Classifying generated white-box tests: an exploratory study

Dávid Honfi¹ · Zoltán Micskei¹

Published online: 17 May 2019
© The Author(s) 2019

Abstract

White-box test generation analyzes the code of the system under test, selects relevant test inputs, and captures the observed behavior of the system as expected values in the tests. However, if there is a fault in the implementation, this fault could get encoded in the assertions (expectations) of the tests. The fault is only recognized if the developer, who is using test generation, is also aware of the real expected behavior. Otherwise, the fault remains silent both in the test and in the implementation. A common assumption is that developers using white-box test generation techniques need to inspect the generated tests and their assertions, and to validate whether the tests encode any fault or represent the real expected behavior. Our goal is to provide insights about how well developers perform in this classification task. We designed an exploratory study to investigate the performance of developers. We also conducted an internal replication to increase the validity of the results. The two studies were carried out in a laboratory setting with 106 graduate students altogether. The tests were generated in four open-source projects. The results were analyzed quantitatively (binary classification metrics and timing measurements) and qualitatively (by observing and coding the activities of participants from screen captures and detailed logs). The results showed that participants tend to incorrectly classify tests encoding both expected and faulty behavior (with median misclassification rate 20%). The time required to classify one test varied broadly with an average of 2 min. This classification task is an essential step in white-box test generation that notably affects the real fault detection capability of such tools. We recommended a conceptual framework to describe the classification task and suggested taking this problem into account when using or evaluating white-box test generators.

Keywords Software testing · White-box test generation · Empirical study · Test classification

✉ Dávid Honfi
honfi@mit.bme.hu

Zoltán Micskei
micskeiz@mit.bme.hu

¹ Department of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary

1 Introduction

Due to the ever-increasing importance of software, assessment of its quality is essential. In practice, *software testing* is one of the most frequently used techniques to assess and improve software quality. Thorough testing of software demands significant time and effort. To alleviate the tasks of developers, several *automated test generation techniques* have been proposed (Anand et al. 2013). These advanced techniques are often available as off-the-shelf tools, e.g., Pex/IntelliTest (Tillmann and de Halleux 2008), Randoop (Pacheco et al. 2007), or EvoSuite (Fraser and Arcuri 2013). These tools can rely *only on the source/binary code* to select relevant test inputs. For the selected inputs these white-box test generators record the implementation's actual output in assertions to be used as test oracles. However, if the implementation is used alone as input for test generation, these *assertions*—created in the generated code—contain the *observed behavior*, not the *expected one*.

As these techniques and tools evolve, more and more empirical evaluations are published to assess their capabilities. In most of the studies, the tools are evaluated in a technology-oriented setting, e.g., Kracht et al. (2014), Wang et al. (2015), and Shamshiri et al. (2015). Only a limited number of studies involved *human participants* performing prescribed tasks with the tools (Fraser et al. 2015; Rojas et al. 2015; Enoiu et al. 2016).

A common aspect of evaluating the effectiveness of test generator tools is the *fault detection capability* of the generated tests. Related studies (Enoiu et al. 2017; Fraser et al. 2015; Panichella et al. 2016a; Rojas et al. 2015; Ramler et al. 2012; Shamshiri et al. 2015; Nguyen et al. 2013) typically employ two metrics for this purpose: (1) mutation score and (2) number of detected faults. Mutation score shows how many mutations of the given source code can be detected by tests. Although mutation score has been shown to be in correlation with real fault detection capability (Just et al. 2014), it has concerns to be aware of Papadakis et al. (2016) (e.g., omitting specific mutants may introduce large bias to fault detection capability). The number of detected faults is commonly measured using a faulty version (with injected faults) and a fault-free version (reference) of the code under test. It is usual in these studies that if a generated test passes on the faulty version and fails on the original, it is considered as a *fault-detector*, because it successfully encoded the wrong behavior.

Motivation However, the fundamental problem is that the developers¹ who use test generators have *no complete and precise knowledge* about the correctness of the implementation in real scenarios (i.e., there is no faulty and fault-free versions of the program like in related studies, but only one version, for which the developer does not know whether it contains any fault or not). If the test generator uses only the program code, then the developer *must validate each assertion* manually in the generated test code to decide whether the test encodes the *expected behavior* with respect to a given specification, also known as oracle checking (Prado and Vincenzi 2018). This is very simple for trivial errors, but it could be rather complex in case of slight mismatches between the implementation and its intended behavior. Although some experiments (Fraser et al. 2015; Pastore and Mariani 2015) mention this issue, or some try to solve it with the crowd (Pastore et al. 2013), most of the studies involving white-box test generation do not consider it as a potential threat to validity in their

¹ In this paper, we will refer to all software and test engineers using white-box test generators as developers.

```

/// <summary>Calculates the sum of given number of
/// elements from an index in an array.</summary>
/// <param name="start">Zero-based index.</param>
/// <param name="number">(Possibly zero) number of
/// elements to summarize.</param>
/// <param name="array">Array to use.</param>
/// <exception cref="ArgumentException">Thrown
/// when the inputs are invalid.</exception>
int CalculateSum(int start, int number, int[] a) {
    if(start+number > a.Length || a.Length <= 1)
        throw new ArgumentException();
    int sum = 0;
    for(int i = start; i < start+number-1; i++)
        sum += a[i];
    return sum;
}

```

Listing 1 Source for the example presenting the motivation

evaluations. The *consequence* of this is that the practical fault-finding capability of generated white-box tests *can be much lower* than presented in experimental evaluations that focus only on the technical aspects of fault detection. Although this paper aims at generated tests only, the problem addressed is also valid for hand-crafted white-box tests as well.

Example Consider the example function presented in Listing 1, which showcases the addressed problem. The intended behavior is described in the commented textual description (“specification”): the method returns the sum of `number` elements starting from `start` index in the array `a`. If the inputs are invalid (e.g., too long or too short), then the method throws an `ArgumentException` exception.

To demonstrate the task of generated white-box test classification, we generated tests with Microsoft IntelliTest (Tillmann and de Halleux 2008) for the function in Listing 1. The generated test inputs and outcomes are found in Table 1. Also, we present the code of the generated tests in Listing 2. What are the traits of these tests and what information do we obtain about the possible faults in the implementation with the help of generated tests?

1. The first test (with ID T1) starts from index 0 and gets the sum of 0 elements. The return value of the function with these two inputs is 0 (therefore, the tool generates an assertion for 0). This is the expected behavior (based on the specification).
2. The second generated test also starts from zero, but sums up 4 elements. The return value for these inputs is 15, which is unexpected with respect to the specification: the expectation is $4 + 5 + 6 + 7 = 22$. However, the fault is only detected if the developer inspects the test code and realizes that this assertion encodes an unexpected behavior.

Objective The example showed that depending on whether a fault is triggered in the implementation by the selected inputs, the generated tests can encode an expected or unexpected behavior with respect to the specification. In this paper, we will classify the expected case as OK and the unexpected case as WRONG.² But unless there are other test oracles available,

²We will revisit this categorization and its implications in the discussion (Section 8.3).

Table 1 A possible set of generated tests for the example

ID	start	number	a	Result	Classification
T1	0	0	0	4, 5, 6, 7, 8	OK
T2	0	4	15	4, 5, 6, 7, 8	WRONG

this classification is not automatic, and the developer must validate the information in the generated tests. However, it is not evident that humans can correctly perform this classification and identify all faults that can be possibly detected using the generated tests. Therefore, the question that motivated our research is the following:

How do developers who use test generator tools perform in deciding whether the generated tests encode expected or unexpected behavior?

Answering this question would help in assessing the fault-finding capability of white-box test generator tools more precisely, and in identifying threats when using or evaluating such tools in practice.

Method We designed and performed an *exploratory study* with human participants that cover a realistic scenario resembling junior software developers testing previously untested code with the help of test generators. Moreover, we performed an *internal replication* of our study to gain more confidence in the results from the first study. The participants' task was to classify tests generated by Microsoft IntelliTest (Tillmann and de Halleux 2008) based on whether the generated tests encode expected or unexpected behavior (OK or WRONG). We carefully selected 4 open-source projects from GitHub, which suit the study's purpose. The activities of participants were recorded using logging and screen capture. Furthermore, the collected data were analyzed both quantitatively and qualitatively.

Results The results show that deciding whether a test encodes expected behavior was a challenging task for the participants even in a laboratory setting with artificially prepared environments. Only 4 of the 106 participants were able to classify all 15 tests correctly. Surprisingly, a large number of tests encoding expected behavior were also misclassified (as WRONG). The time required to classify one test varied broadly with an average of 2 min. In

Listing 2 Source code for tests with ID T1 and T2. The assertion encoding unexpected behavior is highlighted

```
[TestMethod]
public void CalculateSumTest284() {
    int[] ints = new int[5] { 4,5,6,7,8 };
    int i = this.CalculateSumTest(0, 0, ints);
    Assert.AreEqual<int>(0, i);
}

[TestMethod]
public void CalculateSumTest647()
{
    int[] ints = new int[5] { 4,5,6,7,8 };
    int i = this.CalculateSumTest(0, 4, ints);
    Assert.AreEqual<int>(15, i);
}
```

experimental research, the possibility to replicate the study is vital to increase the validity of results. Thus, we made the whole dataset, along with the videos, and the full analysis scripts available for further use (Honfi and Micskei 2018).

Our results have implications both for research and practice. First, the outcomes emphasized that creators of test generator tools should take the test classification problem into account, as it can clearly affect the real fault-finding capability of tools. Furthermore, our recommendation is to consider the revealed threats when evaluating the fault-finding capabilities of white-box test generators in empirical studies. Finally, developers using test generators should pay attention to that generated tests need validation and identifying a fault is not trivial.

Contributions The main contributions of the paper are as follows.

- We *designed* an exploratory study with human participants to investigate the importance of test classification in white-box test generation (Section 3).
- We *performed* two studies with a total of 106 participants (Section 4) and *analyzed* the results (Section 5 and Section 6). The collected data shows that classification of generated tests is not easy, and humans can not necessarily detect all faults.
- When discussing (Section 8) the implications of the results we gave recommendations to improve test generators. Moreover, we identified open questions and *proposed* a preliminary conceptual framework for classifying generated white-box tests to help direct future research.

2 Related work

Test generation and oracles Anand et al. (2013) present a survey about test generation methods, including those that generate tests only from binary or source code. As these methods do not have access to a specification or model, they rely on other techniques than specified test oracles (Barr et al. 2015). For example, for certain outputs, *implicit oracles* can be used: a segmentation fault is always a sign of a robustness fault (Shahrokni and Feldt 2013), while finding a buffer overflow means a security fault (Bounimova et al. 2013). Other implicit oracles include general contracts like $\circ.\text{equals}(\circ)$ is true (Pacheco et al. 2007). However, test generators usually generate numerous tests passing these implicit oracles. For handling these tests there are basically two options. On the one hand the developer could specify *domain-specific partial specifications*, e.g., as parameterized tests (Tillmann and de Halleux 2008) or property-based tests (Claessen and Hughes 2000). On the other hand, the tools usually record the *observed output* of the program for a given test input in assertions, and the developer could *manually examine* these asserts to check whether the observed behavior conforms to the expected behavior.

In our paper, we consider this latter case, i.e., where there is no automatically processable full or partial specification, the generated tests were already filtered by the implicit oracles, but we cannot be sure if they encode the correct behavior. In this case *derived oracles* are commonly used to *decrease* the number of tests to manually examine or ease the validation. For example, existing tests can be used to generate more meaningful tests (McMinn et al. 2010), similarity between executions can be used to pinpoint suspicious asserts (Pastore and Mariani 2015), or clustering techniques can be used to group potentially faulty tests

(Almaghairbe and Roper 2016). Moreover, if there are *multiple versions* from the implementation, then tests generated from one version could be executed on the other one, e.g., in case of regression testing (Yoo and Harman 2012) or different implementations for the same specification (Pacheco et al. 2007). However, even in this scenario, tests do not detect faults, but merely *differences* that need to be manually inspected (e.g., a previous test can fail on the new version not only because of a fault, but because a new feature has been introduced). In summary, none of these techniques can classify all tests perfectly, and the remaining ones still need to be examined by a human.

Testing studies involving humans Juristo et al. (2004) collected testing experiments in 2004, but only a small number of the reported studies involved human subjects (Myers 1978; Basili and Selby 1987). More recently, experiments evaluating test generator tools were performed: Fraser et al. (2015) designed an experiment for testing an existing unit either manually or with the help of EvoSuite; Rojas et al. (2015) investigated using test generators during development; Ramler et al. (2012) compared tests written by the participants with tests generated by the researchers using Randoop; and Enoiu et al. analyzed (2016) tests created manually or generated with a tool for PLCs. These experiments used mutation score or correct and faulty versions to compute fault detection capability. Therefore, they left out the human examination step in the end, which is the focus of the current study.

Related studies We found three studies that are closely related to our objectives. In the study of Staats et al. (2012), participants had to classify invariants generated by Daikon. They found that users struggle to determine the correctness of generated program invariants (that can serve as test oracles). The object of the study was one Java class, and tasks were performed on printouts. Pastore et al. (2013) used a crowdsourcing platform to recruit participants to validate JUnit tests based on the code documentation. They found that the crowd can identify faults in the test assertions, but misclassified several harder cases. Shamshiri et al. (2018) conducted an experiment along with two replications with 75 participants to learn more about how generated tests influence software maintenance. Their setting started with a failing test (caused by an artificial change in the code) and participants were asked to (1) decide whether it is a regression fault or the test itself contains errors and (2) fix the cause of the problem. They found that the regressive maintenance of generated tests can take more time with same effectiveness. However, they do not consider the case when the generated tests is created on originally faulty code (mismatching the specification) on which our study focuses (thus, ours is not a regression scenario). These studies suggest that classification (and regressive maintenance) is not trivial. Our study extends these results by investigating the problem in a setting where participants work in a development environment on new features of a more complex project.

3 Study planning

3.1 Goal and method

Our main goal was to study whether developers can validate the tests generated only from program code by classifying whether a given test encodes an expected or an unexpected behavior.

As there is little empirical evidence about the topic, to understand it better, we followed an *exploratory* and *interpretivist* approach (Wohlin and Aurum 2015). We formulated the following *base-rate research questions* (Easterbrook et al. 2008) to gather data that can direct future research or help to formulate theories and hypotheses.

RQ1 How do developers perform in the classification of generated tests?

RQ2 How much time do developers spend with the classification of generated tests?

As these test generator tools are not yet widespread in industry we selected an *off-line context*. We designed an exploratory study in a laboratory setting using students as human participants. Also, we performed a replication to strengthen our initially explored results. Our research process involved both quantitative and qualitative phases. We collected data using both observational and experimental methods. The data obtained was analyzed using exploratory data analysis and statistical methods. For the design and reporting of our study, we followed the guidelines of empirical software engineering (Easterbrook et al. 2008; Wohlin and Aurum 2015; Wohlin et al. 2012).

3.2 Variable selection

Understanding and classifying generated tests is a complex task and its difficulty can be affected by numerous factors. We focus on the following *independent variables*. For each variable, their possible levels are listed, from which the bolds are the ones we selected for our study design.

- *Participant source*: What type of participants are recruited (**students**, professionals, mixed).
- *Participant experience*: Experience in testing and test generation tools (none, **basic**, experienced).
- *Participant knowledge of objects*: Whether the participant has a priori knowledge about the implementation under test (known, **unknown**).
- *Objects source*: The source, where the objects are selected from (**open source**, closed source, artificial/toy, ...).
- *Object source code access*: Whether the objects are fully visible to the participants (**white-box**, black-box).
- *Fault types*: The source and type of the faults used in the objects (real, **artificial**, mutation-based).
- *Number of faults*: The number of faults injected into the objects (0, 1, 2, **3**, ...).
- *Expected behavior description*: How the specifications of the objects are given (**code comments**, text document, formal, ...).
- *Test generator tool*: Which test generator is used for generating tests (**IntelliTest**, EvoSuite, Randoop, ...).
- *User activity*: The allowed user activities in the study (**run**, **debug**, modify code ...).

The following *dependent variables* are observed:

- *Answers of participants*: Classification of each test as OK (expected) or WRONG (unexpected).

- *Activities of participants*: What activities are performed by participants during the task (e.g., running and debugging tests).
- *Time spent by participants*: How much time participants spend with each individual activity and in each location.

Note that as this is an exploratory research there is no hypothesis yet, and because the research questions are not causality-related or comparative questions, all independent variables had fixed levels (i.e., there are no factors and treatment as opposed to a hypothesis-testing empirical evaluation).

3.3 Test generator tool

There are many off-the-shelf tools for white-box test generation, from which we chose Microsoft IntelliTest (Tillmann and de Halleux 2008). We decided to use it because it is already a state-of-the-art, mature product with a good user interface. IntelliTest currently supports C# language, and it is fully integrated into Visual Studio 2015. IntelliTest's basic concept is the *parameterized unit test* (PUT), which is a test method with arbitrary parameters called from the generated tests with concrete arguments. Also, the PUT serves as an entry point for the test generation process.

3.4 Subjects (participants)

Our goal was to recruit people, who were already familiar with the concepts of unit testing and white-box test generation. We performed the recruitment among MSc students who enrolled in one of our V&V university courses. They were suitable candidates as they already had a BSc degree in software engineering. Furthermore, our course has covered testing concepts, test design, unit testing and test generation prior to the performed study (5×2 h of lectures, 3 h of laboratory exercises, and approximately 20 h of group project work on the topics mentioned). Throughout the course, we used IntelliTest to demonstrate white-box test generation in both the lectures and the laboratory exercises.

Participation in the study was optional. We motivated the participation by giving the students extra points (approximately 5% in the final evaluation of the course). Note that we also announced that these points are given independently from the experiment results to avoid any negative performance pressure.

Using students as participants in a study instead of professionals has always been an active topic in empirical software engineering. However, Falessi et al. (2018) have conducted a survey with empirical software engineering experts, whether they agree or disagree about using students. On the one hand, based on their results, using students is a *valid simplification* of real-world settings for a laboratory study. On the other hand, this remains a threat to the validity as well, which must be considered during the interpretation of results.

3.5 Objects (projects and classes)

In terms of objects, we had to decide whether to (i) select off-the-shelf projects, or (ii) give developers an implementation task (based on a predefined specification). The latter study setup can be more difficult to control and analyze, because generated tests could differ between implementations. Thus, we decided to select objects from an external source, and to prohibit participants implementing or modifying any code.

The main requirements towards the objects were that (i) they should be written in C#, (ii) IntelliTest should be able to explore them, and (iii) they should not be too complex so that participants could understand them during performing their task. We did not find projects satisfying these requirements in previous studies of IntelliTest (Pex); thus, we searched for open-source projects. Based on our requirements, the project selection was performed along the following criteria:

- Shall have at least 400 stars on GitHub: this likely indicates a project that really works and may exclude prototypes and not working code.
- Should not have any relation to graphics, user interface, multi-threading or multi-platform execution: all of these may introduce difficulties for the test generator algorithm by its design.
- Shall be written in C# language: The IntelliTest version used only supports this language.
- Shall be able to compile in a few seconds: this makes users able to run fast debugging sessions during the experiment.

We decided to use *two different classes from two projects* with vastly different characteristics for both the original and the replicated study. The selection criteria for the classes were the following:

- Shall be explorable by IntelliTest without issues to have usable generated tests.
- Shall have more than 4 public methods to have a reasonable amount of generated tests.
- Shall have at least partially commented documentation to be used as specification.

We conducted pilots prior to finalizing our design. We found that participants can examine *15 tests* in a reasonable amount of time. To eliminate the bias possibly caused by tests for the same methods, we decided to have the *15 tests for 5 different methods* (thus 3 tests for each method).

3.5.1 Selected projects and classes

Finding suitable objects turned out to be much harder than we anticipated. We selected 30 popular projects (Honfi and Micskei 2018) from GitHub as candidates that seemed to satisfy our initial requirements. However, we had to drop most of them: either they heavily used features not supported by IntelliTest (e.g., multi-threading or graphics) or they would have required extensive configuration (e.g., manual factories, complex assumptions) to generate non-trivial tests. Finally, we kept the two most suitable projects that are the following:

- *Math.NET Numerics* (MathNET 2017) is a .NET library that offers numerical calculations in probability theory or linear algebra. It contains mostly data structures and algorithms.
- *NBitcoin* (NBitcoin 2017) is a more business-like library, which is available as the most complete Bitcoin library for .NET.

In terms of the replicated study, we performed the same selection procedure on another set of open-source projects from GitHub that suit the initial requirements. We finally decided on the following two.

- *NodaTime* (NodaTime 2018) is an advanced date and time handling library that aims to replace the corresponding built-in .NET types with a richer feature set.

- *NetTopologySuite* (NetTopologySuite 2018) is a .NET library, which implements 2-dimensional linear geometry based on a standard defined by the Open Geospatial Consortium.

Table 2 lists the selected classes of the four projects. Using the requirements for the classes we manually analyzed each method inside them to ensure that they are suitable for the purpose. The *Combinatorics* class implements enumerative combinatorics and counting: combinations, variations and permutations, all with and without repetitions. The *AssetMoney* class implements the logic of the Open Asset protocol for arbitrary currencies that have conversion ratio to Bitcoin. Class *Period* in project *NodaTime* is responsible for describing and handling a given date and time period. The class *CoordinateArrays* of project *NetTopologySuite* is responsible for handling coordinates organized into an array along with providing the corresponding operations as well.

Most of the selected methods originally had method-level comments containing the description of expected behavior. In case of missing descriptions, we extended them; they are still not perfect (nor formally complete), but based on feedbacks from preliminary pilot sessions (discussed later in Section 4), they tend to represent comments used in real projects. It is important to note here that we did not extend anything if the methods invoked (from the unit under test) had clear descriptions. This way participants had to explore and understand the code more deeply to provide classification answers.

3.5.2 Fault selection and injection

To obtain fault-encoding tests from IntelliTest, faults need to be injected into the classes under test. There are multiple alternatives to obtain such faults, each of them affect the validity of the study in different ways.

- *Historical faults* extracted from issue archives would more likely represent real-world scenarios, but would make the control of the study more difficult due to the limited number and type of actual faults for the selected projects.
- *Artificial faults* can be obtained from surveys of typical software defects, e.g., Duraes and Madeira (2006). These surveys rank and categorize the most common faults made by developers during software development. On the one hand, this enables more control over the faults and objects in the study (because of the vast amount of possibilities); on the other hand, it may reduce the similarity to a real scenario if the selection and injection is performed without care (e.g., the fault is too trivial or impossible to find).

Table 2 Details of the selected objects for the original (top) and replicated study (bottom)

Project			Selected class		
Name	No. of classes	KLOC	Name	No. of methods	LOC
NBitcoin	602	29.8	AssetMoney	44	202
Math.NET	271	48.7	Combinatorics	17	95
NodaTime	303	6.7	Period	51	231
NetTopologySuite	611	21.9	CoordinateArrays	22	147

As we did not find a diverse, controllable set of historical faults for the selected classes from the GitHub version history of the projects, we used artificial faults in a systematic way. We selected representative fault types (Duraes and Madeira 2006) from the Orthogonal Defect Classification (Chillarege et al. 1992). The survey we used identifies the most commonly committed types of faults in real-world programs. We selected the actual faults from the top quarters of the ODC categories (see Table 9). During the injection procedure, we made sure that the faults (1) are causing unexpected behavior, (2) have no cross-effects on each other, and (3) have no effect on behavior other than the intended. All three were validated using test generation (IntelliTest) on the classes with and without the injected faults. We injected *three faults in each selected class* in order to have faulty tests in minority, yet in a measurable number.

3.5.3 Generated tests

We generated tests with IntelliTest for each selected method using parameterized unit tests (Tillmann and Schulte 2005). Tests were generated from the version already containing the selected faults. There were methods, where IntelliTest could not generate values that cover interesting or unexpected behaviors. In these cases, we extended the parameterized unit tests with special assumptions that request at least one test from IntelliTest with values that fulfill the preconditions. From each test suite, we selected 3 tests for the study due to the following reasons:

- *Fitting in the time frame:* During the pilot sessions, we measured that a single test classification takes around 2–3 min to finish on average. Using 3 tests per method ensured that participants will most likely fit into the 1-h time frame of the study.
- *Learning effect:* The more tests are selected for a single method, the higher is the probability that participants will understand and learn the features and issues of that method. This could introduce a bias in the final results, which may falsely indicate the simplicity of the classification task.

We chose the most distinct cases that cover vastly different behaviors in the method under test (e.g., from different equivalence partitions). Each test was given an identifier ranging from 0 to 14 (therefore, all four projects have tests T0 to T14). Furthermore, the corresponding method is indicated with a suffix in each test identifier. Thus, for the first method, three cases were generated: T0.1, T1.1, and T2.1. IntelliTest generates one test file for each method, but we moved the tests into individual files to alleviate the tracking of participant activities.

3.6 Environment

A Windows 7 virtual machine was used that contained the artifacts along with Visual Studio 2015 and Google Chrome. Participants were asked to use only two windows: (1) an experiment portal in Chrome (for brief test overview and answer submission) and (2) Visual Studio (for code inspection, test run, and debug).

We designed a special website, the experiment portal (Fig. 1), in order to record the answers of the participants. Here, participants could give their answers while analyzing the generated test code and the expected behavior. This was a more reliable way to collect the results than using some mechanism in the IDE (e.g., using special comments), as participants could not unintentionally delete or regenerate the test code.

Method under test: public static AssetMoney Min(AssetMoney a, AssetMoney b)

```
/// Returns the smaller of two AssetMoney instances.
/// AssetMoney a: The first AssetMoney
/// AssetMoney b: The second AssetMoney
/// returns: The smaller AssetMoney.
public static AssetMoney Min(AssetMoney a, AssetMoney b)
```

Test method: MinTest13 (ID: 9)

```
[TestMethod]
[PexGeneratedBy(typeof(AssetMoneyTest))]
public void MinTest13()
{
    AssetId assetId;
    AssetMoney assetMoney;
    AssetMoney assetMoney1;
    byte[] bs = new byte[0];
    assetId = new AssetId("");
    assetId._Bytes = bs;
    assetMoney = new AssetMoney(assetId, 0uL);
    assetMoney1 = this.MinTest(assetMoney, assetMoney);
    Assert.IsNotNull((object)assetMoney1);
    Assert.AreEqual<long>(0L, assetMoney1.Quantity);
    Assert.IsNotNull(assetMoney1.Id);
    Assert.IsNotNull((object)(assetMoney1.Id._Bytes));
    Assert.AreEqual<int>(0, assetMoney1.Id._Bytes.Length);
}
```

☒ Mark as OK

☐ Mark as WRONG

Fig. 1 A test page in the experiment portal, where participants could give their answers, while examining the generated test code

Participants used this portal to decide whether the test encode expected behavior or not. The portal displayed the test code and the commented specification of the corresponding method. They recorded their answer using two buttons. Moreover, participants could correct their already answered cases. Questions could be skipped if a participant was not sure in the answer (however, nobody used that option).

In Visual Studio, the default development environment was provided with a simple activity tracking extension. Participants got the *full project* with every class. Participants were asked (1) not to modify any code, (2) not to execute IntelliTest, and (3) not to use screen splitting. On the other hand, we encouraged them to *use test execution* and *debugging* to explore the behavior implemented in the code under test.

3.7 Procedure

The main procedure of the 2-h sessions is as follows:

1. Sign informed consent.
2. Find a seat, receive a unique, anonymous identifier.
3. Fill background questionnaire.
4. Listen to a 10-min briefing presentation and go through a 15-min guided tutorial.
5. Perform the assigned classification task in at most 1 h.
6. Fill exit survey.

Participants only received one sheet of paper that describe both the procedure and the task with the path to the project and class under test. To obtain detailed knowledge about the participants, we designed a background questionnaire asking about their experience with development and testing. Also, the questionnaire had a quiz in the end about C# and testing. We designed a 10-min presentation in which the procedure, the project and class under test, the environment, the basic concepts of IntelliTest, and the rules are introduced. Also, participants were implicitly warned to check the invoked methods of the method under test to obtain full overview of both the required and the actual behavior.

To make participants familiar with the environment and the task, a 15-min guided tutorial was held on a simple project created for this specific purpose. The tutorial had both types of tests to be classified (OK and WRONG). The main task was to classify each of the 15 generated tests in the portal whether they encode expected (OK) or unexpected (WRONG) behavior. Finally, an exit survey was filled that asked participants about their feelings regarding the task accomplished.

We planned to perform 2-2 sessions for both the original and the replicated study, as the room where the study was planned to be conducted had only 40 seats available.

3.8 Data collection

We used two data collection procedures. On the one hand, we extended the development environment so that it logged every window change, test execution, and debug. Also, we wrote a script that documented every request made to the experiment portal. On the other hand, we set up a screen recording tool to make sure that every participant action is recorded.

Each participant had 6 output files that were saved for data analysis.

- *Answers*: The answers submitted to the portal in JSON format.
- *Background*: The answers given in the background questionnaire in CSV format.
- *Exit*: The answers given in the exit survey in CSV format.
- *Portal log*: The user activity recorded in the portal.
- *Visual Studio log*: The user activity recorded into a CSV-like format using a custom Visual Studio extension.
- *Screen recorded video*: The participant activity during the main session in MP4 format.

3.9 Data analysis

First, the raw data was processed by checking the answers of the participants along with their activities via parsing the logs or coding the screen capture videos. Next, the processed data was analyzed using exploratory techniques.

Table 3 Confusion matrix for participant answers

Correct answer	Participant answer	
	OK	WRONG
OK	True negative (TN)	False positive (FP)
WRONG	False negative (FN)	True positive (TP)

Analysis of answers We analyzed the answers obtained from the experiment portal using binary classification for which the confusion matrix is found in Table 3.

Video coding In the original study, we annotated every recorded video using an academic behavioral observation and annotation tool called Boris (Friard and Gamba 2016). We designed a behavioral coding scheme that encodes every activity, which we were interested in. The coding scheme can be found in Table 4; all occurrences of these events are marked in the videos (Fig. 2). Note that, during the video coding, we only used point events with additional modifiers (e.g., change of page in the portal is a point event along with a modifier indicating the identifier of the new page). In order to enable interval events, we created modifiers with start and end types.

Exploratory analysis We performed the exploratory data analysis (EDA) using R version 3.3.2 (R. Core Team 2016) and its R Markdown language to document every step and result of this phase. We employ the most common tools of EDA: box plots, bar charts, heat maps, and summarizing tables with aggregated data.

3.10 Threats to validity

During the planning of our study, we identified the threats to its internal, external, and construct validity. In terms of *internal threats*, our results might be affected by the common threats of human studies (Ko et al. 2013). For instance, this includes the maturation effect caused by the learning of exercises, and the natural variation in human performance as well. We addressed this threat by randomly ordering the methods and generated tests to classify. Also, to reduce fatigue and boredom, participants only had to deal with 3 tests per method.

Table 4 Coding scheme of the video analysis

Behavior	Modifiers
Portal activated	—
Visual Studio activated	—
Changed page in portal	T0-14, Home
Change window in VS	CUT, SUT, PUT, T0-14
Marked as OK	—
Marked as WRONG	—
Remove answer	T0-14
Running test	—
Debugging test	Start, End
Submit	—

CUT class under test, *SUT* other system under test, *PUT* parameterized unit test

Fig. 2 Coded events in Boris for one participant (excerpt)

Moreover, the students might know each other and thus they could talk about the tasks of the study between the sessions (see Section 4). We eliminated this threat by using different projects and faults at each occasion. The data collection and analysis procedure might also affect the results; however, we validated the logs by R scripts and the portal functions by testing.

The *generalization* of our results (external validity) might be hindered by several factors, including the following:

- *Professionals or students*: The performances of students and professional users of white-box test generators may differ. Yet, involving students is common in software engineering experiments (Sjøberg et al. 2005), and results suggest that professional experience does not necessarily increase performance (Dieste et al. 2017). Our graduate students typically have at least 6 months of work experience; thus, they are on the level of an average junior developer.
- *A priori knowledge of objects*: The generalization of the results could be affected also by a less likely fact that some participant may had a priori knowledge about the selected projects as they are open-source; thus, they could classify tests better.
- *Completeness of described behavior*: Another threat to external validity is the expected behavior given in comments, and not in a precise program specification. However, our goal during the study design was to carefully select open-source projects, which do not have formal specifications of behavior in general. This decision on the one hand may reduce the genericity of results for projects with formal specifications (incomplete specifications may reduce classification performance), but on the other hand, it increases the genericity for open-source software.
- *Number and size of objects*: The number and size of classes and methods under test may affect the classification performance, and thus the generalization of our results. However, the objects used in the study were carefully selected using rigorous criteria, which were exactly prescribed by our study design (with considering the required times and learning curves). These requirements ensured that participants could finish their task in the given time frame.
- *Fault injection*: Fault injection procedure could have effects on the genericity of the results; however, we selected this approach after considering several other alternatives along with their trade-offs as discussed in Section 3.
- *Test generator tool*: We used IntelliTest during our experiments, which reduces the generalization of our results. However, IntelliTest is one of the major white-box test generators. The effects of another tool can be investigated in replication studies.

- *User activity*: Our study allowed participants to run and debug generated tests, but modification of the source code was prohibited. Although this fact reduces the similarity to a real development scenario, modifications in the code would hinder the comparability of participant results. Moreover, it would require a vastly different study design that would introduce several new threats to the validity.

The threats to the *construct validity* in our study is concerned with the independent variables. It might be the case that some of the variables we selected do not have effects on the difficulty of classification of generated white-box tests. We addressed this threat by carefully analyzing related studies and experiments in terms of design and results in order to obtain the most representative set of variables.

4 Execution

4.1 Pilots

Our original study was preceded by two separate pilot sessions. First, we performed the tasks using ourselves as participants. After fixing the discovered issues of the design, we chose 4 PhD students—having similar knowledge and experience as our intended participants—to conduct a pilot. We refined the study design based on the feedback collected (see object selection and project selection in Section 3). For the replicated study, we also had two 2 PhD students performing a pilot session for both newly selected projects.

4.2 Sessions

We separated our original live study into two different sessions. On the first occasion the NBitcoin project, on the second one Math.NET was used. The sessions were carried out on 1st and 8th December 2016.

Our replication study was also separated into two sessions: on 30th November 2017 participants dealt with project NodaTime, while on 7th December 2017, they had NetTopologySuite as their system under test. All four sessions followed the same, previously designed and piloted procedure and could fit in the preplanned 2-h slot.

4.3 Participants

In the original study, altogether 54 students volunteered of the 120 attending the course: 30 came to the first occasion (NBitcoin) and 24 to the second (Math.NET). Thirty-four of the students had 4 years or more programming experience, while 31 participants had at least 6 months industrial work experience. They scored 4.4 out of 5 points on average on the testing quiz of the background questionnaire.

The replication study involved 52 students (no intersection with the participants of the original study), from which 22 dealt with the project NodaTime, while the other 30 examined NetTopologySuite in the second session. Out of the 52 students, 43 had 4 or more years programming experience. In terms of work experiences, 36 participants have worked at least 6 months in the programming industry. The students in the replicated study scored 4.5 out of 5 on average in the testing quiz of the background questionnaire.

4.4 Data collection and validation

We noticed three issues during the live sessions in the original study. In the first session, Visual Studio cached the last opened window (in the virtual machine used); thus, participants got three windows opened on different tabs when they started Visual Studio. In the second session, we omitted the addition of a file to the test project of Math.NET that led to 3 missing generated tests in Visual Studio (for method `CombinationsWithRepetition`). We overcame this issue by guiding the participants step-by-step on how to add that test file. This guided part lasted approximately 9 min; thus, we extended the deadline to 69 min in that session. Finally, unexpected shutdown of two computers caused missing timing data for the first two tests for two participants (ID: 55 and 59). The rest of their experiments were recorded successfully. The experiment portal has a continuous saving mechanism; therefore, their classification answers were stored permanently. We took all these issues into account in the timing analysis. During the validation of the recorded data, we discovered only one issue. The network in the lab room went off on 1st December 2016, and due to this the experiment portal was not able to detect every activity. This data was recovered with the coded videos for each participant.

In the replicated study, there were no issues; thus, we used the raw activity logs as our main data source.

5 Results of the original study

5.1 RQ1: Performance in classification

To evaluate the overall performance of participants in the classification task, we employed binary classification using the confusion matrix presented in Table 3. Figures 3 and 4 present

Fig. 3 NBitcoin results of the participants measured with the common binary classification measures

Fig. 4 Math.NET results of the participants measured with the common binary classification measures

the overall results with all the answers given. The figure encodes all four outcomes of evaluated answers. The first and foremost fact visible in the results is that there are numerous erroneous answers (marked with two shades of red). This implies that not only **WRONG** cases were classified as **OK**, but also there were **OK** cases classified as **WRONG**.

In case of NBitcoin, there was only one participant (ID: 10) who answered without any errors. Also, there was no test, which was not marked falsely by at least one of the participants. Furthermore, one can notice two patterns in the results for NBitcoin. First, tests T0.1 and T2.1 show very similar results for the same participants. This can be caused by similarity of their source codes and names. However, there were no injected faults in the code, both cases encode expected behaviors with respect to the specification. The other noticeable result is that T11.4 has more wrong answers than correct ones. This test causes an exception to occur, yet it is an expected one. Although throwing an exception is not explicitly stated in the method comment, the specification of the invoked and exception-causing method implies its correctness.

In case of Math.NET, the overall results show similar characteristics to NBitcoin: there is no test, which was correctly classified by everyone, and also only one participant (ID: 47) was able to classify every test correctly. In this project, two tests show larger deviations in terms of results: T2.1 and T8.3.

- **T2.1:** Taking a closer look at T2.1 (encoding unexpected behavior) reveals that its functionality was simple: participants had to examine the binomial coefficient $\binom{n}{k}$ calculation. The fault was injected into the sanity check found at the beginning of the method (this sanity check is not explicit in the description, however, the definition of the binomial coefficient implies the restrictions). In this particular test, the test inputs should have triggered the sanity check to fail, however—due to the fault injected—the execution went through the check and the method yielded an unexpected behavior.
- **T8.3:** For test T8.3, the misunderstanding could have come from an implementation detail called factorial cache, which pre-calculates every factorial value from 1 to 170. The original documentation states that numbers larger than 170 will “overflow,” but does not detail its exact procedure and outcome (no explicit statement of exception is

given). The fault injected causes the input check to treat some invalid inputs as valid. Test T8.3 uses 171 as one of its inputs (incorrectly treated as valid) for which the faulty implementation returns positive infinity (which is consistent with other parts of the program). Some participants probably expected an overflow exception here, even though the specification did not state it. Thus, as the assertion generated (positive infinity for the result) is an expected behavior, the test can be marked as OK.

We also analyzed the data in terms of different metrics for binary classification. We consider the following widely used metrics suitable for measuring performance in our context: true positive rate (TPR), true negative rate (TNR), and Matthews correlation coefficient (MCC) (Powers 2011). Summary of these metrics is shown in Fig. 5.

In terms of TPR, participants of the NBitcoin session outperformed the results of participants working with Math.NET. For NBitcoin, the median is 1, which means that more than half of the participants were able to classify all tests encoding unexpected behavior as WRONG. In contrast, results for Math.NET show that the upper quartile starts from 0.75, which is much lower.

For TNR, the two projects show very similar results with almost the same medians and inter-quartile ranges. Only a slightly wider distribution is visible for NBitcoin. This and the results for TPR suggest that the classification could be easier for NBitcoin.

MCC is a correlation metric between the given and the correct answers, and thus gives a value between -1 and 1 . If MCC is zero, then the participant's classification has no relationship with the correct classification. For NBitcoin, the MCC values show slightly worse

Fig. 5 Box plots of the results containing detailed metrics of binary classification

results than what can be expected from TPR and TNR values. The median is only around 0.55, which is only a moderate correlation. In the case of Math.NET, the inter-quartile range of MCC is between 0.5 and 0.2, which can be considered as a low correlation between the correct classification and the ones given by participants. Note that both experiment sessions had participants with negative correlation that indicates mostly faulty answers.

Summary The overall results of the participants showed a moderate classification ability. Many of them committed errors when gave their answers. Some of these errors were possibly caused by the misunderstanding of the specification; however, a large portion of wrong answers may have been caused by the difficulty of the whole structured problem (involving understanding the specification, inspecting the code and its execution, and classifying the tests).

5.2 RQ2: Time spent for classification

We analyzed the data obtained from the video annotations from various aspects to have an overview of the time management of participants. Note that during the time analysis we excluded the data points of participants 55 and 59, who had missing time values for T0.1 and T1.1 (caused by unexpected computer shutdowns), as these may affect the outcome of the results.

Table 5 summarizes the total time and time spent on a single test. Total time was calculated using the interval between the first and last recorded activities. For the tests, we summed (1) the time spent in the IDE on the code of a specific test and (2) the time spent on the portal page of the given test. The total time spent during the sessions is very similar for the two projects. There is a roughly 17-min difference between the fastest and slowest participants, while the average participant required 45 and 46 min to finish the classification. Note that this involves every activity including the understanding of the code under test. The time results show rather large deviations between tests. The shortest times in the case of NBitcoin were probably caused by two factors. First, there were participants who gained understanding of the code under test, thus were able to quickly decide on some of the tests. Second, each method had 3 tests, and the third cases could be classified in a shorter amount of time, which emphasizes a presence of a learning curve. In contrast, participants required a rather long time period to classify some of the tests. Based on our results, a rough estimation for the average time required for classifying a single test is around 100 s.

To understand how participants managed their time budget, we analyzed their time spent on each of the possible locations (Fig. 6). These locations are the followings: portal pages of the tests, the Visual Studio windows including the test codes, class under test (CUT), other system under test (SUT) than CUT, and parameterized unit test (PUT). Note that we excluded the home page of the portal from this analysis, as it contains only a list of

Table 5 Descriptive statistics of time spent by the participants during the whole session (min) and on each test (s)

	Project	Min	Median	Avg.	Max	sd
Total (min)	NBitcoin	37.41	46.94	46.26	54.90	3.96
	Math.NET	34.88	44.57	44.52	52.61	5.24
Tests (s)	NBitcoin	1.82	117.82	146.87	818.12	121.89
	Math.NET	6.07	86.24	113.90	555.03	87.50

Fig. 6 Time spent on each of the possible locations

the cases, thus served only for navigation. The results are similar for both projects, yet there is a difference to mention. It is clear that participants mostly used the test code and the corresponding specification in the portal and in Visual Studio to understand the behavior. However, in the case of NBitcoin, they analyzed the class under test almost as much as the test code in Visual Studio. This is not the case for Math.NET, probably because participants were already familiar with the domain (combinatorics) of the tested class.

In order to gain deeper insights into the time budget, we analyzed the time required for each test (Figs. 7 and 8). We calculated this metric by summarizing five related values: the time spent in the portal page of the test, the time spent in the Visual Studio window of the test, the time spent with CUT (class under test), PUT (parameterized unit test), and SUT (other system under test than CUT) for the test currently opened in the portal. On a high-level overview, two trends can be noticed in the values. The first one is the decreasing amount of time required as participants progressed. The second factor is the first-test effect causing the first test to have longer required times for several methods.

Fig. 7 Full time spent with each test for project NBitcoin (wrong cases marked with orange)

Fig. 8 Full time spent with each test for project MathNet (wrong cases marked with orange)

Summary The analysis of the time spent by participants emphasized that they spend roughly around 100 seconds on average in the IDE to classify a particular generated white-box test. Based on the results, the developers may have to spend a noticeable amount of time to decide whether the encoded behavior is expected or not.

6 Results of the replication

6.1 RQ1: performance in classification

Figures 9 and 10 show the binary classification results of the participants in the replicated study. Taking a first look at the results yields a similar classification performance compared with the original study: there are misclassifications for both OK and WRONG tests.

For NodaTime, only one participant (ID: 22) was able to classify all generated tests correctly; meanwhile, there was no test, which was correctly classified by all participants. Also, participants with IDs 9 and 12 had a very poor classification performance with misclassification ratio of 53.33% and 40.00%, respectively. Based on the background questionnaire, participant 9 rarely writes unit tests, but has 1–2 years of work experience and has been programming for 3–5 years. Participant 12 occasionally writes unit tests, but has been programming for 4 years without any real work experience.

In the case of NetTopology, the misclassifications mostly exist for certain tests, which were seem to be hard-to-classify. Similarly to NodaTime, there was only one participant (ID: 62) who was able to classify all tests correctly. However, 5 tests were correctly classified by all of the participants. The most notable misclassification rate is found in tests T1.1 and T2.1 for the method `Extract`. The method gets a subsequence of an array while clamping the inputs to the size of the underlying structure. These cases are very similar as they have identical outcomes (an `ArgumentException`) with slightly different inputs. The described behavior of the method does not state that this method throws this type of exception; however, the description of the method that is called inside explicitly describes this behavior. Thus, these outcomes are expected for both tests.

Fig. 9 NodaTime results of the participants measured with the common binary classification measures

We analyzed the true positive (TP) and negative rate (TN) along with the Matthews correlation coefficient (MCC) for the replication study (Fig. 11).

Considering TPR, participants working with NodaTime had better performance at classifying WRONG tests correctly. Also, more than half of the participants in the NodaTime session were able to correctly classify these WRONG cases. For NetTopologySuite, the inter-quartile range is much wider, and also the median TPR is only 66.67%, which is mostly caused by the previous two widely misclassified, exception-throwing tests.

Fig. 10 NetTopologySuite results of the participants measured with the common binary classification measures

Fig. 11 Box plots of the results containing detailed metrics of binary classification

In terms of TNR, the two projects show very similar results while having much lower variance than for TPR. There are two noticeable outliers for NodaTime, who are the two participant we have already considered during the overview of the overall results (IDs 9 and 12).

The Matthews correlation coefficients (MCC) for both projects show a moderate correlation between the participants' classifications and the correct ones. However, in NetTopologySuite, there were participants with negative MCC, which yields that some of them had very poor performance in identifying whether the tests encode real expected behavior (e.g., with IDs 55 and 61, who misclassified all of the WRONG cases along with some other OK cases as well). Surprisingly, despite the poor performance, participant with ID 55 has 4 years of programming experience and often writes unit test. The other participant having low MCC has only 2 years of programming experience and only occasionally writes unit tests.

Summary In overall, participants of both replicated sessions (projects NodaTime and NetTopologySuite) were able to classify the generated tests with a good performance; however, most of the participants had some misclassified cases. Also, for NetTopologySuite, two OK tests were classified as WRONG by the larger portion of participants due to the fact that they failed to explore all of the called methods inside the code under test, and thus they were not aware of an expected behavior.

Table 6 Descriptive statistics of time spent by the participants during the whole session (min) and on each test (s)

	Project	Min	Median	Avg.	Max	sd
Total (min)	NodaTime	26.75	33.38	33.66	41.62	4.60
	NetTopologySuite	32.57	40.33	41.35	55.33	6.17
Tests (s)	NodaTime	1.02	55.83	83.17	516.05	82.73
	NetTopologySuite	1.02	61.00	86.96	519.68	78.24

6.2 RQ2: time spent for classification

For the replicated study, as the recordings were complete, we used the activity logs for the analysis of time spent by the participants. We also archived the videos for these sessions; thus, if there were validity concerns, we could annotate the videos to extract measured values.

In Table 6, we present the descriptive statistics of the total amount of time (in minutes) and time spent on a test (in seconds) in both Visual Studio and the web portal. First, each participant finished their task in the 60-min time frame. For NodaTime, most of the participants were able to classify the assigned 15 tests in around 33 min on average, which yields roughly 2.2 min of work per test. In the case of NetTopology, participants required around 41 min to finish on average, which gives an average of 2.7 min per test. The standard deviation between the finishing time was 4 and 6 min for the two projects respectively, which is a realistic difference between (1) the participants' management of time and (2) the difficulty of the projects.

To gain further understanding of participants' behavior, we analyzed the time spent in different locations during the session (shown in Fig. 12), which includes the following: pages of the portal (excluding index) along with the test codes, the class under test (CUT), other system under test (SUT), and the parameterized unit tests (PUT) opened in Visual Studio windows. The two projects show similar results in all types of times, only one exception is noticeable: the time spent with test codes in Visual Studio. This metric shows somewhat larger deviations (broader inter-quartile range) in the case of NetTopologySuite, which could indicate that the test codes were harder to read, understand, and classify in that project.

We also analyzed how participants spent their time budget on each test by summing the amount of time spent in the portal and in Visual Studio for a particular case. The results are

Fig. 12 Time spent on each of the possible locations during the replicated study

Fig. 13 Full time spent with each test for NodaTime (wrong cases marked with orange)

shown in Figs. 13 and 14 for projects NodaTime and NetTopologySuite, respectively. The first phenomenon that can be noticed is that there is a learning curve for each method, which can indicate that the participants tried to understand the behavior of a method during the analysis of the first corresponding test (similarly to the original study). The second effect is the influence of WRONG tests (marked with orange on both figures); there are no clear, observable differences in the time spent for WRONG cases compared with the correct ones. Also, the amount of time spent for a test decreases slowly on average from the first tests to the last one. This may be caused by the learning curve of classification task similarly to the original study.

Summary During the analysis of RQ2 for the replicated study, we found that most of the participants spent roughly 55–60 s with each test and finished their whole classification task

Fig. 14 Full time spent with each test for NetTopologySuite (wrong cases marked with orange)

in around 33–40 min. Classification of a single test in 60 s could be a noticeable amount of effort in a large-scale software having hundreds of generated white-box tests.

7 Conclusions across studies

We performed our study along with a replication examining participants' performance in classification of generated white-box tests. For that, we changed the projects used, the faults injected, and also we recruited new participants from a population having same characteristics.

7.1 RQ1: performance in classification

In RQ1, we investigated how participants perform in classifying the generated white-box test in terms of their correctness. Both studies showed that participants cannot perfectly perform the classification task. Moreover, they also tend to commit mistakes both in classifying the WRONG and the OK cases as well.

In the original study, we found that the Matthews correlation coefficient (indicating the correlation between the correct and the given classification) was between only around 0.4 and 0.55 for most of the participants. This can be considered as a moderate positive correlation. In the replicated study, the results have shown that the MCC was roughly between 0.55 and 0.75. Combining and averaging these results together will yield only a moderate classification performance of 0.56. These results have also shown that the change of project for the replication (with keeping participant's overall knowledge on the same level) may have effects on the classification performance. Also, we discovered that in both the original and replicated study, there were participants who had negative MCC values, meaning very poor classification performance.

7.2 RQ2: time spent for classification

In terms of participants' time management, the original study has shown that they usually require roughly 100 s to classify a single test on average. In the replicated study, we found that they require an average of 60 s to classify a case. This yields a difference in the average time required; however, the minimum (~1 s) and maximum (around 500 s) were common for the two studies. These results strengthens that more than 1 min is usually required for a participant to classify a single generated white-box test.

In both studies, we investigated that in which location participants spend their time. Both studies suggested that they mostly used the portal to inspect the test code and the corresponding method specification. In the original study, the participants spent a vast amount of time to inspect the generated test code in Visual Studio, more time than examining the class under test. Contrasting to this, the participants of the replicated study spent more time with the class under test than with the generated tests. This may indicate that the original study had tests that were more complex and harder to understand. In other parts of the code (system under test and the parameterized unit test) participants of both studies spent similar amount of time.

Finally, we examined the time spent with each of the tests by summing their times in the portal and in Visual Studio. We observed three learning curve effects in both of the studies: (1) task learning, (2) class learning, and (3) method learning (first test) effects. The task and class learning had similar effects which is that the classification required less time

as participants progressed from method to method. The method learning (first test) effect can be observed in the time spent for a single method: for most of the participants, the classification of the first test required much more time than the second, and also the second test mostly required more time to classify than the third.

8 Discussion

White-box test generators can be applied for many testing *use cases* throughout the development lifecycle. When using it for *security checking* purposes, the outcome of the generated tests and their tendency to encode faults are less likely to be in focus; the gist is to bring the system under test into an unsecure, faulty state. White-box tests are also widely applied for *regression testing* in order to obtain differences between the behaviors of two program versions. In this case, the outcome of a given test may differ between versions. However, if one would like to investigate the root cause of the differences, the tests and their outcomes must be scrutinized and classified to decide whether the difference is due to a new feature or one of the old or new versions is faulty. Finally, a similar classification must be performed when *testing new features* of a program using white-box tests (which was the focus of the current study).

8.1 Implications of the results

The results for RQ1 in both studies showed that classifying the correctness of generated white-box tests could be a challenging and noticeably time-consuming task (also depending on the project). The median of misclassification rate in the original study was 33% for WRONG tests and 25% for OK tests. In terms of the replication, the misclassification rate in WRONG and OK tests were 27% and 14%, respectively. Both could be caused by several factors such as (1) the misunderstanding of the described behavior, (2) the misunderstanding of source code behavior, (3) the misunderstanding of generated tests, or even (4) the underlying fault types in the software under test and (5) the participants' experience.

For RQ2, our results showed that participants spent significant amount of time to understand the encoded behavior and functionality in the tests. In a more realistic setting, there could be ten times more generated tests for a single class. The classification should be also performed for all of these white-box cases, which may require vast amount of time and effort. This result indicates that developers and testers could spend a noticeable amount of time with the classification, which may reduce the time advantage provided by generated white-box tests.

8.2 Recommendations based on the study results

This section collects our recommendations for enhancing the usage of white-box test generators based on insights from observing the participants, and analyzing the answers and suggestions in the exit survey.

8.2.1 Insights from participants' behavior

By watching and coding the original study's screen capture videos, we gained important insights into the user activities and behaviors during the classification of generated white-box tests.

As expected, many participants employed debugging to examine the code under test. They mostly checked the exceptions being thrown, the parameterized unit tests for the test methods, and assertions generated into the test code. This emphasizes the importance of debugging as a tool for investigating white-box test behavior. Almost all of the participants executed the tests to check the actual outcomes. Based on this, it seems that generated tests codes are not clearly indicating their outcomes. Some cases contained unexpected exceptions, and thus failed after execution, which may have confused some participants. These insights show that there is a clear demand for a tool, which is able to support this classification task in various ways.

Another interesting insight we obtained is that some participants spent only seconds with the examination of the last few tests. This could pinpoint that they either gained understanding of the code under test by the end of the session (i.e., learning factor), or they got tired by the continuous attention required during the classification task. The latter could also emphasize the need for a supporting tool for this task.

8.2.2 Results from exit survey

The participants in our study filled an exit survey at the end of the sessions. They had to answer both Likert-scaled and textual questions.

Original study The results (shown in Fig. 15) for the agreement questions yielded that participants had enough time to understand the class under test and to review the generated tests. Most of them also answered that it was easy to understand the class and the tests. Also, they agreed that the generated tests were difficult to read; however, the answers were almost equally distributed for the questions about the difficulty of the task and the confidence in their answers. Their answers show that they are mostly not very confident about

Fig. 15 Likert-scale exit survey answers in the original study

their own answers. Based on the feedback about the time required and task's difficulty, our study design was *suitable for its purpose* (i.e., no complains were made about the selected variables of the study design).

Replication As shown in Fig. 16, the largest portion of participants stated that they had enough time both to understand the class under test and to finish the classification task. Also, most of them agreed with the facts that the classes and the tests were easy to understand. Moreover, as opposed to the original study, the majority of the participants were certain that they chose the right answer. In their answers, most of them stated that the generated tests are difficult to read, too short, but had too many assertions without useful meaning. Forty-two percent of the participants were uncertain whether it was easy to select tests with wrong assertions.

8.2.3 Suggestions by participants

In their textual answers participants mentioned the difficulties in reviewing the tests and gave several suggestions to improve the test code. Some of these were also reported in the literature (Tillmann et al. 2014). We selected the most descriptive ones.

- “It was hard to decide whether a test is OK or wrong when it tests an unspecified case (e.g. comparing with null, or equality of null).”
- “Distinguishing between the variables was difficult (assetMoney, assetMoney1, assetMoney2).”
- “Tests should compare less with null and objects with themselves.”
- “I think that some assertions are useless, and not asserting ‘real problems,’ just some technical details.”
- “I would somehow write a short cause in comment next to an assertion about why the generator thinks that assertion needs to be there.”

Fig. 16 Likert-scale exit survey answers in the replication study

- “Generated tests are not separated into Arrange, Act, Assert and should create more private methods for these concerns.”
- “Generate comments into tests describing what happening.”

8.2.4 Summary of recommendations

Based on the results and the feedbacks, our recommendations for improving test generators to help developers and testers with generated assertions consist of the followings.

- Instead of using the `assert` keyword, test generators could use a different keyword (e.g., `observed`) emphasizing that these outputs are just recorded observations and some kind of validation is needed.
- Similarly, it could be confusing for the developers that most of the generated tests pass and are “green.” Some test execution engines allow to mark test runs as inconclusive (meaning neither pass nor fail). If no other test oracles are available, then white-box test generator could use inconclusive as the default outcome.
- Assertions could be categorized and arranged by the types of checks they perform (e.g., observed values, sanity checks).
- Assertions could have a short and descriptive comment about why they were generated.
- Generated tests having null inputs could be distinguished from the others.
- The generated tests could have short descriptions about what they are checking and covering, similarly to Panichella et al. (2016b).
- Generated tests could contain variables with more meaningful names, e.g., Daka et al. (2017) for test names.
- The generated tests could employ the Arrange, Act, Assert pattern in the structure of generated tests.
- The tests could contain intra-line comments that describe what the given line is responsible for.

8.3 Towards a theory of classification

Based on the results and insights of the study we now revisit the classification problem presented in the introduction, clarify its details and challenges, and identify possible research questions for future studies.

As the results of this study showed, classifying generated white-box tests is a non-trivial task. The underlying issue is that the classification is affected by many factors and some combinations are especially puzzling (e.g., when the selected inputs should raise a specific exception, but the tests observe and encode a different exception and report a pass outcome). We are not aware of any systematic discussion about the topic, and there seems to be no consensus in the literature. For example, various test generators handle exceptions and assign pass and fail outcomes differently to the same type of generated tests. We believe that further research and a fundamental theory is needed for understanding generated white-box tests.

As an initial contribution towards this goal, we hereby outline a *preliminary conceptual framework* for the classification of white-box tests (summarized in Table 7). We identified the following three main, boolean conditions that affect the classification task.

- *Shall raise exception*: Based on the specification, the selected inputs shall raise an exception (T, F).

Table 7 Possible cases of the classification task

ID	Conditions			Test encodes	User action	Classification	Test outcome
	Shall raise exception	Fault is triggered	Exception is raised				
C1	F	F	F	Expected behavior	Acknowledge the test	OK	Pass
C2	F	F	T	–	–	–	–
C3	F	T	F	Unexpected behavior	Realize faulty impl.	WRONG	Pass
C4	F	T	T	Unexpected exception	Recognize the fault-indicating exception	WRONG	Pass or fail
C5	T	F	F	–	–	–	–
C6	T	F	T	Expected exception	Acknowledge the test	OK	Pass or fail
C7	T	T	F	Unexpected behavior	Realize an exception is missing	WRONG	Pass
C8	T	T	T	Unexpected exception	Recognize the invalid exception	WRONG	Pass or fail

- *Fault is triggered*: Execution of the generated test triggers a fault (T, F).
- *Exception is raised*: Execution of the generated test with the given inputs raises an exception (T, F).

The eight combinations of the above conditions represent different cases that should be analyzed. These cases determine what the tests are encoding, what the user should do, and result of the classification. (Note, we will describe later the last column in the table.)

- *Test encodes*: what kind of observed behavior does the generated test encode (expected behavior, unexpected behavior, expected exception, unexpected exception).
- *User action*: In most cases, the user shall acknowledge that the test encodes an expected behavior or expected exception (and hence gain confidence that the implementation is correct w.r.t. the specification). However, when the selected inputs in the test trigger a fault, then the user shall realize that the encoded behavior is unexpected (i.e., the observed result is unexpected, an expected exception is missing or an unexpected exception was observed).
- *Classification*: classifying the test whether it encodes an expected or unexpected behavior (OK, WRONG).

The detailed description of each case listed in the rows of Table 7 is as follows.

- C1 Describes that no exception is thrown, no fault is triggered in the implementation and thus the user shall acknowledge the expected behavior and shall mark the generated test as OK.
- C2 Not possible, because the test cannot raise an exception if the expected behavior is no exception and no fault is triggered.

- C3 The generated white-box test triggers a fault, but does not raise any exception (e.g., there is an algorithmic fault in the code). The implementation returns an incorrect value that gets encoded in the test. In this case, the user shall recognize that an unexpected behavior was observed with respect to the specification and shall classify this test as **WRONG**.
- C4 The expected behavior is no exception, but due to an underlying fault in the implementation, an exception is thrown when running the test. This exception is usually encoded in the test in some form. Thus, the classification shall mark this test as **WRONG**.
- C5 Not possible, because it represents that a generated test executes a fault-free code path that shall raise an exception, but the exception is not triggered during execution.
- C6 The generated test raises an exception, which is expected according to the specification. The case shall be classified as **OK** by the user acknowledging the correct exception.
- C7 The generated test inputs shall trigger an exception. However, the generated test does not raise an exception due to executing a code path containing a fault. The user must recognize this contradiction, and shall mark the test as **WRONG**.
- C8 The generated inputs shall raise an exception, but a different kind of exception is raised due to a fault. In this situation, the user shall recognize that the observed exception is unexpected, and the generated test shall be classified as **WRONG**.

Example In Section 1, we showed a motivating example of the classification problem. The details of those tests are found in Table 1. Test T1 asserts the expected behavior (case C1 in Table 7). The second test encodes an unexpected value, and the user had to classify it as **WRONG** (C3). We provide four additional generated tests that cover the remaining cases (Table 8). Test T3 passes – 1 as the number of elements to be summed from the array. The generated result of this test is 0, which is an unexpected behavior: the code should have thrown an exception. Thus, the test is classified as **WRONG** (C7). Test T4 tries to execute the method with a single-element array with zero as start index and the number of elements to sum. The result should be 0 according to the specification; however, the actual test triggers an `ArgumentException` in the implementation for these inputs (C4) (**WRONG**). Test T5 passes an array with five elements, but tries to sum up six. These are invalid inputs, and the test throws an `ArgumentException` and can be marked as **OK** (C6). Finally, test T6 uses null as the array parameter, which results in a `NullReferenceException`. However, this is not the expected behavior (`ArgumentException` should have been raised due to the invalid inputs). Therefore, it can be classified as **WRONG** (C8).

Table 8 A possible set of generated white-box tests for the motivating example

ID	Start	Num	a	Result	Classification	Classification mapping
T1	0	0	4, 5, 6, 7, 8	0	OK	C1
T2	0	4	4, 5, 6, 7, 8	15	WRONG	C3
T3	0	– 1	4, 5, 6, 7, 8	0	WRONG	C7
T4	0	0	4	<code>ArgumentException</code>	WRONG	C4
T5	0	6	4, 5, 6, 7, 8	<code>ArgumentException</code>	OK	C6
T6	0	0	Null	<code>NullReferenceException</code>	WRONG	C8

Test outcome We have not discussed yet one crucial factor that can greatly affect whether the user can classify the test correctly, namely the outcome of the generated test. When the tests are executed, usually two outcomes are used:³

- *pass*: test method finishes successfully, i.e., no assertion is violated, and no exception is raised or the raised exception was defined as an expected exception for the given test,
- *fail*: an assertion is violated, or an exception is raised that was not defined as expected in the test code.

Assigning the outcome for some of the cases is straightforward (see the last column in Table 7). For example, C1 will pass. C3 and C7 will also pass, as the test generator does not know the expected behavior and can only generate asserts capturing the observed behavior. However, assigning the outcome for the cases involving exceptions is not trivial, and sometimes the outcome can mislead users; we could also observe this effect in the results of the replicated study.

Test generator tools handle the exception-related cases differently and use different constructs and rules to encode exceptions. To demonstrate this issue, we generated tests for the motivating example using three tools (see Appendix 2 for the source code of the generated tests).

- *IntelliTest* uses a simple strategy when assembling generated tests: if the exception is explicitly thrown inside the class under test, then it is defined as an expected exception for the generated test, and thus the test will pass during execution. In every other case, if an exception occurs (e.g., when dividing by zero or using a null pointer), it will generate a failing test (see Listing 3 for both types). In both cases, no asserts are generated.
- *EvoSuite* (Fraser and Arcuri 2013) detects every occurring exception and wraps the call to the unit under test into a try-catch block matching for the given type of exception. This will cause every exception-throwing case to pass (see Listing 4). These tests only fail if the observed exception is not raised during execution.
- *Randoop* (Pacheco et al. 2007) tries to classify generated tests into error-revealing or expected tests. Error-revealing tests indicate a potential error and fail when executed. Expected tests form regression tests that capture the current behavior (values returned or exceptions thrown), and they currently pass. The tool uses several heuristics during the automated classification.⁴ By default, exceptions are treated as expected behavior, and these tests pass, but this can be customized (see Listing 5).

As these examples presented, various test generator tools assign test outcomes sometimes in conflicting ways to the same tests. Therefore, Table 7 contains “pass or fail” for the C4, C6, and C8 cases because the test outcome will depend on the actual tool. Ideally, the test outcome should not affect the classification task (as the same exception type is present in every tool’s generated tests). However, the results of the study showed that users could get confused even if the unexpected exception is explicitly captured in the generated test.

Future research directions Research on white-box test generators is typically concerned with the number of faults the generated tests could detect, and evaluate this capability using proxy measures like the number of faults encoded or mutation score. However, the analysis of the participants’ answers and behavior suggests that these proxy measures could overestimate the number of faults that are detected by the humans using these generated tests. Therefore, instead of asking “*Do this test encode a fault?*,” a more suitable question could

³Several test execution engines support other outcomes, but we are not aware of any test generation tool using these.

⁴See Randoop’s online manual for the specific rules.

be “Does the user recognize that there is a fault with the help of this test?” It is sometimes not enough to encode a fault in the test. For example, if an unexpected exception was observed and captured in the test code, but the test passes, then the user can easily overlook the fault. Therefore, a test should not only be fault-encoding, but it should also help to detect the fault. We recommend using the term “appropriate test” to describe this aspect.

As our preliminary framework showed, there are several questions for which there is no clear consensus yet. We identified the following questions that could direct future research and help to build a theory for understanding generated white-box tests.

- When should a generated white-box test pass or fail?
- What should a test contain if an exception is raised for the selected inputs?
- When is a generated test useful for developers?
- How can the generated tests indicate that the implementation is possibly faulty?

9 Conclusions

This paper presented an exploratory study on how developers classify generated white-box tests. The study and its replication performed in a laboratory setting with 106 graduate students resembled a scenario where junior developers having a basic understanding of test generation had to test a class in a larger, unknown project with the help of a test generator tool. The results showed that participants tend to incorrectly classify tests both encoding expected and unexpected behavior, even if they did not consider the task difficult. Moreover, it turned out that the classification may require a considerable amount of time, which may slow down the software testing process. When discussing the results, we proposed a preliminary framework for the classification problem and identified future research questions. The implication of the results is that (1) this often overlooked validation of the generated tests is an essential and non-trivial step in the test generation process and (2) the actual fault-finding capability of the test generator tools could be much lower than reported in technology-focused experiments. Thus, we suggest taking this factor into account in future studies or when using white-box test generators in practice.

An experimental study always has limitations. We collected important context variables that could affect the classification performance (e.g., experience, source code access), and defined the levels chosen in the current study that collectively reflect one possible scenario. As in our study all variables had fixed levels, this naturally limits its validity. Future studies altering these settings could help to build a “body of knowledge” (Basili et al. 1999). Moreover, designing a study where participants work on a familiar project or perform regression testing would be important future work. Thus, we made available our full dataset, coded videos and lab package to support further analyses or replications (Honfi and Micskei 2018).

Acknowledgements We would like to thank the anonymous reviewers, István Majzik, Vince Molnár, and Dániel Varró for their valuable remarks.

Funding Information Open access funding provided by Budapest University of Technology and Economics (BME). This work was partially supported by the BME-Artificial Intelligence FIKP grant of EMMI (BME FIKP-MI/SC) and a scholarship from Pro Progressio Foundation.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

Appendix 1: Details of injected faults for both study

Details of the injected faults are listed in Table 9.

Table 9 Selected faults for the study. The upper six are the faults for the original study, while the other six are selected for the replication

Project	Name	ODC category	Original snippet	Faulty snippet
NBitcoin	Extraneous assignment using another variable	Assignment	–	dec = dec + divisibility;
	Missing function call	Algorithm	return _Quantity .Equals(other.Quantity);	return true;
	Algorithm - large modification	Function	left.Quantity <= right.Quantity;	left.Id.Bytes.Length <= right.Id.Bytes.Length;
Math.NET	Missing OR sub-expression used as branch condition	Check	k < 0 n < 0 k > n	k < 0 n < 0
	Wrong logical expression used as branch condition	Check	k < 0 n < 0 k > n	k < 0 && n < 0 k > n
	Wrong arithmetic expression in parameter of function call	Interface	SpecialFunctions .FactorialLn(n + k - 1)	SpecialFunctions .FactorialLn(n - k + 1)
	Wrong arithmetic expression in parameter of function call	Interface	Add(result, seconds*scalar)	Add(result,seconds/scalar)

Table 9 (continued)

Project	Name	ODC category	Original snippet	Faulty snippet
NodaTime	Missing OR sub-expr in expression used as branch condition	Check	months != 0 weeks != 0	months != 0
	Wrong logical expression used as branch condition	Check	Months != 0 Years != 0	Months != 0 && Years != 0
	Wrong value assigned to variable	Assignment	–	npts += 1;
NetTopology	Missing function call	Algorithm	pts[i].CompareTo(pts[j])	-1
	Large modification in the algorithm	Function	return false	return true

Appendix 2. Generated tests for the example

Source code of tests generated by IntelliTest (Listing 3), EvoSuite (Listing 4), and Randoop (Listing 5).

```

[TestMethod]
1320 [ExpectedException(typeof(ArgumentException))]
    public void SumTestThrowsArgumentException246()
    {
        int i;
        int[] ints = new int[1];
1325     ints[0] = 4;
        i = this.SumTest(0, 0, ints);
    }

[TestMethod]
1330 [PexGeneratedBy(typeof(CalculatorTest))]
    [PexRaisedException(typeof(NullReferenceException))]
    public void SumTestThrowsNullReferenceException616()
    {
        int i;
1335     i = this.SumTest(0, 0, (int[])null);
    }

```

Listing 3 IntelliTest-generated tests with IDs T4 and T6 generated for the motivating example

```

@Test(timeout = 4000)
1340 public void test12() throws Throwable {
    int[] intArray0 = new int[1];
    // Undeclared exception
    try {
        Calculator.sum(0, 0, intArray0);
        fail("Expecting exception: IllegalArgumentException");
    } catch (IllegalArgumentException e) {
1345         verifyException("Calculator", e);
    }
}

@Test(timeout = 4000)
1350 public void test02() throws Throwable {
    int[] intArray0 = null;
    // Undeclared exception
    try {
        Calculator.sum(0, 0, (int[]) null);
1355     fail("Expecting exception: NullPointerException");

    } catch (NullPointerException e) {
        verifyException("Calculator", e);
    }
1360 }

```

Listing 4 EvoSuite-generated tests with IDs T4 and T6 generated for the motivating example

```

@Test
public void test01() throws Throwable {
    int[] intArray3 = new int[] { '4' };
    try {
1365         int int4 = Calculator.sum(0, 0, intArray3);
        Assert.fail("Expected exception of type
                    java.lang.IllegalArgumentException; message: Too long");
    } catch (IllegalArgumentException e) { }
    Assert.assertNotNull(intArray3);
1370 }

// default classification of handling
// NullPointerException changed
// --npe-on-null-input=ERROR
1375 @Test
public void test02() throws Throwable {
    if (debug)
        System.out.format("%n%s%n", "ErrorTest0.test02");
    int[] intArray5 = null;
1380    Calculator.Sum(0, (int) (short) 100, intArray5);
}

```

Listing 5 Randoop-generated tests with IDs T4 and T6 generated for the motivating example

References

- Almaghairbe, R., & Roper, M. (2016). Separating passing and failing test executions by clustering anomalies. *Software Quality Journal*, 25(3), 1–38. <https://doi.org/10.1007/s11219-016-9339-1>.
- Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8), 1978–2001. <https://doi.org/10.1016/j.jss.2013.02.061>.
- Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S. (2015). The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5), 507–525. <https://doi.org/10.1109/TSE.2014.2372785>.
- Basili, V., & Selby, R. (1987). Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, SE-13(12), 1278–1296. <https://doi.org/10.1109/TSE.1987.232881>.
- Basili, V., Shull, F., Lanubile, F. (1999). Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4), 456–473. <https://doi.org/10.1109/32.799939>.
- Bounimova, E., Godefroid, P., Molnar D (2013). Billions and billions of constraints: Whitebox fuzz testing in production. In: International conference on software engineering, IEEE, ICSE, pp. 122–131. <https://doi.org/10.1109/ICSE.2013.6606558>.
- Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M.Y. (1992). Orthogonal defect classification—a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11), 943–956. <https://doi.org/10.1109/32.177364>.
- Claessen, K., & Hughes, J. (2000). Quickcheck: A lightweight tool for random testing of haskell programs. In: International Conference on Functional Programming, ACM, ICFP, pp. 268–279. <https://doi.org/10.1145/351240.351266>.
- Daka, E., Rojas, J.M., Fraser, G. (2017). Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?. In: Int. symp on software testing and analysis, ISSTA, pp 57–67. <https://doi.org/10.1145/3092703.3092727>.
- Dieste, O., Aranda, A.M., Uyaguari, F., Turhan, B., Tosun, A., Fucci, D., Oivo, M., Juristo, N. (2017). Empirical evaluation of the effects of experience on code quality and programmer productivity: an exploratory study. *Empirical Software Engineering*, 22(5), 1–86. <https://doi.org/10.1007/s10664-016-9471-3>.
- Duraes, J.A., & Madeira, H.S. (2006). Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11), 849–867. <https://doi.org/10.1109/TSE.2006.113>.
- Easterbrook, S., Singer, J., Storey, M.A., Damian, D. (2008). Selecting empirical methods for software engineering research, Springer, chap 11, pp. 285–311. https://doi.org/10.1007/978-1-84800-044-5_11.

- Enoiu, E.P., Causevic, A., Sundmark, D., Pettersson, P. (2016). A controlled experiment in testing of safety-critical embedded software. In: Int. conf. on software testing, verification and validation, ICST, pp. 1–11. <https://doi.org/10.1109/ICST.2016.15>.
- Enoiu, E.P., Causevic, A., Sundmark, D., Pettersson, P. (2017). A comparative study of manual and automated testing for industrial control software. In: Int. conf. on software testing, verification and validation, ICST, pp 412–417. <https://doi.org/10.1109/ICST.2017.44>.
- Falessi, D., Juristo, N., Wohlin, C., Turhan, B., Münch, J., Jedlitschka, A., Oivo, M. (2018). Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering*, 23(1), 452–489. <https://doi.org/10.1007/s10664-017-9523-3>.
- Fraser, G., & Arcuri, A. (2013). Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2), 276–291. <https://doi.org/10.1109/TSE.2012.14>.
- Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F. (2015). Does automated unit test generation really help software testers? a controlled empirical study. *ACM Trans Softw Eng Methodol*, 24(4), 23:1–23:49. <https://doi.org/10.1145/2699688>.
- Friard, O., & Gamba, M. (2016). Boris: a free, versatile open-source event-logging software for video/audio coding and live observations. *Methods in Ecology and Evolution*, 7(11), 1325–1330. <https://doi.org/10.1111/2041-210X.12584>.
- Honfi, D., & Micskei, Z. (2018). Classifying generated white-box tests: An exploratory study [Data set]. <https://doi.org/10.5281/zenodo.2596044>.
- Juristo, N., Moreno, A.M., Vegas, S. (2004). Reviewing 25 years of testing technique experiments. *Empirical Software Engineering*, 9(1), 7–44. <https://doi.org/10.1023/B:EMSE.0000013513.48963.1b>.
- Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G. (2014). Are mutants a valid substitute for real faults in software testing?. In: Int. symp. on foundations of software engineering, ACM, FSE, pp. 654–665. <https://doi.org/10.1145/2635868.2635929>.
- Ko, A.J., LaToza, T.D., Burnett, M.M. (2013). A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 20(1), 110–141. <https://doi.org/10.1007/s10664-013-9279-3>.
- Kracht, J., Petrovic, J., Walcott-Justice, K. (2014). Empirically evaluating the quality of automatically generated and manually written test suites. In: Int. Conf on Quality Software, QSIC, pp. 256–265. <https://doi.org/10.1109/QSIC.2014.33>.
- MathNET (2017). Math.NET numerics. <https://numerics.mathdotnet.com/>.
- McMinn, P., Stevenson, M., Harman, M. (2010). Reducing qualitative human oracle costs associated with automatically generated test data. In: Int. Workshop on Software Test Output Validation, ACM, STOV '10, pp 1–4. <https://doi.org/10.1145/1868048.1868049>.
- Myers, G.J. (1978). A controlled experiment in program testing and code walkthroughs/inspections. *Commun. ACM*, 21(9), 760–768. <https://doi.org/10.1145/359588.359602>.
- NBitcoin (2017). NBitcoin. <https://github.com/MetacoSA/NBitcoin>.
- NetTopologySuite (2018). NetTopologySuite. <https://github.com/NetTopologySuite/NetTopologySuite>.
- Nguyen, C.D., Marchetto, A., Tonella, P. (2013). Automated oracles: An empirical study on cost and effectiveness. In: Foundations of software engineering, ACM, ESEC/FSE, pp. 136–146. <https://doi.org/10.1145/2491411.2491434>.
- NodaTime (2018). NodaTime. <https://github.com/nodatime/nodatime>.
- Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T. (2007). Feedback-directed random test generation. In: Int. conf on software engineering, ICSE, pp. 75–84. <https://doi.org/10.1109/ICSE.2007.37>.
- Panichella, S., Panichella, A., Beller, M., Zaidman, A., Gall, H.C. (2016a). The impact of test case summaries on bug fixing performance: An empirical investigation. In: Int. conf on software engineering, ICSE, pp. 547–558. <https://doi.org/10.1145/2884781.2884847>.
- Panichella, S., Panichella, A., Beller, M., Zaidman, A., Gall, H.C. (2016b). The impact of test case summaries on bug fixing performance: An empirical investigation. In: Int. conf on software engineering, ICSE, pp. 547–558. <https://doi.org/10.1145/2884781.2884847>.
- Papadakis, M., Henard, C., Harman, M., Jia, Y., Le Traon, Y. (2016). Threats to the validity of mutation-based test assessment. In: Int. symp. on software testing and analysis, ACM, ISSA, pp. 354–365. <https://doi.org/10.1145/2931037.2931040>.
- Pastore, F., & Mariani, L. (2015). Zoomin: Discovering failures by detecting wrong assertions. In: Int. conf. on software engineering, IEEE, ICSE, pp. 66–76. <https://doi.org/10.1109/ICSE.2015.29>.
- Pastore, F., Mariani, L., Fraser, G. (2013). Crowdoracles: Can the crowd solve the oracle problem?. In: Int. conf. on software testing, verification and validation, ICST, pp. 342–351. <https://doi.org/10.1109/ICST.2013.13>.
- Powers, D.M.W. (2011). Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *International Journal of Machine Learning Technology*, 2(1), 37–63.

- Prado, M.P., & Vincenzi, A.M.R. (2018). Towards cognitive support for unit testing: A qualitative study with practitioners. *Journal of Systems and Software*, 141, 66–84. <https://doi.org/10.1016/j.jss.2018.03.052>.
- R. Core Team (2016). R: A language and environment for statistical computing, R Foundation for Statistical Computing, Vienna. <https://www.R-project.org/>.
- Ramler, R., Winkler, D., Schmidt, M. (2012). Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code? In: EUROMICRO Conf. on Software Engineering and Advanced Applications (SEAA), pp. 286–293. <https://doi.org/10.1109/SEAA.2012.42>.
- Rojas, J.M., Fraser, G., Arcuri, A. (2015). Automated unit test generation during software development: A controlled experiment and think-aloud observations. In: Int. symp. on software testing and analysis, ACM, ISSTA, pp. 338–349. <https://doi.org/10.1145/2771783.2771801>.
- Shahrokni, A., & Feldt, R. (2013). A systematic review of software robustness. *Information and Software Technology*, 55(1), 1–17. <https://doi.org/10.1016/j.infsof.2012.06.002>.
- Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMin, P., Arcuri, A. (2015). Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges. In: Int. conf. on automated software engineering, ACM, ASE, pp. 201–211. <https://doi.org/10.1109/ASE.2015.86>.
- Shamshiri, S., Rojas, J.M., Galeotti, J.P., Walkinshaw, N., Fraser, G. (2018). How do automatically generated unit tests influence software maintenance? In: Int. conf on software testing, verification and validation, ICST.
- Sjøberg, D., Hannay, J., Hansen, O., Kampenes, V., Karahasanovic, A., Liborg, N.K., Rekdal, A. (2005). A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, 31(9), 733–753. <https://doi.org/10.1109/TSE.2005.97>.
- Staats, M., Hong, S., Kim, M., Rothermel, G. (2012). Understanding user understanding: Determining correctness of generated program invariants. In: Int. symp. on software testing and analysis, ACM, ISSTA, pp. 188–198. <https://doi.org/10.1145/2338965.2336776>.
- Tillmann, N., & de Halleux, J. (2008). Pex—White Box Test Generation for.NET. In Beckert, B., & Hähnle, R. (Eds.) *Tests and Proofs, LNCS*, (Vol. 4966 pp. 134–153): Springer. https://doi.org/10.1007/978-3-540-79124-9_10.
- Tillmann, N., & Schulte, W. (2005). Parameterized unit tests. In: Foundations of Software Engineering, ESEC/FSE, pp. 253–262. <https://doi.org/10.1145/1081706.1081749>.
- Tillmann, N., de Halleux, J., Xie, T. (2014). Transferring an automated test generation tool to practice: From Pex to Fakes and Code Digger. In: Int. conf on automated software engineering, ASE, pp. 385–396. <https://doi.org/10.1145/2642937.2642941>.
- Wang, X., Zhang, L., Tanofsky, P. (2015). Experience report: How is dynamic symbolic execution different from manual testing? A study on KLEE. In: int. symp. on software testing and analysis, ACM, ISSTA, pp 199–210. <https://doi.org/10.1145/2771783.2771818>.
- Wohlin, C., & Aurum, A. (2015). Towards a decision-making structure for selecting a research design in empirical software engineering. *Empirical Software Engineering*, 20(6), 1427–1455. <https://doi.org/10.1007/s10664-014-9319-7>.
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A. (2012). *Experimentation in software engineering*. Berlin: Springer. <https://doi.org/10.1007/978-3-642-29044-2>.
- Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software Testing. Verification and Reliability*, 22(2), 67–120. <https://doi.org/10.1002/stvr.430>.

Dávid Honfi is a PhD student at the Doctoral School of Informatics, Budapest University of Technology and Economics. His research area is focusing on the practical application of symbolic execution-based white-box test generation. He has developed multiple tools (e.g., SEViz and AutoIsolator) aiding the practical applicability of white-box test generators and he is the author of several international publications on this topic.

Zoltán Micskei received the MSc and PhD degree in software engineering from the Budapest University of Technology and Economics in 2005 and 2013, respectively. He is currently an associate professor of the same university. His research interests include software testing with focus on test generation and empirical studies. He participated in several EU research projects and is currently the principal investigator at his university of the ADVANCE project. He served as the general co-chair for DISC'19, as PC member for various international conferences (ER, ICTSS, HASE...), and as an external reviewer for international journals. He was a visiting researcher at LAAS-CNRS, Toulouse.