Generating Understandable Unit Tests through End-to-End Test Scenario Carving

Amirhossein Deljouyi
Delft University of Technology
The Netherlands
a.deljouyi@tudelft.nl

Andy Zaidman
Delft University of Technology
The Netherlands
a.e.zaidman@tudelft.nl

***Abstract*—Automatic unit test generators such as EvoSuite are able to automatically generate unit test suites with high coverage. This removes the burden of writing unit tests from developers, but the generated tests are often difficult to understand for them. In this paper, we introduce the *MicroTestCarver* approach that generates unit tests starting from manual or scripted end-to-end (E2E) tests. Using carved information from these E2E tests, we generate unit tests that have meaningful test scenarios and contain actual test data. When we apply our *MicroTestCarver* approach, we observe that 85% of the generated tests are executable. Through a user study involving 20 participants, we get indications that tests generated with *MicroTestCarver* are relatively easy to understand.**

***Index Terms*—Automatic Test Generation, Carving and Replaying, Readability, Understandability, Unit Testing**

I. Introduction

In the software-enabled world that we live in, reliable and correct software is crucial [1]. As such, software quality assurance has become a critical asset in the software engineer's toolbox. For example, automated testing in the form of unit tests has become an important ingredient to ensure high quality software [2]. While the importance of testing is generally acknowledged, writing tests is seen as a tedious and time-consuming task [3]–[6]. To relieve developers and/or testers of the burden of writing test cases, the research community has invested in developing and evaluating automatic test generation approaches [7]–[10]. Two important test generation approaches are Randoop [11] and EvoSuite [9]. For example, EvoSuite is a search-based test case generation tool that uses genetic algorithms to construct a test suite [12]. While the results in terms of coverage are very convincing, industrial case studies have indicated that the understandability of the generated test cases is a considerable limitation [13]. The understandability is hampered by the difficulty to follow the scenario depicted in the test case, the unclear test data, and the meaningfulness of generated assertions [14], [15].

Motivating example: Consider Listing 1 which depicts both a manually written JUnit test and an EvoSuite-generated JUnit test. When we compare the scenarios of these test cases, the manually written one is seemingly easier to understand. For example, when we zoom in on line 3 of and 11 of Listing 1, we can see that in the former case we are constructing an object to represent *rainy weather*, while the latter case does not correspond to an actual weather situation (*"S:q$ZHC!0J3"*). Moreover, in Line 6 a REST API response is mocked, which checks if the weather that is returned by the mock corresponds to an expected weather situation. In the case of the generated test in lines 12–14, the test checks whether the object is null, and checks the result of the `toString()` method, albeit with constants that do not make sense in the domain.

In this paper we present an approach that carves information from end-to-end (E2E) tests to generate understandable unit tests. Resting on the assumption that E2E tests are available for the system, during carving we extract the execution trace from a running E2E tests, including the order of calls and the actual inputs. Using that information, we gather scenarios that are meaningful in the domain, and (parameter) values to instantiate objects and pass to method calls.

When could our approach be of use to software engineers? In a situation where a system is evolving and mainly has E2E tests, e.g., Selenium tests, a software engineer might decide that it is good to also have lower-level test cases, e.g., unit tests, to act as a safety net during evolution. This safety net will enable faster fault localisation than a typical E2E test can. In this scenario, a software engineer can use our approach to quickly and efficiently generate understandable unit tests.

We have created a prototype implementation for our approach, which we have coined *MicroTestCarver*. In this paper, we evaluate that prototype. Our investigation is steered by the following research questions:

RQ$_1$ Can *MicroTestCarver* generate unit tests based on information carved from E2E tests? (Feasibility)

RQ$_2$ How do the tests generated by our approach compare to EvoSuite-generated tests in terms of understandability?

RQ$_3$ How do the tests generated by our approach compare to manually written tests in terms of understandability?

We carry out an exploratory case study on 4 subject systems and a user study involving 20 participants to evaluate *MicroTestCarver*. Our initial findings are that *MicroTestCarver* is quite successful in generating unit tests, and in the comparison with EvoSuite-generated and manually written tests, it generates tests that are relatively easy to understand.

II. Background

A. Unit Test Generation

Automated test generation approaches have been developed in order to reduce testing costs. Today, tools such as Evo-

```java
@Test
public void shouldCallWeatherService() {
    var expectedResponse = new WeatherResponse("raining", "a light drizzle");
    given(restTemplate.getForObject("Weather API", WeatherResponse.class))
        .willReturn(expectedResponse);
    var actualResponse = subject.fetchWeather();
    assertThat(actualResponse, is(Optional.of(expectedResponse)));
}
----------------------------------------------------------
@Test(timeout = 4000)
public void testEqualsWithNull() throws Throwable {
    WeatherResponse weatherResponse0 = new WeatherResponse("S:q$ZHC!0J3", "&_>!@K");
    boolean boolean0 = weatherResponse0.equals((Object) null);
    assertFalse(boolean0);
    assertEquals("WeatherResponse{weather=[Weather{main='S:q$ZHC!0J3', description='&_>!@K'}]}", weatherResponse0.toString());
}
```

Listing 1: An example of (A) a manually written unit test and (B) a EvoSuite-generated unit test

Suite [9] and Randoop [11] generate a test suite starting from Java source code using a search-based or random approach to reach higher coverage [16], [17]. Several recent empirical studies focused on the challenges automated test generators face in real life, and the quality of the tests generated [14], [18]–[22]. Even though automated unit test generation has made significant progress, generated unit tests are less readable than their human-written counterparts [23]. Almasi et al. have conducted an extensive evaluation of automatically generated unit tests in the financial services domain, observing that developers (1) find it difficult to follow the scenario of the test case, (2) find the test data unclear, and (3) have difficulties with the meaningfulness of generated assertions [14].

B. *Readability and Understandability*

Readability and understandability are two similar terms, but have different meanings. *Readability* entails structural and semantic characteristics that allow developers to understand source code, while *understandability* is defined as the ease by which developers can extract information from a program [24].

Buse and Weimer [25] built a readability metric for source code. A predictive model was developed by Daka et al. [26] to assess the readability of unit tests, which was applied to EvoSuite to produce more readable tests by including readability as a secondary objective. However, understandability is more qualitative and difficult to capture in a model. In this paper, we use the term understandability to signify this difference.

C. *Capturing and Replaying*

The purpose of carving unit tests is to automatically extract a collection of unit tests replicating the calls seen during the higher-level test [27]. The process is sometimes called “record and replay”, as the key idea is to record calls, and replay them later – either collectively or selectively [28].

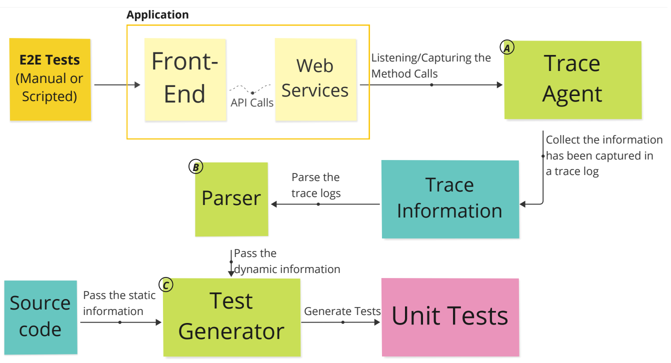
Application
E2E Tests (Manual or Scripted)
Front-End
API Calls
Web Services
Listening/Capturing the Method Calls
A
Trace Agent
Collect the information has been captured in a trace log
Trace Information
Parse the trace logs
B
Parser
Pass the dynamic information
Source code
Pass the static information
C
Test Generator
Generate Tests
Unit Tests

Fig. 1. Overview of the carving framework

III. The MicroTestCarver Framework

Figure 1 presents an overview of our approach. The *MicroTestCarver* framework takes an E2E test, which can either be a manual or scripted E2E test, and it generates unit tests in three phases: *instrumenting*, *parsing*, and *generating unit tests*. In the first step, it instruments the E2E tests and records information, such as calls and input data; in the second step, it parses this information, and finally, it uses a template-based approach to generate unit tests based on the parsed data.

Our approach “carves scenarios” from the E2E tests to reproduce (smaller elements of) them in the form of unit tests. Our hypothesis is that these higher-level test scenarios embedded in the E2E tests and containing concrete values, can lead to easier to understand unit test scenarios.

Our approach is implemented in a Java-based prototype called *MicroTestCarver (MTC)*. Our tool focuses on generating tests for public methods, which is similar to how a developer would produce unit tests for their production code.

Next, we describe each phase of our approach in detail.

A. *E2E tests instrumentation*

We use BTrace [29] as the basis for our instrumentation tool. We have created a fork of BTrace and developed some additional functionality for our carving approach. In particular, we now collect detailed information of types, and we also collect and serialize information on fields, arguments, and callbacks in a uniform manner. In addition, we chose to use XStream [30] in the modified Btrace because it is an advanced and robust serialization library for Java, and it can handle complex custom objects effectively. The modified version of BTrace is available in the replication package [31].

All recorded information is written to a trace log. Objects of non-primitive types are serialized into a serialized object pool. An example of a trace log is shown in Listing 2.

Basically, a trace log is a graph, in which we identify two types of methods: a *NodeMethod*, which is a method that calls other methods of interest in the test generation process, and a *LeafMethod*, which could still call other methods, but those methods are no longer of interest in the test generation process (and could for example be mocked). The concept of a trace log is illustrated in Figure 2, in which three NMs are highlighted that are called in the `ExampleController` class. Each NM

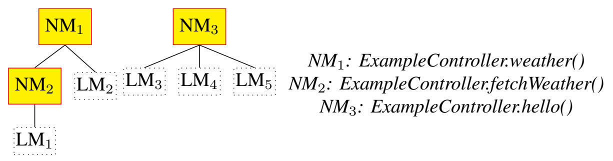
NM$_1$
NM$_3$
NM$_2$
LM$_2$
LM$_3$
LM$_4$
LM$_5$
LM$_1$
NM$_1$: ExampleController.weather()
NM$_2$: ExampleController.fetchWeather()
NM$_3$: ExampleController.hello()

Fig. 2. An example of trace log with their classes

and their LMs are exemplified in Listing 2. The leaves in a trace log are either a LM or a NM without a method call. We will now explain both concepts.

LeafMethod. A LeafMethod (LM) is a called method that does not have a callee that the trace script is watching. An LM refers to a method that is outside the package being watched; it may be a method in a third-party library. Each LM has a *name*, a *type*, a set of *arguments*, a state of the object (i.e., attribute values) before executing the method body, and if the return type is not void, a *return* value. In lines 11–19 of Listing 2, an LM with the name of `RestTemplate` is illustrated: its arguments are a string, class, and an array of objects. In lines 14–19, its callback is shown, which is a container object (Optional) of type `WeatherResponse`; it is serialized in a serialized object pool with the name `1e23ee0e`.

NodeMethod. A NodeMethod (NM) is a method that is called within the scope of tracing, and a test will be generated according to this method during the test generation phase. Each NM, in addition to all the properties of an LM, includes a set of methods called in it; these methods can be a LeafMethod or NodeMethod. On line 1 of Listing 2, `weather` is a NM with no arguments and several fields, such as `personRepository` and `weatherClient`. `fetchWeather()`, which is a NodeMethod, is also called in this NM; it finally returns a `String` on lines 22–24.

An important element to consider is what constitutes a *LeafMethod* and a *NodeMethod*. During the instrumentation phase, we focus on a particular packageName; we watch the classes inside the package and generate tests for them. Classes outside of the package, but that are called in methods residing inside the package, are labeled LeafMethods.

B. Parsing

As we aim to create building blocks for the test generation phase, we parse the trace log and deserialize the serialized objects with the aim of reconstructing the trace data into actual runtime objects. We do so by unifying all elements that were recorded, i.e., the trace log itself and the serialized objects. We create a set of classes that group together NodeMethods based on the classname in the fully qualified path. For example, in Fig. 2, `ExampleController` will be reinitiated based on the NMs whose class name is `ExampleController`. In order to create a class, its arguments, fields, and methods will be assigned based on its NMs, and its constructor method is a NodeMethod with `<init>` name.

C. Generating Unit Tests

We generate test cases based on the classes created in the Parsing phase and the analysis of the existing source code. We

```
1  public String ExampleController#weather:{
                    NodeMethod1
2    Args: []
3    Fields: [{
4      name: personRepository,
5      type: person.PersonRepository, ...
6    }, {
7      name: weatherClient,
8      type: weather.WeatherClient, ...
9    }, ]
10    public Optional  weather.WeatherClient#fetchWeather:{ ...
                           NodeMethod2
11      virtual Object RestTemplate#getForObject (String,
12        java.lang.Class, Object[])[737ff5c4]
13      Args: [...]
14      Callback: {
15        hash: 1e23ee0e,
16        type: java.util.Optional,                    LeafMethod1
17        serialized: true,
18        object: Optional[WeatherResponse{'Clear', 'clear sky'}],
19      }
       ...
20    }:
21    public String  weather.WeatherResponse#getSummary:{ ... :}
                          LeafMethod2
22    Return: {
23      type: String,
24      object: "Clear: clear sky", ...
25    }
26  }:
27   public String  ExampleController#hello(String):{
                      NodeMethod3
      ...
28    interface Optional
↪   person.PersonRepository#findByLastName(String)
29    public String person.Person#getFirstName:{...}        LeafMethod3-5
30    public String person.Person#getLastName:{...}
      ...
31  :}
```

Listing 2: A shortened example of a trace log

used *JUnit* as test framework and *Mockito* for mocking, and we utilized their annotations to improve legibility. In addition, we used Khorkiov's guidelines [32] for writing unit tests, in order to ensure the carved tests have a clear and readable structure. Listing 3 shows an example of a unit test generated by MTC based on the previous phases. We will first explain how we reproduce objects in a test, and then explain how different components of a test (*fields*, *set-up*, and *test method body*) will be produced.

1) Reproducing Objects: In order to accurately reproduce objects in various parts of a test, such as setting values, invoking methods, and making assertions, we need both dynamic and static analysis. To accomplish this, we combined *Spoon* [33] and Java reflection. As a result of analyzing the source code with Spoon, we are able to identify the appropriate constructor that can recreate the parsed object and set its fields. We have implemented three strategies in order to reproduce the parsed objects: *Unmarshalling*, *ToString*, and *Guessing*. The *Unmarshalling* strategy is used when the object is deserialized, and it will reveal how to recreate the runtime object. We have implemented unmarshallers for various types: primitive types, String, Collection, Map, Optional, Enum, Date, Locale, and BigDecimal. For other types not specifically handled by dedicated unmarshallers, we use the ReflectionUnmarshaller as a fallback, which replicates an object by setting its fields. This strategy reproduces `WeatherResponse` in line 15 and an `Optional` object in line 22. The *guessing* strategy is used when the object is not deserialized, and we are trying to reproduce it like the custom unmarshaller by setting its fields, with this difference that its fields come from the trace log, not a runtime object. The *toString* strategy is used for the assertions when the object is not deserialized, but it overrides the `toString()` method.

2) Fields: The fields of a test class include the declaration of the CUT (class under test) and setting up a mocked object if necessary. As we want to make it very clear what the CUT is, we name that field *subject*. The objects that are annotated as `@Mock` contain methods that are called in the test methods. This is illustrated in Listing 3 where the subject is the declaration of `ExampleController`, the CUT; `weatherClient` and `personRepository` are the objects that should be mocked since its methods (`fetchWeather` and `findByLastName`) are called (lines 14 and 22).

3) Set-up: Every test class has a set-up part containing a common initialization that is repeated in all methods, i.e., the test fixture [32]. The heuristic used for setting the default values of the fields is to select a value that is repeated most often in the NMs of a class. In doing so, a significant amount of duplication can be avoided. The state of the subject object will also be set here, for example, on line 10 the subject object is initiated and used in lines 16 and 23.

4) Test Method: In order to have a simple and uniform structure, we use the Arrange-Act-Assert (AAA) pattern [32]. Additionally, this pattern makes test cases easier to read and understand. We will discuss the elements of a test method in the following: method name, arrange, action, and assertion.

Test Name: When developers navigate among sets of unit tests, the names of the tests aid them in understanding the purpose and scenario of the tests. While there are complex approaches to naming the methods [34], [35], we used a simple heuristic approach for creating unique tests cases based on inputs and output of a NodeMethod. If there is only one NodeMethod for the MUT (method under test), the test name will be [MUT]Test, like `weatherTest` (line 13). If there are multiple NodeMethods, the test name will be a combination of the types and values of the inputs and output like `helloWhereCarterTest` (line 20). This name is unique since if all conditions were the same, a duplicate test would be recognized. The test name pattern is:

$$([MUT][Where[Inputs]^{*}][Returning[Output]]?Test)$$

Arrange: The arrange section involves bringing the subject and its dependencies into the desired state as well as mocking any other methods in the MUT that need to be called.

By first determining which objects to mock, which is done in the fields section, we can mock the methods based on the NM's callees (LeafMethods) and their callbacks. As shown in line 14 of Listing 3, the behavior of `weatherClient.fetchWeather`, which retrieves the weather from the weather service, is mocked.

```
1 public class ExampleControllerTest {
2   private ExampleController subject;
3    @Mock
4    private WeatherClient weatherClient;
5    @Mock                                              Parameters
6    private PersonRepository personRepository;

7    @BeforeEach
8    public void setUp() throws Exception {
9      MockitoAnnotations.openMocks(this);
10     subject = new ExampleController(personRepository,   Set-Up
↪  weatherClient);
11   }

12   @Test
13   public void weatherTest() throws Exception{
14     given(weatherClient.fetchWeather()).willReturn(
15       Optional.of(new WeatherResponse("Clear", "clear sky")));
                                                     Test Method
16     String weather = subject.weather();

17     assertThat(weather, is("Clear: clear sky"));
18   }

19   @Test
20   public void helloWhereCarterTest() throws Exception{
21     Person carter = new Person("james", "carter");

22     given(personRepository.findByLastName("carter"))
           .willReturn(Optional.of(carter));

23     String hello = subject.hello("carter");

24     assertThat(hello, is("Hello james carter!"));
25   }
26 }
```

Listing 3: An example of a MTC-generated unit test

Additionally, if the value of the fields in the NodeMethod is different from the one set in the set-up, it will be reset in this section.

Act: This section contains the method called on the CUT, the input values passed to it, and output values captured. The return type of the NM is assigned to the output type, e.g., `weather()` (line 16) and `hello()` (line 23).

Assert: This section contains the verification of the return value, or the final state of the subject with the expected results. We use the *assertThat* assertion, which compares the output of the MUT and the expected result captured in the NM, e.g, lines 17 and 24 in Listing 3.

IV. Study Setup and Design

To evaluate the effectiveness of *MicroTestCarver* in improving the generation of understandable test cases in real-world applications, this section describes the methodology of evaluation. We investigate the following research questions.

RQ$_1$ Can *MicroTestCarver* generate unit tests based on information carved from E2E tests? (Feasibility)

RQ$_2$ How do the tests generated by our approach compare to EvoSuite-generated tests in terms of understandability?

RQ$_3$ How do the tests generated by our approach compare to manually written tests in terms of understandability?

TABLE I
PROJECTS USED IN THE EVALUATION

Application	Version	#Test Files	#Stars	#Forks	#Commits	Scale
Alfio	2.0.5	189	1.5K	2.5K	3.6K	Large
Lab-Insurance	1.0.0	24	0.5K	0.2K	384	Large
Petclinic	2.7.3	23	6.5k	20K	0.8K	Mid
Spring-Testing	0.0.1	13	0.9K	0.4K	130	Small

A. *Study Setup*

We have carried out the evaluation on four popular open-source Java web applications. We have used the following criteria for our selection of projects:

- The project is an open-source Java web application.
- The project is popular, active, and mature as measured in terms of forks, stars, and commits.
- The project has a test suite with tests for different layers of the test pyramid, especially unit tests.

Using these criteria, we found nearly 200 projects on GitHub. Next, we manually search among these projects to select four projects from different domains with varying sizes. Table I shows the selected projects: *Alfio*[1], *LAB-Insurance*[2], *Spring-Testing*[3], and *PetClinic*[4]. We manually conducted end-to-end tests for these projects, which covered their core functionalities; the E2E tests are recorded and included in the replication package [31]. Manually written unit tests are available for all of these applications, which makes it possible to compare them with carved tests for RQ3.

B. *Study Design*

The first research question investigates the feasibility and analyzes the *MicroTestCarver* approach. For the second and third research questions, we designed and conducted a survey involving 20 participants. The participants compared understandability of the *MicroTestCarver* tests with tests that are automatically generated by EvoSuite, and to tests that were manually written and are part of the open source projects.

Although we had 20 respondents in total, we did collect 30 responses for each question. This is due to us assigning two test methods to 10 of the participants, while the other 10 respondents got a shorter survey. This enabled us to get a wider range of responses.

1) Participants: We have conducted a survey among BSc, MSc and PhD students. In order to recruit participants, we advertised our survey on our university's internal Mattermost[5] chat service, where computer science students can be found. Our participants include 10 BSc, 3 MSc and 7 PhD students.

2) Survey: The 20 participants were asked to read and evaluate a set of test cases in six questions, which are depicted in Table II. In the first round (questions 1–3), participants selected more understandable test cases among MTC, EvoSuite, and manually written tests. By using text highlighting and open-ended questions combined in questions 1 and 2, we can gain more detailed and accurate insights into why participants think the test is more understandable [36]. Text highlighting allows participants to quickly and accurately refer to specific parts of the test that they believe made it more understandable, while open-ended questions provide a more in-depth understanding of their opinion. In the second round (questions 4–6) participants rated the understandability of the test cases (MTC, EvoSuite, and manually written) based on the understandability criteria, shown in Table III (similar to related work [35], [37]).

[1]https://github.com/alfio-event/alf.io
[2]https://github.com/asc-lab/micronaut-microservices-poc
[3]https://github.com/hamvocke/spring-testing
[4]https://github.com/spring-projects/spring-petclinic
[5]https://mattermost.com, last visited June 6th, 2023.

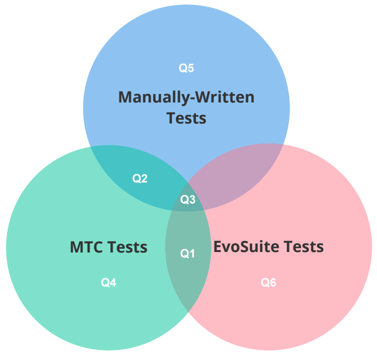

Fig. 3. Coverage of questions with regard to test sources.

In order to provide a systematic and unbiased survey we took three considerations: 1) Test cases were displayed without identifying which test generator was used to create them or whether they were manually created, 2) for each question, we had a pool of test methods that we wanted to evaluate (see column 4 of Table II); we let the online survey (Qualtrics) randomly select, 3) for the first three questions in which we want to compare tests from different sources (MTC, manual, and EvoSuite), we made sure to compare tests that validate the same functionality, and we randomized the order of the presented fragments of test code. Figure 3 illustrates how the question rounds cover the different types of tests.

In Q1, we compared 21 pairs of MTC and EvoSuite tests, where pair means that both tests evaluate the same functionality. In the second question, we compared MTC with manually written tests through 12 pairs. The third question ranks MTC, EvoSuite, and manually written tests in terms of

TABLE II
QUESTIONNAIRE OVERVIEW

#	Title	Type of Question	#Tests	RQs
Q1	Comparison with EvoSuite tests	Multiple Choice, Highlight, Open	21	RQ2
Q2	Comparison with manually written tests	Multiple Choice, Highlight, Open	10	RQ3
Q3	Ranking Tests (MTC, EvoSuite, manually written tests)	Ranking	3	RQ2, RQ3
Q4	Evaluate MTC test based on the criteria	Likert, Open	10	RQ2, RQ3
Q5	Evaluate EvoSuite test based on the criteria	Likert, Open	12	RQ2
Q6	Evaluate manually written test based on the criteria	Likert, Open	12	RQ3

TABLE III
UNDERSTANDABILITY CRITERIA

Criterion	Sub-Criterion	Description
Semantic	Descriptive naming	The names of variables and methods that describe their functions.
	Descriptive test data	Test data clearly (input, output, mock data) illustrate the test scenario.
Naturalness	Meaningful in the context	The test scenario is meaningful in the domain of the system.
	Intent (easy to understand)	The test behavior is easy to understand.

their understandability among 3 test pairs. In questions 4 to 6, participants will rate the understandability of MTC, EvoSuite, and manually written test cases in isolation according to our proposed understandability criteria on a scale of 1 to 5. In each of these questions, we randomly selected 12 unit tests that are not included in questions 1–3. We can gain insight into how different parts of a test are perceived in terms of understandability, as well as the understandability of tests not common to each group of tests, since questions 1–3 only address tests that cover the same functionality.

V. RESULT

Our focus in this investigation is on the understandability of the generated test cases, and not so much on their effectiveness (e.g., in reaching high code coverage). In the following we discuss the results of our research questions.

A. *RQ1: Feasibility of the unit test generation based on E2E Tests*

Table IV presents the results of the carved unit tests in terms of execution results. In total, 69 tests are carved for 35 CUTs of the four study subjects. Of the 69 carved tests, 59 are executable (85%), 61 have an executable body (88%), and of the 35 test classes, 31 have executable test fixtures (88%). In addition, 96% of the tests passed.

Next, we investigate the reasons for generating both non-executing and failing tests:

1) Execution Failure Analysis: We have identified four causes for the execution failure of six tests. We annotated them R1 to R4 and analyze them.

R1: *Passing a class as an argument.* In the tests of `WeatherClientTest` (row 2) and `AlfioMetadata` (row 21) a class is passed statically as an argument to invoke a method (e.g., for mocking, or initializing a CUT). However, as BTrace only has access to runtime objects during carving, we cannot determine which object is being passed statically.

R2: *Type conversion error.* `PetTest` (row 8) fails to execute because of a failing type conversion. More specifically, in this case, Hibernate acts as a proxy and changes the object type at runtime, which leads to type inconsistency at runtime.

R3: *Unable to reproduce an object.* An object may fail to reproduce due to failures during the instrumentation or generation phases. For instance, in row 21, 30, and 33, XStream failed to serialize an object, and MTC failed to reproduce an object using alternative unmarshalling strategies such as ToString and a guessing approach. In row 14, the failure to reproduce an object is attributed to the absence of a suitable unmarshaller for the given type. Furthermore, in row 22, BTrace was unable to instrument an argument, leading to the inability to construct an object.

R4: *Private method.* If the method is private, it is not possible to instantiate from this class, and technically, it is out of the project scope to generate tests for private methods. `systemLevelTest` (row 23) has a private method, and this method cannot be invoked in the class.

2) Test Failure Analysis: We use the Hamcrest matcher (`assertThat` and `is`) for assertions, which internally invokes the `equals` method to compare two objects. In order for the test to pass, the `equals` method needs to be overridden for the CUT, otherwise it relies on the memory address comparison implementation of `equals` in the `Object` class. `petTest` (row 8) and `CalculatePriceHandlerTest` (row 33) failed because `equals` was not implemented for their classes, but when we we implemented it, the tests passed.

Even though *MicroTestCarver* is not designed to optimize test coverage (i.e., it is not search-based), Figure 4 compares instruction coverage for each application using *MicroTest-Carver*, existing manually written tests, EvoSuite-generated tests, and their combinations for each application. Figure 4 demonstrates the effectiveness of MTC in enhancing coverage by including scenarios not covered by EvoSuite-generated or manually written tests. It is important to note that although search-based approaches excel at generating tests for corner cases and improving coverage, they may fail to mock certain methods due to a lack of runtime information access. Notably, EvoSuite-generated tests combined with MTC-generated tests provided higher coverage in the projects.

As Alfio is quite a large project, we exclude certain classes, such as configuration classes, to measure coverage with a better representation of the core functionality. In the Lab-Insurance, coverage evaluation was limited to pricing and product microservices.

Highlight of RQ1: Feasibility
Our results indicate that 85% of the unit tests that *MicroTestCarver* generates from carved information from E2E tests are executable. We further provide reasons for the non-compilation or failure of *MicroTestCarver* -generated tests.

B. *RQ2: Understandability of carved tests vs EvoSuite tests*

Figure 5 illustrates the understandability of MTC and Evo-Suite generated tests based on comparisons, rankings, and

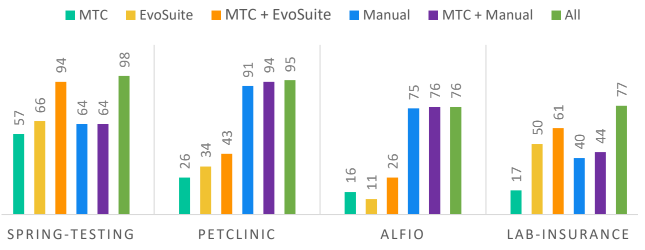

Fig. 4. Instruction coverage for each study subject

TABLE IV
EXPERIMENTAL RESULTS OF THE GENERATED UNIT TESTS ON THE STUDY SUBJECTS USING MICROTESTCARVER

#	Test Class	Executable Fixture	Executable Tests	Failures	Pass/Fail	Executable
1	ExampleControllerTest	True	4/4		4/4	4/4
2	WeatherClientTest	False	0/1	R1	-	0/1
3	PersonTest	True	2/2		2/2	2/2
4	WeatherResponseTest	True	1/1		1/1	1/1
	Spring-Testing: 8 Tests	3/4 (75%)	7/8 (87%)	1	7/7 (100%)	7/8 (87%)
5	NamedEntityTest	True	2/2		2/2	2/2
6	BaseEntityTest	True	1/1		1/1	1/1
7	PersonTest	True	2/2		2/2	2/2
8	PetTest	True	2/3	R2	1/2	2/3
9	PetTypeFormatterTest	True	3/3		3/3	3/3
10	PetTypeTest	True	2/2		2/2	2/2
11	SpecialtyTest	True	2/2		2/2	2/2
12	VetsTest	True	1/1		1/1	1/1
13	OwnerTest	True	5/5		5/5	5/5
14	OwnerControllerTest	True	1/2	R3	1/1	1/2
15	ClinicServiceImplTest	True	1/1		1/1	1/1
	PetClinic: 24 Tests	11/11 (100%)	22/24 (91%)	2	21/22 (95%)	22/24 (91%)
16	AuthorityTest	True	1/1		1/1	1/1
17	ConfigurationKeyValuePathLevelTest	True	2/2		2/2	2/2
18	ProviderAndKeysTest	True	1/1		1/1	1/1
19	EventDescriptionTest	True	2/2		2/2	2/2
20	LanguageTest	True	2/2		2/2	2/2
21	AlfioMetadataTest	False	0/1	R1, R3	-	0/1
22	ConfirmationEmailConfigurationTest	False	1/1	R3	-	0/1
23	SystemLevelTest	False	1/1	R4	-	0/1
24	LocaleDescription	True	2/2		2/2	2/2
25	OrganizationContactTest	True	2/2		2/2	2/2
26	TicketReservationStatus	True	1/1		1/1	1/1
	Alfio: 16 Tests	8/11 (72%)	15/16 (93%)	3	13/13 (100%)	13/16 (81%)
27	ChoiceQuestionDTOTest	True	1/1		1/1	1/1
28	NumericQuestionDTOTest	True	1/1		1/1	1/1
29	ChoiceDTOTest	True	2/2		2/2	2/2
30	ProductsControllerTest	True	0/2	R3	-	0/2
31	CalculatePriceCommandTest	True	5/5		5/5	5/5
32	CalculatePriceResultTest	True	2/2		2/2	2/2
33	CalculatePriceHandlerTest	True	1/3	R3	0/1	1/3
34	CalculationTest	True	3/3		3/3	3/3
35	CoverTest	True	2/2		2/2	2/2
	LAB-Insurance: 21 Tests	9/9 (100%)	17/21 (80%)	4	17/17 (100%)	17/21 (80%)
	Total: 69 Tests, 35 Test classes	31/35 (88.5%)	61/69 (88.4%)	10	56/58 (96.5%)	59/69 (85%)

criteria-based questions. The results indicate that MTC tests are generally easier to understand than EvoSuite tests, as MTC received a higher score across all questions.

When participants were asked to select which test case is more understandable, out of the 30 responses, the majority (70%) expressed a preference for MTC over EvoSuite, while the remaining 30% selected EvoSuite. In the ranking question, 24 out of 30 responses indicated MTC tests to be more understandable than EvoSuite tests.

During the highlight and open-ended questions, participants emphasized that MTC tests provided clearer test data and logic compared to EvoSuite tests (8 mentions). Additionally, five participants found the separate test setup in MTC tests to make the overall testing process simpler to understand. However, two respondents highlighted a preference for EvoSuite tests because objects were instantiated within the same test method.

Regarding identifiers, participants indicated a preference for clear, sensible, and concise names rather than long and vague ones. Furthermore, participants noted that the construction of objects influenced their understanding of identifiers. In this regard, five respondents favored identifiers in MTC tests, while three preferred identifiers in EvoSuite tests.

In terms of assertions, participants expressed a preference for specific assertions that go beyond simple functionality testing. Three participants specifically mentioned a preference for assertions such as `assertNull` and `assertEquals` over `assertThat`. Three other participants found the assertions in EvoSuite to be insufficient for adequately testing the intended functionality. Participants also emphasized the importance of legibility in tests and favored the use of AssertJ annotations (for mocking and assertions) over pure JUnit. They suggested that providing enough space for different parts of the tests enhances their understandability. One participant noted that MTC demonstrated a good separation between different parts of the test, while another mentioned that the separation was not satisfactory in a question where the test setup was longer.

Furthermore, some participants stated that shorter lines contribute to simpler tests. Two participants preferred EvoSuite tests due to this factor, while one participant selected the MTC test for the same reason. Listing 4 presents examples of tests generated by MTC and EvoSuite in order to give the reader a sense of how the tests are generated by different tools.

The criteria-based questions showed that MTC had a significantly higher average score than EvoSuite (4.23 vs. 2.85 — see Figure 5). Figure 6 illustrates the distribution of participants' opinions regarding each criterion for both MTC and EvoSuite tests. Analyzing the results of criteria 1 and 2 for MTC and EvoSuite, it becomes evident that MTC tests' semantics are clearer and more understandable for developers. Additionally, criteria 3 and 4 suggest that MTC tests feel more natural to developers. In response to the open-ended question, participants expressed concerns about non-descriptive identifiers and test data. One participant remarked, *"The names of the variables are not at all descriptive, and the meaning of them is hard to figure out. Also, the data are hard to understand."* They also criticized the assertions as obvious and shallow. Conversely, participants found the test data and identifiers in MTC tests to be descriptive, albeit with some lines of code in the *arrangement* being redundant and unnecessary.

When analyzing the survey results and conducting manual analysis, we discovered the crucial role of test data in comprehending a test case. Specifically, when random, null, empty, or mocked inputs are employed in EvoSuite, it becomes more challenging to grasp the underlying logic and purpose of a test. However, it should be noted that EvoSuite-generated tests tend to be shorter due to the utilization of test case minimization as a secondary search objective.

Highlight of RQ2
When we compare carved tests with EvoSuite-generated tests, we observe that participants value the use of actual test data which is derived from E2E in carved tests makes the test easier to understand and more meaningful. Search-based approaches are good at generating short test cases.

C. RQ3: Understandability of the carved tests vs manual tests

Figure 7 presents a comparative analysis of the understandability of MTC and manually written tests through comparisons, rankings, and criteria-based questions. The results indicate a relatively similar level of understandability between MTC tests and manually written tests.

The participants were asked to select the most understandable test case, with 52% choosing MTC tests and 48% opting for manually written tests. MTC tests perform slightly better in the ranking question, with 18 responses ranking them as more understandable against 12 responses for EvoSuite tests.

When participants provided feedback through highlighting and open-ended questions, their opinions regarding MTC-generated and manually written test cases were mostly similar. However, they did indicate that comprehending the test data

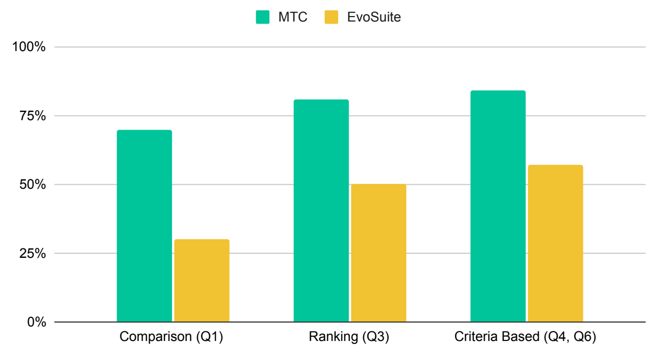

Fig. 5. Understandability of MicroTestCarver versus EvoSuite tests

```
@Test
public void testCreatesPetType()  throws Throwable {
    PetType petType0 = new PetType();
    assertNull(petType0.getName());
}
------------------------------------------------------
public void setUp() throws Exception {
    subject = new PetType();
    subject.setName("cat");
    subject.setId(1);
}
@Test
public void getTypeTest() throws Exception {
    PetType getType = subject.getType();

    PetType petType = new PetType();
    petType.setId(1);
    petType.setName("cat");

    assertThat(getType, is(petType));
}
```

Listing 4: First test is a test generated by EvoSuite, and second one is generated by *MicroTestCarver* .

and the mocking process in MTC tests was relatively easier in comparison to manually written tests. Specifically, participants mentioned the superiority of MTC test data seven times, while favoring manually written test data in three instances. Additionally, in one case, they indicated a preference for strong typing over the use of "var" in manually written tests.

Regarding identifiers, they observed that the identifiers (variable names and test method names) in MTC tests were shorter, while those in manually written tests were more descriptive and clear.

Participants had mixed views on the length and structure of the tests. In five instances, they mentioned that MTC tests were shorter, while in the other five instances, they mentioned that manually written tests were shorter. Notably, in two pairs, we observed that manually written tests are overly long and try to test too much (so-called *eager tests* [38]). Furthermore, participants mentioned that the manually written tests had less code duplication due to the use of best practices such as the factory method [39] and parametrized test cases. An example of a comparison illustrating the greater understandability of a manually written test than a carved one is depicted in Listing 5. Although both tests employ meaningful test data and exhibit understandable logic, the manually written test has better structure and less duplication. For mocking the `petType` list the manual test utilizes the `makePetTypes()` factory method, enhancing readability and reusability.

Moreover, the findings from the criteria-based questions indicate that MTC exhibited a slightly higher average score (4.23) compared to manually written tests (3.95), as can be seen in Figure 5. Figure 6 visually represents the participants' opinions on test cases generated by MTC and manually written tests. Although the results for criteria 1 and 2 were similar between MTC and manual testing, the participants found the semantics of MTC tests to be more comprehensible. Additionally, both manual testing and MTC were perceived by respondents as being almost equally natural.

In the open-ended question, participants mainly emphasized that the intent behind manually written tests was easily

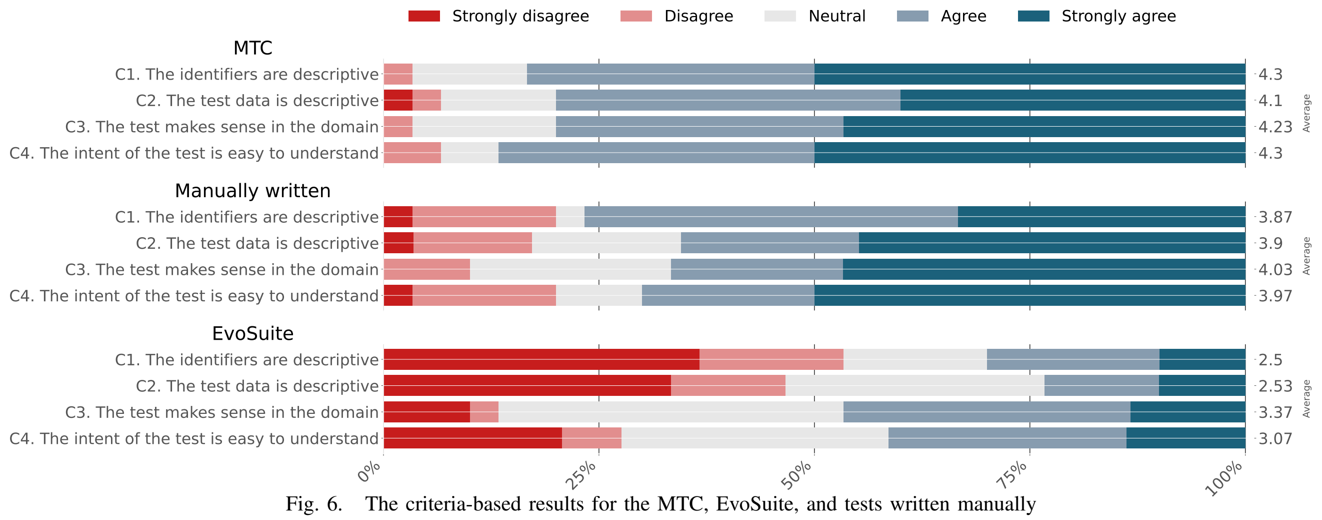

Fig. 6. The criteria-based results for the MTC, EvoSuite, and tests written manually

understandable, a consequence of comprehensible test data and logic. However, their perception of the descriptiveness of identifiers and test data was mixed. While participants expressed positive opinions about the identifiers on numerous occasions (ten times), they also indicated that the identifiers of the test cases were sometimes unclear (six times). The understandability of manually written tests varies depending on the project, e.g., in Spring-Testing and Petclinic, manually written tests achieved higher scores in the criteria-based questions than in the Alfio project. In contrast, MTC employs a template-based approach, and the rating is almost consistent. Participants often raised concerns about lengthy lines in the setup methods in MTC tests, which are addressed in manually written tests through best practices such as parametrized tests.

Highlight of RQ3
Comparing participants' answers we get indications that using data derived from E2E tests brings MTC tests closer to manually written tests in terms of understandability. We also observe that some manually written tests use some best practices, e.g., parameterized tests, which enhances sometimes the understandability of manually written tests.

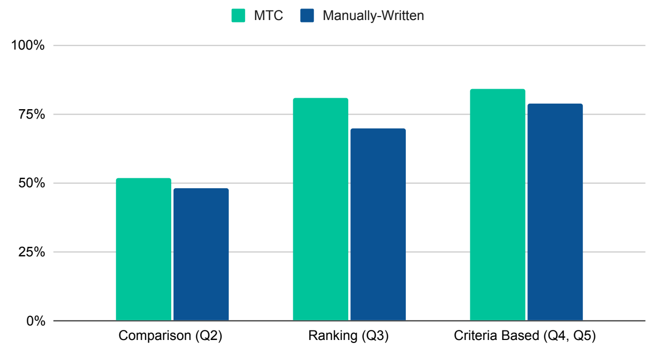

Fig. 7. Understandability of MicroTestCarver versus manually written tests

```
@Test
void shouldParse() throws ParseException {
  given(this.pets.findPetTypes()).willReturn(makePetTypes(⌋
  ↪  ));
  PetType petType = petTypeFormatter.parse("Bird",
  ↪  Locale.ENGLISH);
  assertThat(petType.getName()).isEqualTo("Bird");
}
private List<PetType> makePetTypes() {
  List<PetType> petTypes = new ArrayList<>();
  petTypes.add(new PetType() {
    { setName("Dog"); }
  });
  petTypes.add(new PetType() {
    { setName("Bird"); }
  });
  return petTypes;
}
-----------------------------------------------------------
public void parseWhereBirdTest() throws Exception {
  PetType PetType = new PetType();
  PetType.setId(5);
  PetType.setName("bird");
  ArrayList<PetType> petTypes = new ArrayList<>();
  petTypes.add(PetType);

  given(owners.findPetTypes()).willReturn(petTypes);

  PetType parse = subject.parse("bird", Locale.ENGLISH);

  PetType PetType_1 = new PetType();
  PetType_1.setId(5);
  PetType_1.setName("bird");

  assertThat(parse, is(PetType));
}
```

Listing 5: Comparison of a manual (A) and carved test (B)

D. *Threats to validity*

Threats to construct validity pertain to how we make our observations. To mitigate this potential issue, we opted for conducting our survey in person, ensuring that participants accurately completed the questionnaire and did not select questions at random. Additionally, prior to starting the survey, we provided a comprehensive explanation of the study scenario to ensure participants' attentiveness to our study's context. Moreover, in Section IV-B, we discussed our considerations to minimize the potential bias in the participants' behaviors.

Overall, 84% of the open-ended questions were completed.

This high response rate enhances the reliability of our data. However, it is important to acknowledge that our study exclusively focused on students, which limits the generalizability of our findings. To address this limitation in future work, we intend to expand our evaluation by involving practitioners to obtain a more diverse audience perspective.

An important threat to validity is that while we examined test cases from four different subject systems that vary in size and domain, we only examined 69 generated test cases for RQ1, and the pool of 68 test cases for RQ2 and RQ3 is also limited. In future work, we will extend our investigation to more subject systems and more test cases.

To ensure the reliability of our conclusions, we employed a two-phase approach that lets respondents both compare and use criteria to judge test cases. However, it is worth noting that in the criteria-based phase, we randomly selected test cases for MTC, EvoSuite, and manually written tests. Consequently, there exists a possibility that a test case chosen represents a complex scenario, is more effective, and is less understandable (especially in manually-written tests); on the other hand, it might represent a simpler scenario but with higher understandability. This discrepancy could threaten our conclusion validity. In future studies, it would be beneficial to explore more sophisticated selection strategies to account for the variations in test case scenarios.

VI. Related Work

In this section we briefly compare how our approach compares to relevant other initiatives for test understandability.

A. Improving the understandability of automated testing

A number of studies have tried to improve the readability of generated unit tests, focusing on the following aspects:

1) Naming and summarization: Zhang et al. proposed an NLP-based technique that automatically generates descriptive names for unit tests based on the common structure and names of tests [40]. Daka et al. used coverage criteria to generate unique names for automatically generated unit tests [34]; Nijkamp et al. adapted this approach to fit test amplification [41]. Roy et al. developed DeepTC-Enhancer, which uses deep learning to automatically generate method-level summaries and rename identifiers for the generated test cases [35]. Panichella et al. proposed TestDescriber, which generates test case summaries that describe the intent of a generated unit test [42]. Panichella et al. have established that developers working with the test case descriptions are quicker in resolving bugs indicated by failing tests.

While these works focus on enhancing identifier names and documentation, *MicroTestCarver* aims to generate tests that are closer to manually written tests in terms of scenario.

2) Realistic Inputs: Afshan et al. have combined a natural language model with a search-based test generation to improve the readability of generated inputs [43]. Through a user study they have observed that participants are faster at evaluating inputs generated with their language model. Knowledge bases have been used in some studies to generate realistic inputs; Alonso et al. [44] utilized this approach to generate realistic web APIs, and Wanwarang et al. [45] have used it to test mobile applications. It is important to note that these aforementioned works only provide linguistically realistic data. On the other hand, MicroTestCarver can be used to generate actual test data in a variety of dimensions; it can generate test data.

B. Capturing/Replaying and Test Carving

Elbaum et al. [27], [46] proposed an approach to carving and replaying differential unit tests (DUTs) from system tests, as well as strategies to filter and prune test cases. DUTs are a hybrid of unit and system tests that keep the system state. Tiwari et al. [47] designed a tool to monitor the production workload to generate DUTs. Kampmann et al. [48] used a carving approach to extract parameterized unit tests from system test executions. Thummalapenta et al. [49] mine dynamic traces to generate .NET parameterized unit tests (PUTs) for the purpose of regression testing. Derakhshanfar et al. [50] reproduce a crash based using a search-based algorithm. *MicroTestCarver* and these tools use dynamic information, but their purposes are different; in comparison, *MicroTestCarver*'s primary objective is to enhance the understandability of its generated unit tests.

VII. Conclusion

In this paper we present the *MicroTestCarver* approach and associated tool. *MicroTestCarver* is a test generation tool that tries to generate understandable unit tests by carving information from E2E tests. The premise is that the information that we carve from the E2E test enables to create a sensible unit test scenario that contains actual test data, as opposed to synthetic and non-realistic test data.

We have carried out an exploratory case study on 4 software systems and a user study involving 20 participants; we have made the following observations. Firstly, we were able to generate 69 unit tests from carved data for 35 CUTs; 85% of the generated tests are executable, and of those 96% are passing tests (RQ1). Secondly, we found *MicroTestCarver*-generated tests to be more meaningful and easier to understand when we compare them to EvoSuite-generated tests (RQ2). Thirdly, when we compare *MicroTestCarver*-generated tests with manually written tests we observe that *MicroTestCarver*-generated tests are quite close to manually written tests in terms of understandability (RQ3).

In future work, we intend to combine our approach with search-based algorithms to generate unit tests for corner cases and still have meaningful test data and understandable test cases. We aim to leverage NLP and Large Language Models (LLMs) to combine source code and runtime execution information, thereby improving the understandability of identifiers and documentation. We also intend to extend our evaluation by setting up a controlled experiment with practitioners.

Acknowledgments

This research was partially funded by the Dutch science foundation NWO through the Vici "TestShift" grant (No. VI.C.182.032), and the EU Horizon 2020 H2020-ICT-2020-1-RIA "COSMOS" project (No.957254).

References

[1] A. J. Ko, B. Dosono, and N. Duriseti, "Thirty years of software problems in the news," in *Proc. Int'l Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 2014, pp. 32–39.

[2] K. L. Beck, *Test-Driven Development - By Example*, ser. The Addison-Wesley signature series. Addison-Wesley, 2003.

[3] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the IDE: patterns, beliefs, and behavior," *IEEE Trans. Software Eng.*, vol. 45, no. 3, pp. 261–284, 2019.

[4] M. F. Aniche, C. Treude, and A. Zaidman, "How developers engineer test cases: An observational study," *IEEE Trans. Software Eng.*, vol. 48, no. 12, pp. 4925–4946, 2022.

[5] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their ides," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 179–190.

[6] M. Beller, G. Gousios, and A. Zaidman, "How (much) do developers test?" in *37th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2015, pp. 559–562.

[7] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Trans. Software Eng.*, vol. 36, no. 6, pp. 742–762, 2010.

[8] L. Baresi and M. Miraz, "Testful: automatic unit-test generation for java classes," in *32nd IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 281–284.

[9] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proc. Joint Meeting Symp. Foundations of Software Engineering and the European Softw. Eng. Conf. (ESEC/FSE)*. ACM, 2011, pp. 416–419.

[10] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? A controlled empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 4, pp. 23:1–23:49, 2015.

[11] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Conf. on Object-Oriented Programming Systems and Applications (OOPSLA-Companion)*. ACM, 2007, pp. 815–816.

[12] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, no. 3, pp. 783–812, 2015.

[13] A. Arcuri, "An experience report on applying software testing academic results in industry: we need usable automated test generation," *Empirical Software Engineering*, vol. 23, no. 4, pp. 1959–1981, 2018.

[14] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *Int'l Conf. on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 263–272.

[15] C. E. Brandt and A. Zaidman, "Developer-centric test amplification," *Empir. Softw. Eng.*, vol. 27, no. 4, p. 96, 2022.

[16] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[17] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Trans. Software Eng.*, vol. 44, pp. 122–158, 2018.

[18] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: What if test code quality matters?" in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2016, pp. 130–141.

[19] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "On the diffusion of test smells in automatically generated test code: An empirical study," in *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*, 2016, pp. 5–14.

[20] G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H. C. Gall, "Scented since the beginning: On the diffuseness of test smells in automatically generated test code," *Journal of Systems and Software*, vol. 156, pp. 312–327, 2019.

[21] G. Fraser and A. Arcuri, "EvoSuite: On the challenges of test case generation in the real world," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2013, pp. 362–369.

[22] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201—-211.

[23] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, "An empirical investigation on the readability of manual and generated test cases," in *International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 348–351.

[24] D. Oliveira, R. Bruno, F. Madeiral, H. Masuhara, and F. Castor, "A systematic literature review on the impact of formatting elements on program understandability," 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2208.12141

[25] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Trans. on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.

[26] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 2015, pp. 107–118.

[27] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *Proc. Int'l Symposium on Foundations of Software Engineering (FSE)*. ACM, 2006, pp. 253–264.

[28] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2021.

[29] "Btrace - a safe, dynamic tracing tool for the java platform," December 2022. [Online]. Available: https://github.com/btraceio/btrace

[30] "Serialize java objects to xml and back again." December. [Online]. Available: http://x-stream.github.io

[31] A. Deljouyi and A. Zaidman, "generating unit tests based on carving E2E tests," Aug. 2023. [Online]. Available: https://github.com/amirdeljouyi/SCAM-2023-microtestcarver-replication

[32] V. Khorikov, *Unit Testing Principles, Practices, and Patterns*. Manning, 2019.

[33] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155–1179, 2016.

[34] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2017, pp. 57–67.

[35] D. Roy, Z. Zhang, M. Ma, V. Arnaoudova, A. Panichella, S. Panichella, D. Gonzalez, and M. Mirakhorli, "Deeptc-enhancer: Improving the readability of automatically generated tests," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*, 2020, pp. 287–298.

[36] R. G. J. S. R. Ares, G., "Text highlighting combined with open-ended questions: a methodological extension," *Journal of Sensory Studies*, vol. 38, 2023.

[37] D. Winkler, P. Urbanke, and R. Ramler, "What do we know about readability of test code? - a systematic mapping study," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 1167–1174.

[38] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[39] "Factory method pattern," 2023. [Online]. Available: https://refactoring.guru/design-patterns/factory-method

[40] B. Zhang, E. Hill, and J. Clause, "Towards automatically generating descriptive names for unit tests," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2016, pp. 625–636.

[41] N. Nijkamp, C. Brandt, and A. Zaidman, "Naming amplified tests based on improved coverage," in *Proc. Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM)*, 2021, pp. 237–241.

[42] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," in *Proc. Int'l Conference on Software Engineering (ICSE)*, 2016, pp. 547–558.

[43] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2013, pp. 352–361.

[44] J. C. Alonso, A. Martin-Lopez, S. Segura, J. M. Garcia, and A. Ruiz-Cortes, "Arte: Automated generation of realistic test inputs for web apis," *IEEE Transactions on Software Engineering*, pp. 1–1, 2022.

[45] T. Wanwarang, N. P. Borges, L. Bettscheider, and A. Zeller, "Testing apps with real-world inputs," in *Proceedings of the International Conference on Automation of Software Test (AST)*. ACM, 2020, pp. 1–10.

[46] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, “Carving and replaying differential unit test cases from system test cases,” *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 29–45, 2009.
[47] D. Tiwari, L. Zhang, M. Monperrus, and B. Baudry, “Production monitoring to improve test suites,” *IEEE Trans. on Reliability*, 2021.
[48] A. Kampmann and A. Zeller, “Carving parameterized unit tests,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, pp. 248–249.
[49] S. Thummalapenta, J. de Halleux, N. Tillmann, and S. Wadsworth, “Dygen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces,” in *Tests and Proofs*. Springer, 2010, pp. 77–93.
[50] P. Derakhshanfar, X. Devroey, G. Perrouin, A. Zaidman, and A. van Deursen, “Search-based crash reproduction using behavioural model seeding,” *Softw. Test. Verification Reliab.*, vol. 30, no. 3, 2020.