

# How Difficult and Effective is Writing Assertions for Observing Bugs at Runtime?

Fischer Ferreira<sup>1</sup>, Eduardo Fernandes<sup>2</sup>, Johnatan Oliveira<sup>1</sup>,  
Maurício Souza<sup>3</sup>, and Eduardo Figueiredo<sup>1</sup>

<sup>1</sup> Federal University of Minas Gerais (UFMG)

<sup>2</sup> Pontifical Catholic University of Rio de Janeiro (PUC-Rio)

<sup>3</sup> Federal University of Lavras (UFLA)

{fischerjf, johnatan.si, mrasouza, figueiredo}@dcc.ufmg.br,  
emfernandes@inf.puc-rio.br

**Abstract.** *Context:* Executable Assertion (EA) is a boolean expression written to assess the conformance of the program behavior with its requirements. Contrary to unit test cases, EA supports the bug detection at runtime. An advantage of adopting EA is capturing the fine-grained bug location. *Objective:* Writing EA instances can be expensive and time consuming, due to the need for understanding in depth the program structure and invariants. Unfortunately, there is no empirical evidence on how difficult and effective is writing EA instances. This paper fills this literature gap with a quasi-experiment. *Method:* We asked 22 participants to write EA instances for seven AVL Tree properties. We assessed i) the time spent to write each instance, ii) the difficulty faced by the participants, and iii) the correctness of each instance. Finally, we assessed how effective are the EA instances written by the participants in observing bugs. We relied on the analysis of 155 bugs generated via mutation testing. *Results:* Participants spent from 5.7 to 13.2 minutes in average to write each EA instance; they reported an average difficulty from 2.2 to four out of 5. In average, writing EA instances for three out of the seven AVL Tree properties was sufficient to observe 57% of the bugs. *Conclusions:* Our results suggest that writing assertions for observing bugs at runtime is feasible while quite effective, but it requires reasonable effort.

**Keywords:** program bug, executable assertion, runtime debugging, mutation testing, quasi-experiment.

## 1 Introduction

Along with the life cycle of software programs, developers may unexpectedly introduce bugs [28]. A *program bug* is a non-conformance between the expected and the observed behavior of a program [14]. The expected behavior of a program is typically characterized by the software requirements [19]. For instance, the AVL Tree is a traditional data structure designed at the binary search [25]. A major AVL Tree requirement is that inserting nodes should keep balanced the height of both left and right sub-trees for any node [25]. Thus, a bug would be

observing an unbalanced height caused by a node insertion. Bugs can ultimately compromise the experience of stakeholders while using a software program [14]. Therefore, bugs should be detected, reported, and fixed as soon as possible [28].

*Program debugging* is the process of analyzing the occurrence of bugs [28]. Debugging has three major tasks: i) bug detection, ii) bug report, and iii) bug fixing [2]. *Bug detection* means detecting a bug instance in the source code. *Bug report* consists of reporting the location of the detected bug instance. *Bug fixing* means to fix the program behavior implemented by the bug-affected code. Debugging is far from being trivial. Particularly, tracking the exact location of a bug based on the respective bug report is challenging [28]. Through the use of traditional unit test cases, developers typically have a coarse-grained location, e.g. at the method level [28]. However, these unit test cases say little about the exact bug location in a fine-grained level, e.g. at the code statement level.

*Executable Assertion* (EA) [10] aims at a more fine-grained bug report [1, 10, 12]. An EA is a boolean expression written to assess the conformance of the program behavior with its requirements [12, 24]. Writing EA instances means instrumenting, i.e. equipping, the software program with a verification layer [12]. This verification layer is designed to observe the program behavior at runtime, targeting specific parts of the source code such as code statements [12]. Different of unit test cases, EA detects bugs at runtime in a fine-grained level, thereby reporting those code statements affected by bugs. The drawback of adopting EA is the need for deeply understanding the code properties and invariants, which may be expensive and time consuming [1]. Such understanding is require to write effective EA instances. Unfortunately, there is still no empirical study aimed at investigating how difficult and effective writing EA instances can be.

In this paper, we partially address the aforementioned literature gap with an empirical study. We carefully designed and performed a quasi-experiment [26] with 22 students enrolled in undergraduate and graduate courses of Computer Science and related areas. In a laboratory environment, we asked the participants to write EA instances for seven AVL Tree properties. We then assessed time spent, difficulty, and correctness of the written EA instances. Additionally, we assessed the effectiveness of these instances in detecting bugs. For this purpose, we relied on the analysis of 155 bugs automatically generated via mutation testing [14]. Mutant versions of the AVL Tree implementation were generated with MuCclipse [20]. Each mutation version is affected by a bug instance, which represents a recurring bug introduced by real developers [14].

Our study results provided us with a first comprehension on how difficult and effective can be writing EA instances. Participants spent from 5.7 to 13.2 minutes in average to write each EA instance. Additionally, participants have reported an average difficulty from 2.2. to four out of five. These results regard experienced participants in AVL Tree without expertise in EA writing. Finally, in average, writing EA instances for three out of the seven AVL Tree properties was sufficient to observe 57% of the bugs. The conclusion is that, in our quasi-experiment, writing assertions for observing bugs at runtime showed feasible while quite effective. However, writing these assertions requires reasonable effort.

## 2 Background

### 2.1 AVL Tree

As discussed in Section 1, we selected the AVL Tree for supporting our quasi-experiment regarding the difficulty and effectiveness of writing EA instances. We decided to use the AVL Tree for a few reasons: i) this is a traditional data structure designed at the binary search [4, 25], ii) students are quite familiar with AVL Tree properties and implementation; iii) the AVL Tree implementation is sufficiently short and complex to be used for experimentation along a few hours. Table 1 lists seven invariant properties of the AVL Tree, which we extracted from a previous work [4] and selected for the quasi-experiment. *Invariant properties* are properties of that should always be preserved along with the use of the data structure. Due to time constraints for performing the quasi-experiment, we discarded two other properties regarding: i) the tree chaining in terms of child and father nodes and ii) the tree leaves. The AVL Tree properties are too complex to implement in a few hours.

**Table 1.** AVL Tree Invariant Properties Used in the Quasi-Experiment

ID	Property	Description
P1	Tree without elements	If the number of elements in a tree is zero then the root should be null
P2	Tree with only one element	If the root is non-null and the root height is zero then the left and right children must be null
P3	Tree with more than one element	If the root is non-null and the root height is greater than or equal to one then the left, right, or both branches should be different from null
P4	Binary search tree	For every element that belongs to the binary search tree the child on the left must be smaller than his father, and the son on the right must be bigger than your father
P5	Number of children of a node	For every element belonging to the binary search tree, you can have only one left a direct child and one right direct child
P6	Tree balancing	For every balanced tree, the height of the left node minus the height of the right node must be less than or equal to one unit
P7	Tree height	For every binary search tree, the total height must be less than $1.44 * \log(n + 2) - 1.328$

### 2.2 AVL Tree Instrumentation

Writing EA instances implies instrumenting, i.e. equipping, the software program with a verification layer able to observe bugs at runtime [1]. In a previous work, we have introduced a systematic model for instrumenting the source code of a program aimed at incorporating EA [10]. Our model is constituted of the following steps: i) *to extend* the original class we aim at instrumenting with EA instances; ii) in the child class, *to write* each EA instance targeting an invariant property of the original class into a private method; iii) *to create* a public method called `verifier` responsible for calling all private methods that implement an EA instance; iv) *to instantiate* an object of child class for calling the `verifier` method and, therefore, triggering the written EA instances at runtime.

Aimed to support of quasi-experiment, we have provided each participant with a Java implementation of the AVL Tree program based on a previous work [25]. For the sake of simplicity, we have prepared the program to partially instrument the AVL Tree. Thus, the participant should be concerned only

with the EA writing. The program was organized in two packages and seven classes. The main package, called `AVLTree`, has five classes that implement the AVL Tree data structure. The `AVLInstrumentada` class was designed for the participants to write each EA instance. Listing 1 depicts an example of a EA instance targeting the property P6 – Tree balancing (see Table 1).

```

1. private boolean verifyTreeBalancing(AvlNode t) {
2.     if (t != null) {
3.         verifyTreeBalancing(t.left);
4.         if (t.left != null && t.right != null) {
5.             try { if ((t.left.height - t.right.height) > 1) throw new IllegalStructureException(); }
6.             catch (IllegalStructureException e) e.printStackTrace(); }
7.         verifyTreeBalancing(t.right); }
8.     return true;
9. }

```

**Listing 1.** An Example of a Written EA Instance

### 2.3 Unit Test Cases *versus* Executable Assertions

We discuss below the practical difference between adopting unit test cases and EA. Let us assume that one method implements the node insertions into an AVL Tree. A typical unit test case would be written to assess the whole method rather than its statements. Thus, the test case could report eventual bugs at the method level, without further details on the code statement that originated the bug. Consequently, the developer in charge of fixing the bug should understand the method’s source code. The bug could be related to any AVL Tree property, including the height balance. In this case, writing one EA for each AVL Tree property could help to locate the bug. When observing the bug, an EA instance would provide the developers with the exact bug location.

Although EA is quite promising in supporting the bug detection with a fine-grained bug report, writing EA can be expensive and time consuming [1]. In the AVL Tree example, many invariant properties (Table 1) should be deeply understood by the developer. Some properties, such as the tree self-balancing, are not trivial to verify (see Listing 1). More critically, different EA instances should be written to assess each property, which is time consuming. For instance, self-balancing should be assessed for both node insertions and deletions. This study aims at exploring both time spent and difficulty in writing EA instances.

### 2.4 Mutation Testing

Mutation testing is an approach for enhancing the quality of test suites [2, 27]. This approach typically relies on the automatic generation of mutant versions for a given software program. Each mutant version is a similar program with punctual changes, e.g. at the code statement or method levels, applied to the original code structure. These changes represent bugs commonly introduced by real developers along with the life cycle of their programs [13]. The set of mutation versions is submitted to verification by a test suite. The higher is the percentage of bugs captured by the test suite, the higher is the test suite quality [17, 19]. A mutant version is killed whenever the test suite is able to observe the bug implemented by the mutant; otherwise, we say the mutant is alive [5].

### 3 Study Design

We relied on strict empirical guidelines [3, 26] for Software Engineering research in this work. We describe below our goal and research questions (Section 3.1), steps and artifacts (Section 3.2), and a participant overview (Section 3.3).

#### 3.1 Goal and Research Questions

Based on the Goal Question Metric template [3], we systematically defined our study goal as follows: *analyze* EA instances written by participants to observe program bugs at runtime; *for the purpose of* empirically assessing how difficult and effective is writing EA instances; *with respect to* i) time spent by participants to write EA instances, ii) difficulty reported by the participants to write the EA instances, iii) correctness of these instances, and iv) effectiveness of EA instances in observing bugs; *from the viewpoint of* software developers and researchers with expertise in software testing; *in the context of* students enrolled in either undergraduate or graduate courses in Computer Science and related areas. We defined four research questions (RQs) described below.

**RQ<sub>1</sub>:** *How long do participants take to complete the writing of EA instances?* –For agile development teams, time to deliver software programs is typically scarce [23]. Consequently, participants are often forced to prioritize other development tasks, e.g. the addition of new program features [6, 15], rather than testing their programs. Thus, it is desirable that bug detection techniques will not be time consuming to use. We aim at understanding how much time participants usually spent to write an EA instance. As any other lightweight formal method, EA writing can be too costly for practical recommendation. With **RQ<sub>1</sub>**, we expect to acquire a first understanding on the time spent by participants experienced in AVL Tree properties but without experience in EA writing.

**RQ<sub>2</sub>:** *How difficult is writing EA instances?* –Understanding the difficulty of writing EA instances is important to characterize the complexity of this task. Indeed, unit testing is quite popular in industry [22] and participants are used to write unit test cases rather than EA instances. Through **RQ<sub>2</sub>**, we aim at understanding, in a Likert scale [18] from one to five, how difficult is writing EA instances. If our quasi-experiment participants are familiar with unit test cases but still find it easy to write EA instances, then adopting EA may be feasible.

**RQ<sub>3</sub>:** *How often do participants write EA instances that correctly implement the AVL Tree properties?* – We designed our quasi-experiment to be performed by participants with no previous experience with writing EA instances. It may be the case that this task is difficult in such a way participants cannot implement correct EA instances. **RQ<sub>3</sub>** was ultimately designed to support our next RQ.

**RQ<sub>4</sub>:** *How effective are the EA instances written by participants in detecting bugs?* – As important as understanding the difficulty of writing EA instances is assessing how effective these instances are in detecting bugs. Indeed, the major purpose of this technique is observing bug instances in programs. Through **RQ<sub>4</sub>**, we assess the average percentage of bugs detected by the number of EA

instances implemented by the participants. We expect to understand how many EA instances are necessary to detect a reasonable amount of bugs in a program.

### 3.2 Steps and Artifacts

The study artifacts are available in the companion research website [9].

**Step 1: *Select Target Program*** – As discussed in Section 2.1, we have selected an AVL Tree implementation to support our quasi-experiment. Thus, we expected to provide the participants with a program that is sufficiently short and complex for reasoning about and writing EA instances.

**Step 2: *Prepare Forms*** – We have defined three forms aimed at collecting all data for the quasi-experiment. *Consent Form* aims at collecting the participant consent to allow us both collecting and analyzing the quasi-experiment data. *Background Form* has six questions on the participant expertise with Software Engineering techniques, e.g. object-oriented programming and EA. *Experiment Form* describes the seven AVL Tree properties to be implemented as EA instances in up to one hour in total. For each property, we ask the participant to inform: i) start time, finish time, and EA writing difficulty in a Likert scale [18] from one, i.e. the lowest difficulty, to five, i.e., the highest difficulty or impossibility to write the EA instance. We arranged the seven AVL properties of Table 1 from the easiest to the hardest one from our opinion.

**Step 3: *Run Pilot Quasi-Experiment*** – We have run a pilot version of our quasi-experiment with four participants in order to shape our study design and artifacts. This pilot version helped us to refine the artifacts and the total quasi-experiment time. We carefully discarded the results provided by the pilot study participants, thereby preventing biases in our study results.

**Step 4: *Recruit Quasi-Experiment Participants*** – We have recruited 33 undergraduate and graduate students to participate in the quasi-experiment. These participants were enrolled in either undergraduate or graduate courses at the Federal University of Minas Gerais (UFMG). We further filtered these participants according to some important criteria described in Section 3.3.

**Step 5: *Instruct Participants on the Quasi-Experiment*** – All participants were accommodated in a laboratory environment. We introduced the study goal without further details that could affect the experiment with result biases. After that, we briefly instructed the participants on the basics of EA (definition and application) and the AVL Tree properties. Finally, we instructed the participants regarding the quasi-experiment procedures, especially on how to instrument the AVL Tree program to implement the EA instances (cf. Section 2.2).

**Step 6: *Collect Participant Consent and Background*** – We asked each participants to sign the *Consent Form*, thereby allowing us to anonymously collect and analyze the quasi-experiment data. Additionally, each participant has filled out the *Background Form*.

**Step 7: *Allocate Participants to Computers*** – We have allocated each participant to an individual desktop computer equipped with: the Eclipse IDE for reading and writing code; the source code of the AVL Tree program (Section 2.2) already imported into the IDE; and the formal definition of seven AVL Tree

properties listed in (Table 1) for which the participants will implement the EA instances. For illustration purposes, the property P6 – Tree balancing is formalized as follows:  $\forall n \in tree: ((n \rightarrow left \neq null) \ \&\& \ (n \rightarrow right \neq null)) \Rightarrow ((n \rightarrow left \rightarrow height) - (n \rightarrow right \rightarrow height)) \leq 1$ , where  $n$  is an arbitrary tree node,  $\rightarrow left$  (or  $\rightarrow right$ ) is the left (or right) child of the current node, and  $\rightarrow height$  is the tree height from the current node.

**Step 8: Run the Quasi-Experiment** – All participants had up to one hour for completing the quasi-experiment while filling the *Experiment Form* (details in **Step 2**). We answered eventual questions made by the participants along with the quasi-experiment execution. Nevertheless, we carefully avoided biasing the participants’ responses with our answers.

**Step 9: Compute Mutant Versions of the AVL Tree Program** – We used MuClipse<sup>4</sup>, an Eclipse IDE plugin, to generate the mutant versions of the Java-based AVL Tree implementation (Section 2.2). MuClipse supports code changes at both class level – e.g., changes affecting the class coupling or inheritance – and method level – e.g., changes affecting the arithmetical and logical operations. We obtained 155 mutant versions that actually differ from the original program.

**Step 10: Submit Written EA Instances to Mutant Testing** – We used MuClipse to perform the mutation testing on the set of EA instances written by the quasi-experiment participants. Our goal was understanding the effectiveness of these participant-written instances in observing bugs. We set MuClipse to run with its maximal run time available. Thus, we expected to prevent that mutant versions of the program are not killed due to run time exception.

### 3.3 Participant Overview

A total of 11 out of the 33 participants were discarded from the study because: four participants engaged in the pilot study; four participants failed to implement EA instances to all seven AVL tree properties; one participant provided us with unreliable data; and two participants did not send us the code of the implemented EA instances by the end of the quasi-experiment. In the end, 22 participants remained for the data analysis and report. The Background Form revealed the following about the participants’ expertise. The majority of participants (86.4%) have professional development experience. Additionally, 95.5% of the participants reported some experience with Java programming; the same percentage applied to object-oriented programming. Less than a half (40.9%) of the participants had previous knowledge about EA in theory or practice.

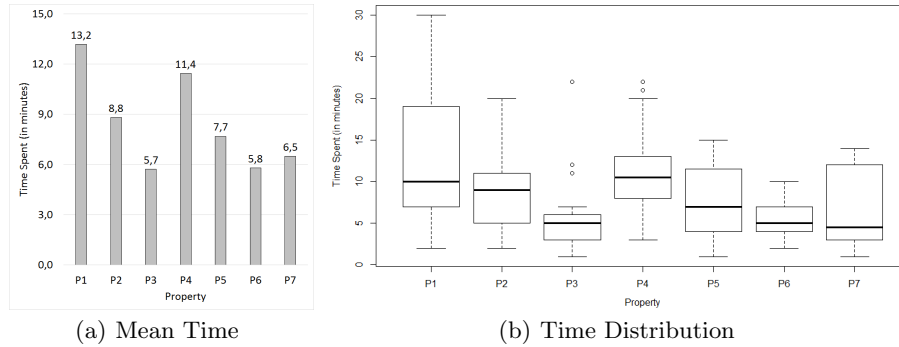
## 4 Results and Discussion

### 4.1 Time Spent to Write EA Instances (RQ<sub>1</sub>)

Figure 1 summarizes both mean and distribution of time spent by the participants to write EA instances for each AVL Tree property. As aforementioned

<sup>4</sup> <http://muclipse.sourceforge.net/>

(Section 3.2), the time spent by participant was collected from the Experiment Form. Based on Figure 1(a), we observe that participants spent from 5.7 to 13.2 minutes to write each EA instance. Properties  $P1$  – *Tree without elements* and  $P4$  – *Binary search tree* required more time to be addressed with assertions: 13.2 and 11.4 minutes, respectively. This result may be partially explained by the fact that P1 was the first property presented in the Experiment Form. Indeed, participants were just starting the quasi-experiment, so that they may have spent more time in understanding the procedures and tasks. Additionally, P4 is quite time consuming to address, especially because it requires inspecting each node of the AVL Tree. Thus, it was expected that participants would spend a considerable time in drawing a strategy to navigate through the tree for inspection. For the record, P5 to P7 have a similar logic; thus, developers tend to spend less time to address these properties after P4 – which our results have confirmed.



**Fig. 1.** Mean and Distribution of Time Spent by AVL Tree Property

Aimed at a detailed understanding, Figure 1(b) illustrates the time distribution for each AVL Tree property. We have grouped these results by the distribution pattern as follows. **Group 1 (P1, P2, and P4)**: P1 and P2, which are the two first properties in the Experiment Form, showed a highly spread distribution. For instance, P1 values ranged from two to 30 minutes. P4, which is a hard property to address, showed a similarly spread distribution: from three to 20 minutes besides outliers lower than 25 minutes. **Group 2 (P3, P5, P6, and P7)**: these properties showed the less spread distributions. This observation could be justified by the fact that P3 is among the easiest addressable properties, while P5, P6, and P7 have a logic quite similar to P4, which is hard rely on a similar logic of P4, which is considerably time consuming.

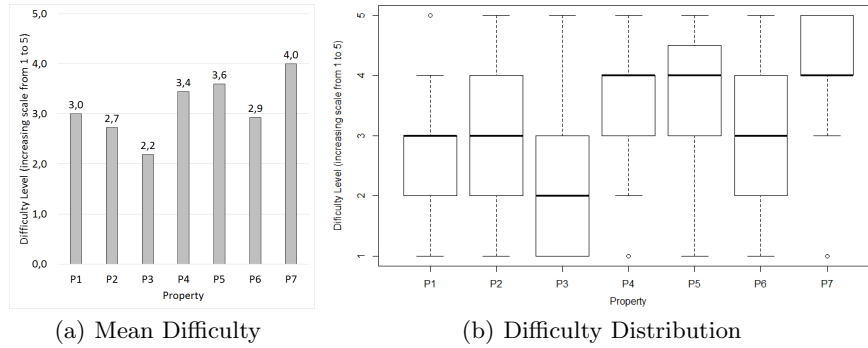
**Summary of RQ<sub>1</sub>:** Participants took from 5.7 to 13.2 minutes in average to write EA instances for each AVL Tree property. Time distribution was highly spread for properties with a justifiable complexity according to our study design (e.g. P1 and



P4). A quite balanced distribution was found for the other properties. In summary, participants spent a reasonable time while writing EA instances.

## 4.2 Difficulty Faced while Writing EA Instances (RQ<sub>2</sub>)

Figure 2 show both mean and distribution of difficulty to write EA instances for each AVL Tree property. Figure 2(a) shows an average difficulty ranging from 2.2 to four out of five. Difficulty increased along with the quasi-experiment execution, expect from P1 to P3. Indeed, *P1 – Tree without elements*, which is the first property in the Experiment Form, was when participants were introduced to EA. After that, P2 and P3 were easier to address with assertions, though these properties are more complex then P1 in our opinion. In summary, the learning factor may have facilitated the EA writing. Differently, difficulty increased from P4 on. It is worth mentioning that *P4 – Binary search tree* has a quite complex logic, but its successors (P5 to P7) share this logic. Although the participants were familiar with the AVL Tree properties (Section 3.3), they found it particularly difficult to address these more complex properties.



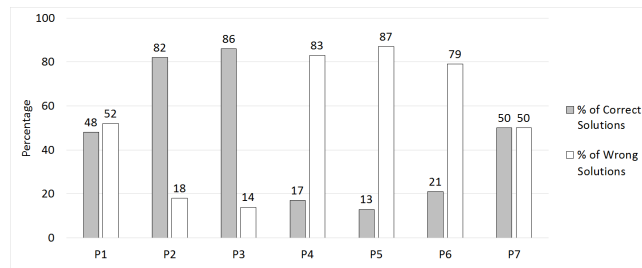
**Fig. 2.** Mean and Distribution of Difficulty for each AVL Tree Property

Figure 2(b) illustrates the difficulty distribution reported for each AVL Tree property. We have found two different groups of distributions described as follows. **Group 1 (P1, P2, P3, and P6)**: these properties presented the lowest distributions. This result can be partially justified by the fact that P1 to P3 have the less complicated logic. This is why we arranged these properties to be addressed first in the Experiment Form. In the case of P6, it is worth remembering that P6 reuses the logic of P4, which is quite complicated. **Group 2 (P4, P5, and P7)**: these properties showed the highest difficulty distributions. The second quantile for all three distributions is greater than three, which confirms a high effort to write EA instances for complex properties.

**Summary of RQ<sub>2</sub>:** Participants found it considerably difficult to write EA instances. The average difficulty ranged from 2.2 to four. The most difficult properties to address with EA instances are those with a more intricate logic.

### 4.3 Correctness of the Participant-Written EA Instances (RQ<sub>3</sub>)

Figure 3 presents the percentage of correctly written EA instances for each AVL Tree property. In this particular case, we observed three major groups of properties, which we describe as follows. **Group 1 (P2 and P3):** only two properties presented a percentage of correctly written EA instances greater than the percentage of wrongly written instances. This result is not exactly surprising because, as discussed in Section 4.2, both P2 and P3 had a quite simple logic. Besides that, these properties succeeded P1 in the order of the Experiment Form. **Group 2 (P1 and P7):** for these two properties, the percentages of correctly and wrongly written EA instances is practically the same. This result is partially justifiable by the fact that, as discussed in Section 4.1 and 4.2, P1 was the first property to be introduced to the participants. It is worth mentioning that only a few participants wrote EA instances for P7. Thus, the balanced rates of correctly and wrongly written EA instances is reasonable. **Group 3 (P4, P5, and P6):** for these three properties, we observed greater rates of wrongly written EA instances when compared to correctly written ones. This result is acceptable once these properties are among the most complex ones.















**Fig. 3.** Percentage of Correct EA Instances for each AVL Tree Property

**Summary of RQ<sub>3</sub>:** Participants rarely wrote EA instances that correctly implement the AVL Tree properties. Once our participants are familiar with AVL Trees, we conclude that non-experts are quite likely to write wrong EA instances.

#### 4.4 Effectiveness of the Participant-Written EA Instances (RQ<sub>4</sub>)

Table 4.4 presents the percentage of participants (second column) and mean of killed mutants (third column) for different quantities of written EA instances (first column). We observed that none out of the 22 participants address all seven AVL Tree properties with EA instances. This result indicates the difficulty to perform this task in practice. Nevertheless, most participants (60%) wrote EA instances for at least tree properties. This result is particularly interesting because, with just three EA instances, an average of 57% mutants were killed. The mean percentage of killed mutants reached (70%) its peak with exactly five written EA instances. It is worth mentioning that, as discussed in Section 2.1, we have originally derived a total of nine invariant properties for the AVL Tree. Therefore, in a general analysis, addressing one third of the AVL Tree properties was sufficient to capture a half of the 155 automatically generated bugs.

**Table 2.** Percentage of Killed Mutants per Number of EAs

Quantity of EA Instances	Percentage	
	Participants	Mean of Killed Mutants
1	23% 	46% 
2	18% 	52% 
3	41% 	57% 
4	9% 	65% 
5	5% 	70% 
6	5% 	66% 

**Summary of RQ<sub>4</sub>:** EA instances showed quite effective in detecting bugs. With only one third of the AVL Tree properties addressed with EA instances, participants were able to observe 57% of the bugs generated via mutation testing.

## 5 Threats to Validity

**Construct Validity:** We have refined our study steps and their respective artifacts through a pilot study with four participants (Section 3.2). Thus, we expected to mitigate biases regarding an insufficient data collection. Additionally, to enable the quasi-experiment execution in a feasible time, we cherry-picked the AVL Tree program for analysis and EA writing. As discussed in Section 2.1, students are familiar with AVL properties and implementation; thus, this program sounds adequate for an academic experiment. Moreover, we discarded those AVL Tree properties that we judged too hard for participants to address in an one-hour quasi-experiment. We carefully selected a laboratory environment so that participants could comfortably perform the quasi-experiment tasks. Finally, we selected a well-known tool, MuClipse, to run the mutation testing. This tool has been successfully used in both academia and industry [20].

**Internal Validity:** Although Internet access was available and allowed along with the quasi-experiment, we assisted the participants in performing their tasks. As discussed in Section 3.2, we addressed the participants’ doubts whenever possible without biasing their responses – especially during the form filling. Thus, we expected to assure that all participants understood the study procedures and tasks. We decided to perform the quasi-experiment during a Software Engineering-related class, so that all participants are motivated to give their best to perform their tasks successfully. We asked each participant to inform both start and finish time for each EA instance implementation. This design decision, inspired by our past work [7], may represent a threat to the study validity if participants forget to compute time along with the quasi-experiment. We minimized threats of this matter by instructing the participants with examples before the quasi-experiment execution.

**Conclusion Validity:** We carefully filtered the quasi-experiment data before performing the data analysis. By eliminating data of participants that could compromise our study results (Section 3.3), we removed possible anomalies in the analysis of time, difficulty, and so forth. Similar to our past work [7, 8], we applied techniques of descriptive analysis focused on the data distribution. We have performed a manual analysis of the EA instances written by the participants, aimed at computing correctness (Section 4.3). Although this manual analysis could represent a threat to the study validity, we have allocated sufficient time to perform this analysis carefully. Finally, with respect to the effectiveness (Section 4.4), we carefully filtered the mutant versions of the AVL Tree program via MuClipse. Especially, we discarded all equivalent mutants, i.e., those mutant versions that are equivalent to the original program [5].

**External Validity:** We have performed our quasi-experiment with 22 Brazilian students. As one could expect, our participant set may not represent all Brazilian students in terms of background and professional experience. Nevertheless, this participant set allowed to achieve some preliminary insights on the difficulty and effectiveness of writing EA instances. Additionally, we restricted our quasi-experiment execution to one hour, especially because the experiment was conducted during a class. This time constraint may affect our findings, since participants may be uncomfortable to write code under time restrictions. However, according to our pilot study (Section 3.2), we observed that one hour was sufficient for participants to write at least a few EA instances correctly. Besides that, we performed the quasi-experiment in a laboratory equipped with sufficient desktop computers, aimed at a successful participation. We invite researchers to replicate our work in other academic settings and validate our study findings.

## 6 Related Work

Studies [1, 11, 12, 16, 21] explored the runtime bug detection via approaches similar to Executable Assertion (EA). One study [1] targeted the incorporation of Design by Contract (DbC) principles – e.g. invariants and preconditions – into concurrent Java programs. Such incorporation was enabled by runtime assertion,

which verify the program conformance with those principles. Other study [16] regarding DbC introduced measures targeting i) the ability of capturing bugs at runtime and ii) the effort required to find the exact location of a program bug. Another work [11] employed EA for verifying the correctness of critical programs through their execution flow. This work, similarly to another one [21], suggested that EA instances may observed a plenty of bug types at runtime. Finally, a recent work [12] empirically validated the potential of EA instances in enhancing the maintainability of data structure invariants, such as the AVL Tree ones (Section 2.1). Nonetheless, previous studies lacked empirical evidence on how difficult and effective is writing EA instances from a developers' perspective.

## 7 Final Remarks

This paper discussed a quasi-experiment with 22 students on how difficult and effective is writing Executable Assertion (EA). Each participant wrote EA instances for seven invariant properties of the AVL Tree. We assessed the average time spent and difficulty of writing EA instances, plus the correctness and the effectiveness of these instances in observing 155 automatically generated bugs. We found out a considerable effectiveness of EA instances in capturing bugs (55% of the bugs observed with only a few EA instances). However, each EA instance can be costly to write, taking up to 13.2 minutes each. Our quasi-experiment provided only a few hints on the difficulty and effectiveness of adopting EA in practice. Further studies are required to: i) understand the EA adoption by developers in real-world development settings, and ii) compare the cost-benefit of adopting EA against traditional bug detection techniques, e.g. unit testing.

## References

1. Araujo, W., Briand, L., Labiche, Y.: On the effectiveness of contracts as test oracles in the detection and diagnosis of functional faults in concurrent object-oriented software. *IEEE Trans. Softw. Eng. (TSE)* pp. 971–992 (2014)
2. Ayari, K., Bouktif, S., Antoniol, G.: Automatic mutation test input data generation via ant colony. In: 9th GECCO. pp. 1074–1081 (2007)
3. Basili, V., Rombach, H.D.: The TAME Project: Towards improvement-oriented software environments. *IEEE Trans. Softw. Eng. (TSE)* **14**(6), 758–773 (1988)
4. Bronson, N., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: ACM SIGPLAN Notices. vol. 45, pp. 257–268 (2010)
5. DeMillo, R., Lipton, R., Sayward, F.: Hints on test data selection: Help for the practicing programmer. *IEEE Computer* **11**(4), 34–41 (1978)
6. Fernandes, E.: Stuck in the middle: Removing obstacles to new program features through batch refactoring. In: 41st ICSE, Doctoral Symposium. pp. 206–209 (2019)
7. Fernandes, E., Ferreira, F., Netto, J.A., Figueiredo, E.: Information systems development with pair programming: An academic quasi-experiment. In: 12th SBSI. pp. 486–493 (2016)
8. Fernandes, E., Ferreira, L.P., Figueiredo, E., Valente, M.T.: How clear is your code? An empirical study with programming challenges. In: 20th CIbSE, ESELaw Track. pp. 1–14 (2017)

9. Ferreira, F., Fernandes, E., Oliveira, J., Souza, M., Figueiredo, E.: Companion research website. <https://fischerjf.github.io/assertions/> (2019)
10. Ferreira, F., von Staa, A., Figueiredo, E.: Uma análise da eficácia de assertivas executáveis como indicadoras de falhas em software. In: 9th SAST. pp. 1–10 (2015), (In Portuguese)
11. Goloubeva, O., Rebaudengo, M., Reorda, M.S., Violante, M.: Soft-error detection using control flow assertions. In: 18th DFT. pp. 581–588 (2003)
12. Gyori, A., Garg, P., Pek, E., Madhusudan, P.: Efficient incrementalized runtime checking of linear measures on lists. In: 10th ICST. pp. 310–320 (2017)
13. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng. (TSE)* **37**(5), 649–678 (2010)
14. Just, R., Jalali, D., Inozemtseva, L., Ernst, M., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: 22nd FSE. pp. 654–665 (2014)
15. Lavallée, M., Robillard, P.: Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. In: 37th ICSE. pp. 677–687 (2015)
16. Le Traon, Y., Baudry, B., Jézéquel, J.M.: Design by contract to improve software vigilance. *IEEE Trans. Softw. Eng. (TSE)* **32**, 571–586 (2006)
17. Lee, S., Bai, X., Chen, Y.: Automatic mutation testing and simulation on owl-s specified web services. In: 41st ANSS. pp. 149–156 (2008)
18. Likert, R.: A technique for the measurement of attitudes. *Arch. Psychol.* **140**(1), 1–55 (1932)
19. Liu, M.H., Gao, Y.F., Shan, J.H., Liu, J.H., Zhang, L., Sun, J.S.: An approach to test data generation for killing multiple mutants. In: 22nd ICSM. pp. 113–122 (2006)
20. Ma, Y.S., Offutt, J., Kwon, Y.R.: MuJava: A mutation system for Java. In: 28th ICSE. pp. 827–830 (2006)
21. Mahbub, K., Spanoudakis, G.: Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. In: 3rd ICWS. pp. 257–265 (2005)
22. Papadakis, M., Shin, D., Yoo, S., Bae, D.H.: Are mutation scores correlated with real fault detection? A large scale empirical study on the relationship between mutants and real faults. In: 40th ICSE. pp. 537–548 (2018)
23. Poth, A., Sasabe, S., Mas, A., Mesquida, A.L.: Lean and agile software process improvement in traditional and agile environments. *J. Softw.: Evol. Process* pp. 1–11 (2019)
24. Venkatasubramanian, R., Hayes, J., Murray, B.: Low-cost on-line fault detection using control flow assertions. In: 9th IOLTS. pp. 137–143 (2003)
25. Weiss, M.: *Data Structures and Algorithm Analysis in Java*. Pearson, 3rd edn. (2011)
26. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A.: *Experimentation in Software Engineering*. Springer Science & Business Media, 1st edn. (2012)
27. Yao, X., Harman, M., Jia, Y.: A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: 36th ICSE. pp. 919–930 (2014)
28. Yi, Q., Yang, Z., Liu, J., Zhao, C., Wang, C.: A synergistic analysis method for explaining failed regression tests. In: 37th ICSE. pp. 257–267 (2015)