathcal{B}, \mathcal{S} \rangle = selectNode(T)$ // Strategy
3) **if** $(\mathcal{S} = wrong)$ **then**
4) **return** $\langle \mathcal{B}, \mathcal{S} \rangle$
5) $answer = askOracle(\mathcal{B})$
6) $\mathcal{A} = \mathcal{A} \cup \langle \mathcal{B}, answer \rangle$
7) $updateEnvironments(\mathcal{A}, N)$
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 $getCode(n = \langle \mathcal{B}, \mathcal{S} \rangle, T = (N,E))$
begin
1) $\mathcal{B} = \langle s, \mathcal{P}_{C_c}(c), s' \rangle$
2) $I = \{\langle e_1, C_z, e_n \rangle \mid (n,n') \in E \wedge$
$n' = \langle \langle e_1, \mathcal{P}_{C_z}(z), e_n \rangle, \mathcal{S} \rangle\}$
3) $B = \{s\} \cup \{e_1 \mid \langle e_1, \_, \_ \rangle \in I\} \cup$
$\{e_n \mid \langle \_, \_, e_n \rangle \in I\} \cup \{s'\}$
4) $Z = \{e_i, \ldots, e_j \mid i < j \wedge e_i, e_j \in B \wedge \nexists y \,.\, i \leq y$
$y < j \wedge e_y, e_{y+1} \in C_z \wedge \langle \_, C_z, \_ \rangle \in I\}$
5) **return** $\bigcup_{C_z \in Z} \mathcal{P}_{C_z}(c)$
end

The main function performs the two phases of AD (Lines 1-2) and then returns a buggy code of the program (Lines 3-5). In the first phase (*getExecutionTree* function), first, all possible transformations in the source code (Line 1) and in the execution (Line 3) are performed. Then, the ET is created. Finally, all possible ET transformations are performed over the created ET (Line 5). Once the ET is created, the second phase (*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 node should be asked about. A survey can be found in [29]. No matter what strategy is used, *selectNode* returns a node to be asked about (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 buggy node found (see Lemma 1). When the algorithm did not find any buggy node, then it returns a message indicating that there does not exist a bug (it is indicated with $\perp$ in Line 4). 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 show 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 different AD transformations such as Loop Expansion, Node Simplification, Tree Balancing, and Tree Compression. 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.

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 language-independent definition that is general enough to represent all current techniques, and it can be easily instantiated to any particular language.

The generalization presented has two main advantages over previous formalizations. First, a node in the ET can contain any amount of code, and it is not limited to represent a routine call. Second, the source code associated with a node in the ET only contains the code that has been actually executed in the computation associated with this node.

We have also identified two dimensions to classify ET transformations. The first dimension considers the target of the transformation (a source code, an execution or an ET). We have presented a taxonomy and classification of current ET transformations according to this dimension. The second dimension is associated with the identification of eleven properties of ET transformations. These properties allow us to understand and predict the effects of an ET transformation. We have also classified several current ET transformations, formally proving the properties they fulfil.

6. Acknowledgements

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] Barbour, T., Naish, L., 1994. Declarative debugging of a logical-functional language. Tech. rep., University of Melbourne.

[2] Bond, G., 1994. Logic programs for consistency-based diagnosis. Ph.D. thesis, Carleton University, Ottawa, Canada.

[3] Braßel, B., Siegel, H., 2007. Debugging lazy functional programs by asking the oracle. In: Chitil, O., Horváth, Z., Zsók, V. (Eds.), Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages. Vol. 5083 of Lecture Notes in Computer Science (LNCS). Springer Berlin Heidelberg, pp. 183–200.

[4] Caballero, R., 2005. 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). ACM Press, New York, USA, pp. 8–13.

[5] Caballero, R., Martin-Martin, E., Riesco, A., Tamarit, S., April 2014. EDD: A declarative debugger for sequential Erlang programs. In: Ábrahám, E., Havelund, K. (Eds.), 20th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014). Vol. 8413 of Lecture Notes in Computer Science (LNCS). Springer, pp. 581–586.

[6] Caballero, R., Riesco, A., Silva, J., August 2017. A survey of algorithmic debugging. ACM Comput. Surv. 50 (4), 60:1–60:35.
URL http://doi.acm.org/10.1145/3106740

[7] Caballero, R., Riesco, A., Verdejo, A., Martí-Oliet, N., July 2011. Simplifying questions in Maude declarative debugger by transforming proof trees. In: Vidal, G. (Ed.), 21st International Symposium Logic-Based Program Synthesis and Transformation (LOPSTR 2011). Vol. 7225 of Lecture Notes in Computer Science (LNCS). Springer, pp. 73–89.

[8] Cheda, D., Silva, J., August 2009. State of the practice in algorithmic debugging. Electronic Notes in Theoretical Computer Science 246, 55–70.

[9] Davie, T., Chitil, O., April 2006. Hat-delta: One right does make a wrong. In: Proceedings of the 7th Symposium on Trends in Functional Programming (TFP'06).

[10] Fritzson, P., Shahmehri, N., Kamkar, M., Gyimóthy, T., December 1992. Generalized algorithmic debugging and testing. ACM Letters on Programming Languages and Systems (LOPLAS) 1 (4), 303–322.

[11] González, J., Insa, D., Silva, J., December 2014. A new hybrid debugging architecture for Eclipse. In: Proceedings of the 23rd International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2013). Vol. 8901 of Lecture Notes in Computer Science (LNCS). Springer, pp. 183–201.

[12] Hermanns, C., Kuchen, H., January 2013. Hybrid debugging of Java programs. In: Escalona, M. J., Cordeiro, J., Shishkov, B. (Eds.), Software and Data Technologies. Vol. 303 of Communications in Computer and Information Science. Springer Berlin Heidelberg, pp. 91–107.
URL http://dx.doi.org/10.1007/978-3-642-36177-7_6

[13] Insa, D., Silva, J., September 2010. An algorithmic debugger for Java. In: Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010). pp. 1–6.

[14] Insa, D., Silva, J., February 2015. Automatic transformation of iterative loops into recursive methods. Information and Software Technology 58, 95–109.

[15] Insa, D., Silva, J., Riesco, A., June 2013. Speeding up algorithmic debugging using balanced execution trees. In: Veanes, M., Viganò, L. (Eds.), Proceedings of the 7th International Conference Tests and Proofs (TAP 2013). Vol. 7942 of Lecture Notes in Computer Science (LNCS). Springer, pp. 133–151.

[16] Insa, D., Silva, J., Tomás, C., September 2012. Enhancing declarative debugging with loop expansion and tree compression. In: Albert, E. (Ed.), Proceedings of the 22nd International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2012). Vol. 7844 of Lecture Notes in Computer Science (LNCS). Springer, pp. 71–88.

[17] Lloyd, J., June 1987. Declarative error diagnosis. New Generation Computing 5 (2), 133–154.

[18] Lux, W., May 2006. Münster Curry user's guide. Available at: http://danae.uni-muenster.de/~lux/curry/user.pdf.

[19] MacLarty, I. D., July 2005. Practical declarative debugging of Mercury programs. Ph.D. thesis, University of Melbourne.
[20] Naish, L., Dart, P. W., Zobel, J., June 1989. The NU-Prolog debugging environment. In: Porto, A. (Ed.), Proceedings of the 6th International Conference on Logic Programming (ICLP'89). Lisboa, Portugal, pp. 521–536.
[21] Nilsson, H., May 1998. Declarative debugging for lazy functional languages. Ph.D. thesis, Linköping, Sweden.
[22] Nilsson, H., November 2001. 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.
[23] Nilsson, H., Fritzson, P., July 1994. Algorithmic debugging for lazy functional languages. Journal of Functional Programming 4 (3), 337–370.
[24] Nilsson, H., Sparud, J., September 1996. The evaluation dependence tree: An execution record for lazy functional debugging. Tech. rep., Department of Computer and Information Science, Linköping University.
[25] Nilsson, H., Sparud, J., April 1997. The evaluation dependence tree as a basis for lazy functional debugging. Automated Software Engineering 4 (2), 121–150.
[26] Pope, B., December 2006. A declarative debugger for Haskell. Ph.D. thesis, The University of Melbourne, Australia.
[27] Shapiro, E. Y., February 1981. Inductive inference of theories from facts. Tech. Rep. RR 192, Yale University (New Haven, CT US).
URL http://opac.inria.fr/record=b1007525
[28] Shapiro, E. Y., April 1982. Algorithmic program debugging. MIT Press.
[29] Silva, J., November 2011. A survey on algorithmic debugging strategies. Advances in Engineering Software 42 (11), 976–991.
URL http://dx.doi.org/10.1016/j.advengsoft.2011.05.024
[30] Silva, J., June 2012. A vocabulary of program slicing-based techniques. ACM Computing Surveys 44 (3).
[31] The Eclipse Foundation, 2003. Eclipse. Available at: http://www.eclipse.org/.
[32] Thompson, B., Naish, L., June 1997. A guide to the NU-Prolog debugging environment. Tech. rep., University of Melbourne.
[33] Wong, W. E., Gao, R., Li, Y., Abreu, R., Wotawa, F., August 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740.
URL http://dx.doi.org/10.1109/TSE.2016.2521368

Appendix A. Property proofs

In this appendix, we prove the results presented in Table 1. In particular, we prove for each transformation in the table whether it holds each of the eleven properties presented.

The appendix has been organized with a different section for each transformation. First, we include a section to prove that properties 1 to 5 are increasingly restrictive. That is:

$$\text{Property 1} \implies \text{Property 2} \implies \text{Property 3} \implies \text{Property 4} \implies \text{Property 5}$$

In the rest of the appendix, for the sake of clarity, we will use $\mathcal{S}$ to refer to the correctness of a node $n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle$ (i.e., correct or wrong).

Appendix A.1. Property relations

Lemma 2. *Property 1 [Buggy Node Completeness] implies Property 2 [Buggy Code Completeness].*

Proof. We prove it by showing that whenever Property 1 is satisfied then Property 2 is also satisfied. If Property 1 holds then we have that, given an ET $T = (N,E)$, its transformed version $\mathcal{T}(T) = (N',E')$, and the identity mapping M_I, $\forall n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N$, if n is a buggy node in T then n' is a buggy node in $\mathcal{T}(T)$ and $(n',n) \in M_I$. Because $n' \in N'$ is buggy in $\mathcal{T}(T)$ and $(n',n) \in M_I \rightarrow n = n'$ then we have that node n' fulfils the condition of Property 2: $\exists n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N'$, n' is a buggy node in $\mathcal{T}(T)$ and $p = p'$. When Property 2 does not hold then by modus tollendo tollens (i.e., $((P \rightarrow Q) \wedge \neg Q) \vdash \neg P$) Property 1 does not hold either. □

Lemma 3. *Property 2 [Buggy Code Completeness] implies Property 3 [Buggy Code Reduction].*

Proof. We prove it by showing that whenever Property 2 is satisfied then Property 3 is also satisfied. If Property 2 holds then we have that, given an ET $T = (N,E)$, and its transformed version $\mathcal{T}(T) = (N',E')$, $\forall n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N$, n is a buggy node in T . $\exists n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N'$, n' is a buggy node in $\mathcal{T}(T)$ and $p = p'$. Because $n' \in N'$ is buggy in $\mathcal{T}(T)$ and $p = p'$ then we have that node n' fulfils the condition of Property 3: $\exists n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N'$, n' is a buggy node in $\mathcal{T}(T)$ and $p = p'$ or $p \supset p'$. When Property 3 does not hold then by modus tollendo tollens (i.e., $((P \rightarrow Q) \wedge \neg Q) \vdash \neg P$) Property 2 does not hold either. □

Lemma 4. *Property 3 [Buggy Code Reduction] implies Property 4 [Buggy Code Semi-reduction].*

Proof. We prove it by showing that whenever Property 3 is satisfied then Property 4 is also satisfied. If Property 3 holds then we have that, given an ET $T = (N,E)$, and its transformed version $\mathcal{T}(T) = (N',E')$, $\forall n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N$, n is a buggy node in T . $\exists n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N'$, n' is a buggy node in $\mathcal{T}(T)$ and $p = p'$ or $p \supset p'$. Because $n' \in N'$ is buggy in $\mathcal{T}(T)$ then we have that node n' fulfils the condition of Property 4: $\exists n' = \langle\langle e'_1, p' \cup p'', e'_n\rangle, \mathcal{S}'\rangle \in N'$, n' is a buggy node in $\mathcal{T}(T)$, $p'' = \emptyset$ and thus p'' is correct, and either $p = p'$ or $p \supset p'$. When Property 4 does not hold then by modus tollendo tollens (i.e., $((P \rightarrow Q) \wedge \neg Q) \vdash \neg P$) Property 3 does not hold either. □

Lemma 5. *Property 4 [Buggy Code Semi-reduction] implies Property 5 [Bug Existence].*

Proof. We prove it by showing that whenever Property 4 is satisfied then Property 5 is also satisfied. If Property 4 holds then we have that, given an ET $T = (N, E)$, and its transformed version $\mathcal{T}(T) = (N', E')$, $\forall n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N$, n is a buggy node in T . $\exists n' = \langle\langle e'_1, p' \cup p'', e'_n\rangle, \mathcal{S}'\rangle \in N'$, n' is a buggy node in $\mathcal{T}(T)$, p'' is correct and either $p = p'$ or $p \supset p'$. Because $n' \in N'$ is buggy in $\mathcal{T}(T)$ then we have that node n' fulfils the condition of Property 5: $\exists n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N'$, n' is a buggy node in $\mathcal{T}(T)$. When Property 5 does not hold then by modus tollendo tollens (i.e., $((P \to Q) \wedge \neg Q) \vdash \neg P$) Property 4 does not hold either. □

Appendix A.2. Properties of Loop Expansion

In this section, we prove whether Loop Expansion [16] holds each property. To do so, we use Figure A.6 in some proofs where the ET on the right has been obtained by applying Loop Expansion to the code that obtains the ET on the left. In the ETs, all nodes labelled with $n_1, n_2, \ldots, n_7$ are nodes that are preserved in the transformation, whereas nodes n'_0 and $m_0, m_1, \ldots, m_7$ are obtained from a modified version of the code that obtains n_0. The number inside a node is the identifier of the executed method. According to Loop Expansion [16], in the code that obtains the left ET of Figure A.6, method 0 implements two nested loops where the outer loop iterates two times, the inner loop iterates three times and method 2 is invoked during the execution of the inner loop (a total of 6 times).

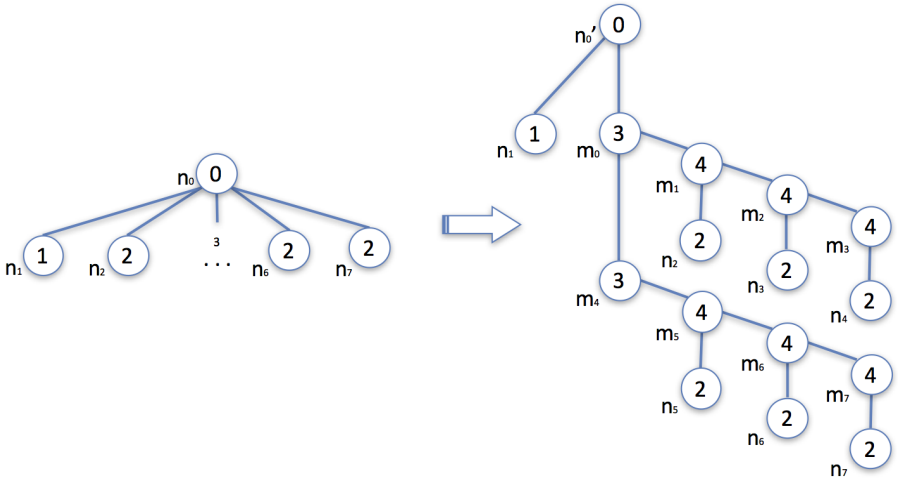

Figure A.6: Loop Expansion transformation

First of all, we need to define how Loop Expansion works according to our formalization. Loop Expansion transforms an ET $T = (N, E)$ into another ET $\mathcal{T}(T) = (N', E')$ with the identity mapping M_I and source mapping M_S. Each node $n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N$ is either preserved in $\mathcal{T}(T)$ (i.e., $\exists (n', n) \in M_I \;.\; n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N' \wedge \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle$) or it is transformed into a set of nodes Z' (i.e., $\forall n' \in Z' \;.\; (n', Z) \in M_S \wedge Z = \{n\}$) that forms a tree whose root preserves the same correctness as n (i.e., $n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in Z' \wedge n'' \in Z' \wedge (n'', n') \notin E' \wedge e_1 = e'_1 \wedge e_n = e'_n \wedge \mathcal{S} = \mathcal{S}'$) and the correctness of the rest of nodes of Z' can be either correct or wrong. Moreover, $\forall n' \in Z' \;.\; n' = \langle\langle e'_1, p' \cup p'', e'_n\rangle, \mathcal{S}'\rangle$ where $p \supset p'$ and p'' is code inserted by the transformation and it is correct by construction.

In the ET in Figure A.6, we have that $M_I = \{(n_1, n_1),\ (n_2, n_2),\ (n_3, n_3),\ (n_4, n_4),\ (n_5, n_5),\ (n_6, n_6),\ (n_7, n_7)\}$, $M_S = \{(n'_0, \{n_0\}),\ (m_0, \{n_0\}),\ (m_1, \{n_0\}),\ (m_2, \{n_0\}),\ (m_3, \{n_0\}),\ (m_4, \{n_0\}),\ (m_5, \{n_0\}),\ (m_6, \{n_0\}),\ (m_7, \{n_0\})\}$, and thus, the Z' of n_0 is $\{n'_0,\ m_0,\ m_1,\ m_2,\ m_3,\ m_4,\ m_5,\ m_6,\ m_7\}$.

Proof. [Property 1 - Buggy Node Completeness] Because Property 2 does not hold then Property 1 does not hold either as can be seen by using *modus tollendo tollens* on Lemma 2. □

Proof. [Property 2 - Buggy Code Completeness] Because Property 3 does not hold then Property 2 does not hold either as can be seen by using *modus tollendo tollens* on Lemma 3. □

Proof. [Property 3 - Buggy Code Reduction] We prove that Property 3 is not satisfied by showing a counterexample. Consider the left ET $T = (N,E)$ of Figure A.6 with node $n_0 = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N$ as wrong and the rest of nodes as correct. Node n_0 is buggy in T, but in the right ET $\mathcal{T}(T) = (N',E')$ with the source mapping M_S, $\nexists n' \in N' \,.\, n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle$ and either $p = p'$ or $p \supset p'$ because $\forall n' \in Z' \,.\, (n',Z) \in M_S \wedge Z = \{n_0\}$ we have that $n' = \langle\langle e'_1, p' \cup p'', e'_n\rangle, \mathcal{S}'\rangle$ where p'' is additional code inserted by the transformation, therefore $p \neq p' \cup p''$ and $p \not\supset p' \cup p''$. Hence, the property does not hold. □

Proof. [Property 4 - Buggy Code Semi-reduction] We prove that Property 4 is satisfied in all possible transformations. We divide the proof into two possibilities: either node $n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle$ is preserved or it is transformed. When it is preserved (i.e., $\exists (n',n) \in M_I \,.\, n' \in N'$) then it is also buggy in $\mathcal{T}(T)$ because the correctness of n' and all its children are preserved (regardless of whether they are preserved or transformed), hence it is still buggy in $\mathcal{T}(T)$. When it is transformed into Z' (i.e., $\forall n' \in Z' \,.\, (n',Z) \in M_S \wedge Z = \{n\}$) then $\exists n' = \langle\langle e'_1, p' \cup p'', e'_n\rangle, \mathcal{S}'\rangle \in Z'$ such that n' is buggy in $\mathcal{T}(T)$ because the surrounding nodes of Z' preserve their correctness (i.e., all of them are still correct regardless of whether they are preserved or transformed). But it does not matter which of them is buggy because all of them satisfy the property (i.e., p'' is correct, and either $p = p'$ or $p \supset p'$). □

Proof. [Property 5 - Bug Existence] Because Property 4 holds then Property 5 also holds as can be seen by using *modus ponendo ponens* on Lemma 5. □

Proof. [Property 6 - Keep Traceability] We prove that Property 6 is not satisfied by showing a counterexample. Consider the left ET $T = (N,E)$ of Figure A.6 where Loop Expansion transforms n_0 into a set of nodes Z'. Consider also that nodes $n_0, n_2 \in N$ are wrong and the rest of nodes in N and all nodes in Z' are correct except n'_0 which is wrong. In such a case, node n_2 is traceable in the left ET, but not in the right ET. □

Proof. [Property 7 - Zoom in Nodes] We prove that Property 7 is satisfied by showing an example. Consider the left ET $T = (N,E)$ of Figure A.6 where Loop Expansion transforms $n_0 = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N$ into a set of nodes Z' where $n'_0 = \langle\langle e'_1, p' \cup p'', e'_n\rangle, \mathcal{S}'\rangle \in Z'$. Hence, because $p \supset p'$ and p'' is correct, the property holds. □

Proof. [Property 8 - Zoom out Nodes] We prove that Property 8 is not satisfied in any possible transformation. We divide the proof into two possibilities: either node $n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle$ is preserved or it is transformed. When it is preserved (i.e., $\exists (n',n) \in M_I \,.\, n' \in N'$) then $n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle$ and $p = p'$, hence the property does not hold. When it is transformed into Z' (i.e., $\forall n' \in Z' \,.\, (n',Z) \in M_S \wedge Z = \{n\}$) then $\forall n' \in Z' \,.\, n' = \langle\langle e'_1, p' \cup p'', e'_n\rangle, \mathcal{S}'\rangle \wedge p \supset p' \wedge p''$ is correct, hence the property does not hold either. □

Proof. [Property 9 - May Hide Bugs] We prove that Property 9 is satisfied by showing an example. Consider the left ET $T = (N,E)$ of Figure A.6 where Loop Expansion transforms n_0 into a set of

nodes Z'. Consider also that node $n_0 \in N$ is wrong and the rest of nodes in N and all nodes in Z' are correct except n'_0 and m_0 which are wrong. Now consider that n_0 contains two bugs, one b_1 outside of the loop and another one b_2 inside the loop. In such a case, bug b_1 is hidden by the transformation because node n'_0 is not buggy in $\mathcal{T}(T)$. □

Proof. [Property 10 - May Reveal Bugs] We prove that Property 10 is satisfied by showing an example. Consider the left ET $T = (N,E)$ of Figure A.6 where Loop Expansion transforms n_0 into a set of nodes Z'. Consider also that nodes $n_0, n_2 \in N$ are wrong and the rest of nodes in N and all nodes in Z' are correct except n'_0 which is wrong. In such a case, $n'_0 = \langle\langle e'_1, p' \cup p'', e'_n\rangle, \mathcal{S}'\rangle$ is buggy in $\mathcal{T}(T)$ and there is a bug $b \in p'$ (p'' is correct). $n_0 = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle$ is the only node where b is contained in N because $p \supset p'$, but this node is not buggy in T and thus p is not reported as containing a bug. Hence, because there is not any other node whose code contains b in N, the property holds. □

Proof. [Property 11 - ET size] Loop Expansion either preserves each node n (i.e., $\exists(n', n) \in M_I \wedge n' \in N'$) or transforms it into a set of nodes Z' (i.e., $\forall n' \in Z' \,.\, (n', Z) \in M_S \wedge Z = \{n\}$) such that $|Z'| \geq 1$. Therefore, Loop Expansion removes one node and adds at least another node (i.e., $|N| - |\{n\}| + |Z'|$ such that $|Z'| \geq 1$). Hence, when Loop Expansion is applied, the number of nodes remains equal or greater in the transformed ET than in the original ET. □

Appendix A.3. Properties of Node Simplification

In this section, we prove whether Node Simplification [7] holds each property. To do so, we use Figure A.7 in some proofs, where the ET on the right has been obtained by applying Node Simplification to the ET on the left. In the ETs, nodes n_0, n_5, n_6 are nodes that are preserved in the transformation, whereas nodes n'_1, n'_2, n'_3, n'_4 are obtained from nodes n_1, n_2, n_3, n_4, n_5. The ET is inspired in the ET used by the authors in [7]. The technique obtains an equivalent ET where eager evaluation is simulated. Function *f/1* is evaluated in a lazy way in the left ET, whereas it is evaluated in an eager way in the right ET. Moreover, those terms that have been reduced are replaced with their reduced term (e.g., *f(3)* in nodes n_3, n_4 to *3 f(4)* in nodes n'_3, n'_4 obtained from node n_5; and *f(2)* in nodes n_1, n_2 to *2 3 f(4)* in nodes n'_1, n'_2 obtained from nodes n_3 and n_5).

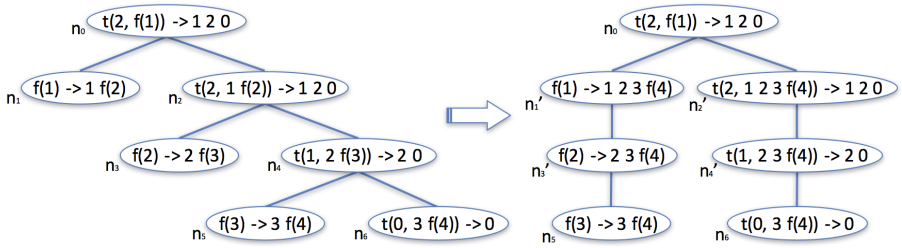

Figure A.7: Node Simplification transformation

First of all, we need to define how Node Simplification works according to our formalization. Node Simplification transforms an ET $T = (N, E)$ into another ET $\mathcal{T}(T) = (N', E')$ with the identity mapping M_I and source mapping M_S. Each node $n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N'$ is either preserved from T (i.e., $\exists (n', n) \in M_I \;.\; n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N \wedge \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle$) or it is obtained from a set of nodes Z (i.e., $\exists (n', Z) \in M_S \;.\; Z \subseteq N$), its correctness can be either correct or wrong, $\exists n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in Z$ that is the main source of n' ($n = MS_{n'}$), $e_1 \neq e'_1 \wedge p \supset p' \wedge e_n = e'_n$ or $e_1 = e'_1 \wedge p \subset p' \wedge e_n \neq e'_n$, and $\nexists n'' \in N' \;.\; (n'', n) \in M_I \vee n = MS_{n''}$. Moreover, a node removed from T is always the main source of a new node in T' (i.e., $\forall n \in N \wedge (n', n) \notin M_I \;.\; n', n'' \in N' \wedge (n'', Z) \in M_S \wedge Z \subseteq N \wedge n \in Z \wedge n = MS_{n''}$).

In the ET in Figure A.7, we have that $M_I = \{(n_0, n_0),\ (n_5, n_5),\ (n_6, n_6)\}$, and $M_S = \{(n'_1,\ \{n_1, n_3, n_5\}),\ (n'_3,\ \{n_3, n_5\}),\ (n'_2,\ \{n_2, n_3, n_5\}),\ (n'_4,\ \{n_4, n_5\})\}$, where $n_1 = MS_{n'_1}$, $n_2 = MS_{n'_2}$, $n_3 = MS_{n'_3}$, $n_4 = MS_{n'_4}$.

Proof. [Property 1 - Buggy Node Completeness] Because Property 2 does not hold then Property 1 does not hold either as can be seen by using *modus tollendo tollens* on Lemma 2. □

Proof. [Property 2 - Buggy Code Completeness] Because Property 3 does not hold then Property 2 does not hold either as can be seen by using *modus tollendo tollens* on Lemma 3. □

Proof. [Property 3 - Buggy Code Reduction] Because Property 4 does not hold then Property 3 does not hold either as can be seen by using *modus tollendo tollens* on Lemma 4. □

Proof. [Property 4 - Buggy Code Semi-reduction] We prove that Property 4 is not satisfied by showing a counterexample. Consider the left ET $T = (N, E)$ of Figure A.7 with node $n_1 \in N$ as

wrong and the rest of nodes as correct. Consider also that node $n'_1 \in N'$ is the only node that comes from n_1 (i.e., $\forall (n', Z) \in M_S \wedge n_1 \in Z \,.\, n' = n'_1$). Hence, because $p \subset p'$, the property does not hold. □

Proof. [Property 5 - Bug Existence] We prove that Property 5 is satisfied by contradiction. The only case where there is a buggy node $n = \langle \langle e_1, p, e_n \rangle, \mathcal{S} \rangle$ in T but there is not any buggy node in $\mathcal{T}(T)$ is when n is wrong in T but all nodes in $\mathcal{T}(T)$ are correct. Node n can be either maintained (i.e., $(n,n) \in M_I$) or replaced with $n' = \langle \langle e'_1, p', e'_n \rangle, \mathcal{S}' \rangle$ (i.e., $n = MS_{n'}$). If it is maintained then the state is maintained too, and therefore it is also wrong in $\mathcal{T}(T)$. When it is replaced, then it is possible that n is wrong and n' is correct, but therefore (as the contradiction reveals) there must be another wrong node in $\mathcal{T}(T)$. In the replacing case, we know that either $e_1 \neq e'_1 \wedge p \supset p' \wedge e_n = e'_n$ or $e_1 = e'_1 \wedge p \subset p' \wedge e_n \neq e'_n$. In the former (i.e., $e_1 \neq e'_1 \wedge p \supset p' \wedge e_n = e'_n$), the remaining code (i.e., $p \backslash p'$) is the code that transforms e_1 into e'_1. If n' is correct, then the only part in n that can manifest a bug is the generation of e'_1 (because the rest is proven correct as can be seen in n'). But here, the node in $\mathcal{T}(T)$ that transforms e_1 into e'_1 must be wrong. In the latter (i.e., $e_1 = e'_1 \wedge p \subset p' \wedge e_n \neq e'_n$), the surplus code (i.e., $p' \backslash p$) is the code that transforms e_n into e'_n. If n' is correct, then the only part in n that can manifest a bug is the generation of e'_n (because the rest is proven correct as can be seen in n'). But here again, the node in $\mathcal{T}(T)$ that transforms e_n into e'_n must be wrong. Therefore, it is impossible that all nodes in $\mathcal{T}(T)$ are correct. Because at least one node must be wrong in $\mathcal{T}(T)$, then trivially there is at least one buggy node in $\mathcal{T}(T)$. □

Proof. [Property 6 - Keep Traceability] We prove that Property 6 is not satisfied by showing a counterexample. Consider the left ET $T = (N, E)$ of Figure A.7 with node $n_1 \in N$ as wrong and the rest of nodes as correct. In such a case, node n_1 is buggy in T, but $\nexists (n', n_1) \in M_I \,.\, n' \in N'$. Hence, the property does not hold. □

Proof. [Property 7 - Zoom in Nodes] We prove that Property 7 is satisfied by showing an example. Consider the left ET $T = (N, E)$ of Figure A.7. Consider also that node $n'_2 \in N'$ is the only node that comes from n_2 (i.e., $\forall (n', Z) \in M_S \wedge n_2 \in Z \,.\, n' = n'_2$). Hence, because $p \supset p'$, the property holds. □

Proof. [Property 8 - Zoom out Nodes] We prove that Property 8 is satisfied by showing an example. Consider the left ET $T = (N, E)$ of Figure A.7. Consider also that node $n'_1 \in N'$ is the only node that comes from n_1 (i.e., $\forall (n', Z) \in M_S \wedge n_1 \in Z \,.\, n' = n'_1$). Hence, because $p \subset p'$, the property holds. □

Proof. [Property 9 - May Hide Bugs] We prove that Property 9 is satisfied by showing an example. Consider the left ET $T = (N, E)$ of Figure A.7. Consider also that n_3 is the only node that executes any part of its p and should have obtained *4 f(3)*, and that node n_5 should have obtained *9 f(4)* by executing another code p'. Node n_3 is buggy in T and thus there is a bug $b \in p$, whereas n_5 is the only buggy node in the right ET. Hence, because $b \notin p'$, the property holds. □

Proof. [Property 10 - May Reveal Bugs] We prove that Property 10 is satisfied by showing an example. Consider the left ET $T = (N, E)$ of Figure A.7. Consider also that all nodes are correct

except n_4 and n_5 that should have obtained *2* and *9 f(4)*, respectively. In such a case, node n'_4 that executes p' should obtain *2* as well, and thus there exists a bug $b \in p'$ because n'_4 is buggy in the right ET. However, b is not detectable in T because n_5 is the only buggy node in T. Hence, the property holds. □

Proof. [Property 11 - ET size] Node Simplification either preserves each node n' (i.e., $\exists(n', n) \in M_I \wedge n \in N$) or it is obtained from a set of nodes Z (i.e., $\exists(n', Z) \in M_S \ . \ Z \subseteq N$) such that $\exists n \in Z \ . \ n = MS_{n'}$ and $\nexists n'' \in N' \ . \ (n'', n) \in M_I \vee n = MS_{n''}$. Therefore, either a node n' is preserved from T or it is created by removing n, which cannot be the main source of another node of N' (i.e., $\nexists n'' \in N' \ . \ n = MS_{n''}$). Hence, because each node in T' is either preserved from T or it is created by removing a node from T, and a node from T cannot be removed without creating a node in T', then the amount of nodes is always maintained. □

Appendix A.4. Properties of Tree Balancing

In this section, we prove whether Tree Balancing [15] holds each property. To do so, we use Figures A.8 and A.9 in some proofs, where the ETs on the right have been obtained by applying Tree Balancing to the ETs on the left. In the ETs, all nodes labelled with $n_0, n_1, \ldots, n_j$ are nodes that are preserved in the transformation, whereas nodes labelled with m_0 are new nodes added by the technique from n_1, n_2, n_4 in Figure A.8 and from n_1, n_2, n_3 in Figure A.9. The number inside a node is the identifier of the executed method.

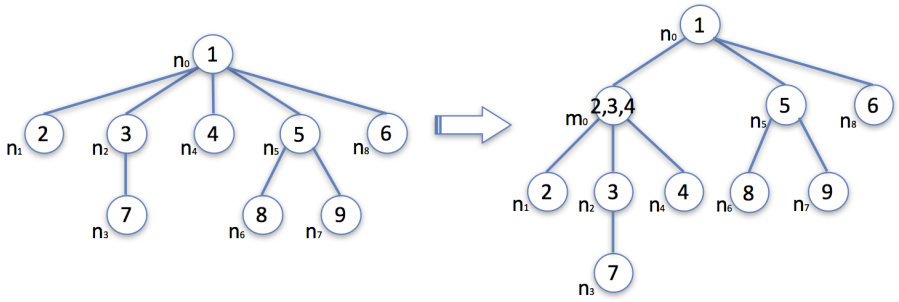

Figure A.8: Tree Balancing - Projection transformation

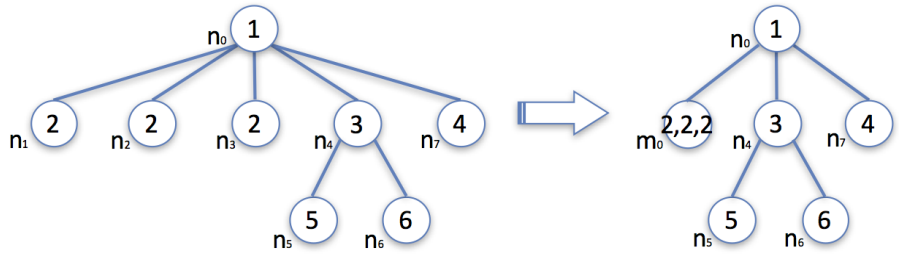

Figure A.9: Tree Balancing - Collapse transformation

First of all, we need to define how Tree Balancing works according to our formalization. Tree Balancing transforms an ET $T = (N, E)$ into another ET $\mathcal{T}(T) = (N', E')$ with the identity mapping M_I and source mapping M_S. Each node $n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N$ is either preserved in $\mathcal{T}(T)$ (i.e., $\exists (n', n) \in M_I \;.\; n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N' \wedge \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle$), preserved and projected into another node n', or just collapsed into another node n' (i.e., $\exists (n', Z) \in M_S \;.\; Z \subseteq N \wedge n \in Z$) where the correctness of n' can be either wrong (when a bug in a projected/collapsed node produces a wrong final context in the projection/collapse node) or correct otherwise.

In the ET in Figure A.8, we have that $M_I = \{(n_0, n_0),\ (n_1, n_1),\ (n_2, n_2),\ (n_3, n_3),\ (n_4, n_4),\ (n_5, n_5),\ (n_6, n_6),\ (n_7, n_7),\ (n_8, n_8)\}$ and $M_S = \{(m_0, \{n_1, n_2, n_4\})\}$. In the ET in Figure A.9 we have that $M_I = \{(n_0, n_0),\ (n_4, n_4),\ (n_5, n_5),\ (n_6, n_6),\ (n_7, n_7)\}$ and $M_S = \{(m_0, \{n_1, n_2, n_3\})\}$.

Proof. [Property 1 - Buggy Node Completeness] Because Property 2 does not hold then Property 1 does not hold either as can be seen by using *modus tollendo tollens* on Lemma 2. □

Proof. [Property 2 - Buggy Code Completeness] Because Property 3 does not hold then Property 2 does not hold either as can be seen by using *modus tollendo tollens* on Lemma 3. □

Proof. [Property 3 - Buggy Code Reduction] Because Property 4 does not hold then Property 3 does not hold either as can be seen by using *modus tollendo tollens* on Lemma 4. □

Proof. [Property 4 - Buggy Code Semi-reduction] Because Property 5 does not hold then Property 4 does not hold either as can be seen by using *modus tollendo tollens* on Lemma 5. □

Proof. [Property 5 - Bug Existence] We prove that Property 5 is not satisfied by showing a counterexample. Consider the left ET $T = (N,E)$ of Figure A.9 with node $n_1 \in N$ as wrong and the rest of nodes as correct. Consider that n_1 does not produce a wrong final context in the collapse node m_0. In such a case, node m_0 is correct and therefore $\nexists n' \in N'$ such that n' is buggy in $\mathcal{T}(T)$. Hence, the property does not hold. □

Proof. [Property 6 - Keep Traceability] We prove that Property 6 is not satisfied by showing a counterexample. Consider the left ET $T = (N,E)$ of Figure A.9 with nodes $n_0, n_1 \in N$ as wrong and the rest of nodes as correct. Node n_1 is traceable in the left ET, but $\nexists (n', n_1) \in M_I \;.\; n' \in N'$. Hence, the property does not hold. □

Proof. [Property 7 - Zoom in Nodes] We prove that Property 7 is not satisfied in any possible transformation. We divide the proof into three possibilities: either node $n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N'$ is preserved from T, it is a collapse node created from Z, or it is a projection node created from Z. When it is preserved (i.e., $\exists (n', n) \in M_I \;.\; n \in N$) then $n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle$ and $p = p'$, hence the property does not hold. When it is a collapse node (i.e., $\exists (n', Z) \in M_S \;.\; Z \subseteq N$) then $\forall n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in Z \;.\; p = p'$, hence the property does not hold either. When it is a projection node (i.e., $\exists (n', Z) \in M_S \;.\; Z \subseteq N$) then $p' = \bigcup \{p \mid n \in Z \wedge n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle\}$, and thus $p \subseteq p'$, hence the property does not hold either. □

Proof. [Property 8 - Zoom out Nodes] We prove that Property 8 is satisfied by showing an example. Consider the left ET $T = (N,E)$ of Figure A.8 where Tree Balancing creates the projection node m_0 from $n_1, n_2, n_4 \in N$ (i.e., $(m_0, Z) \in M_S \wedge Z = \{n_1, n_2, n_4\}$). In such a case, $n_1 = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle$, $n_2 = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle$, $n_4 = \langle\langle e''_1, p'', e''_n\rangle, \mathcal{S}''\rangle$ and thus $m_0 = \langle\langle e'''_1, p''', e'''_n\rangle, \mathcal{S}'''\rangle$ where $p''' = p \cup p' \cup p''$. Hence, because $p \subset p'''$, the property holds. □

Proof. [Property 9 - May Hide Bugs] We prove that Property 9 is satisfied by showing an example. Consider the left ET $T = (N,E)$ of Figure A.9 with node $n_1 \in N$ as wrong and the rest of nodes as correct. Consider also that the collapse node is correct. In such a case, n_1 is buggy in T but $\nexists n' \in N'$ such that n' is buggy in $\mathcal{T}(T)$. Hence, the property holds. □

Proof. [Property 10 - May Reveal Bugs] We prove that Property 10 is satisfied by showing an example. Consider the left ET $T = (N,E)$ of Figure A.8 with nodes $n_0, n_1 \in N$ as wrong and the rest of nodes as correct. Consider also that the projection node is correct. In such a case, $n_0 = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N'$ is buggy in $\mathcal{T}(T)$ and thus there is a bug $b \in p$, but b cannot be detectable in T because n_0 is not buggy in T. Hence, because there is not any other node whose code contains b, the property holds. □

Proof. [Property 11 - ET size] Tree Balancing either preserves each node n' from T (i.e., $\exists(n', n) \in M_I \ . \ n \in N$), or creates a collapse/projection node n' from Z (i.e., $\exists(n', Z) \in M_S \ . \ Z \subseteq N$) by removing or not Z, respectively. Therefore, Tree Balancing can either decrease the amount of nodes (e.g., no projection nodes are inserted and two nodes are collapsed into a collapse node), maintain the amount of nodes (e.g., one projection node is inserted and two nodes are collapsed into a collapse node), or increase the amount of nodes (e.g., one projection node is inserted and no collapse nodes are created). □

Appendix A.5. Properties of Tree Compression

In this section, we prove whether Tree Compression [9] holds each property. To do so, we use Figure A.10 in some proofs, where the ET on the right has been obtained by applying Tree Compression to the ET on the left. In the ETs, all nodes labelled with $n_0, n_1, \ldots, n_j$ are nodes that are preserved in the transformation. The number inside a node is the identifier of the executed method. In the code of Figure A.10 there are three recursive calls: one in method 1, another in method 4 and a multiple recursion in method 6.

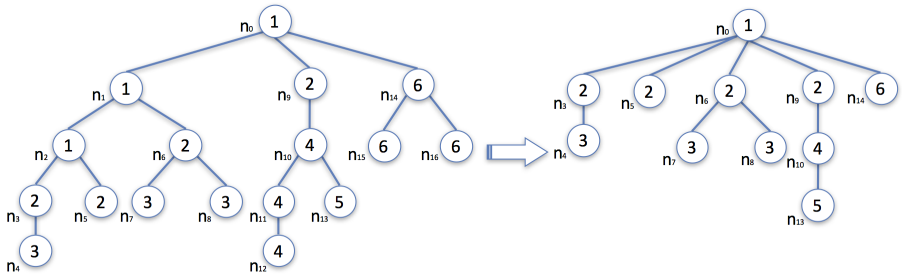

Figure A.10: Tree Compression transformation

First of all, we need to define how Tree Compression works according to our formalization. Tree Compression transforms an ET $T = (N,E)$ into another ET $\mathcal{T}(T) = (N',E')$ with the identity mapping M_I. Each node $n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N$ is either preserved in $\mathcal{T}(T)$ (i.e., $\exists (n',n) \in M_I \;.\; n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N' \wedge \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle$) or removed (i.e., $\nexists (n',n) \in M_I \;.\; n' \in N'$).

In the ET in Figure A.10, we have that $M_I = \{(n_0,n_0),\ (n_3,n_3),\ (n_4,n_4),\ (n_5,n_5),\ (n_6,n_6),\ (n_7,n_7),\ (n_8,n_8),\ (n_9,n_9),\ (n_{10},n_{10}),\ (n_{13},n_{13}),\ (n_{14},n_{14})\}$.

Proof. [Property 1 - Buggy Node Completeness] Because Property 2 does not hold then Property 1 does not hold either as can be seen by using *modus tollendo tollens* on Lemma 2. □

Proof. [Property 2 - Buggy Code Completeness] Because Property 3 does not hold then Property 2 does not hold either as can be seen by using *modus tollendo tollens* on Lemma 3. □

Proof. [Property 3 - Buggy Code Reduction] Because Property 4 does not hold then Property 3 does not hold either as can be seen by using *modus tollendo tollens* on Lemma 4. □

Proof. [Property 4 - Buggy Code Semi-reduction] Because Property 5 does not hold then Property 4 does not hold either as can be seen by using *modus tollendo tollens* on Lemma 5. □

Proof. [Property 5 - Bug Existence] We prove that Property 5 is not satisfied by showing a counterexample. Consider the left ET $T = (N,E)$ of Figure A.10 with node $n_2 \in N$ as wrong and the rest of nodes as correct. In such a case, node n_2 is buggy in T, but in the right ET $\mathcal{T}(T) = (N',E')$, $\nexists n' \in N'$ such that n' is buggy in $\mathcal{T}(T)$. Hence, the property does not hold. □

Proof. [Property 6 - Keep Traceability] We prove that Property 6 is not satisfied by showing a counterexample. Consider the left ET of Figure A.10 with nodes n_0, n_1 and n_2 as wrong and the rest

of nodes as correct. In such a case, node n_2 is traceable in the left ET, but $\nexists(n', n_2) \in M_I \,.\, n' \in N'$. Hence, the property does not hold. □

Proof. [Property 7 - Zoom in Nodes] We prove that Property 7 is not satisfied in any possible transformation. There is only one possibility where this can occur: node $n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle$ is preserved. When it is preserved (i.e., $\exists(n', n) \in M_I \,.\, n' \in N'$) then $n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle$ and $p = p'$. Hence, the property does not hold. □

Proof. [Property 8 - Zoom out Nodes] We prove that Property 8 is not satisfied in any possible transformation. There is only one possibility where this can occur: node $n = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle$ is preserved. When it is preserved (i.e., $\exists(n', n) \in M_I \,.\, n' \in N'$) then $n' = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle$ and $p = p'$. Hence, the property does not hold. □

Proof. [Property 9 - May Hide Bugs] We prove that Property 9 is satisfied by showing an example. Consider the left ET $T = (N, E)$ of Figure A.10 with node $n_2 \in N$ as wrong and the rest of nodes as correct. In such a case, $n_2 \in N$ is buggy in T but $\nexists n' \in N'$ such that n' is buggy in $\mathcal{T}(T)$. Hence, the property holds. □

Proof. [Property 10 - May Reveal Bugs] We prove that Property 10 is satisfied by showing an example. Consider the left ET $T = (N, E)$ of Figure A.10 with nodes $n_0, n_1 \in N$ as wrong and the rest of nodes as correct. In such a case, $n_0 = \langle\langle e'_1, p', e'_n\rangle, \mathcal{S}'\rangle \in N'$ is buggy in $\mathcal{T}(T)$ with a bug $b \in p'$ and $n_1 = \langle\langle e_1, p, e_n\rangle, \mathcal{S}\rangle \in N$ is buggy in T with $p' \supseteq p$. Hence, because it is possible that $b \in p' \setminus p$, the property holds. □

Proof. [Property 11 - ET size] Tree Compression either preserves each node n (i.e., $\exists(n', n) \in M_I \,.\, n' \in N'$) or removes it (i.e., $\nexists(n', n) \in M_I \,.\, n' \in N'$). Hence, when Tree Compression is applied, the number of nodes is always reduced. □