A Multi-Language Tool for Generating Unit Tests from Execution Traces

Gabriel Darbord, Nicolas Anquetil, Benoit Verhaeghe, Anne Etien

A Multi-Language Tool for Generating Unit Tests from Execution Traces

Gabriel Darbord
Univ. Lille, Inria, CNRS, Centrale Lille
UMR 9189 CRIStAL
F-59000 Lille, France
0000-0001-7364-7567

Nicolas Anquetil
Univ. Lille, CNRS, Inria, Centrale Lille
UMR 9189 CRIStAL
F-59000 Lille, France
0000-0003-1486-8399

Benoit Verhaeghe
Berger-Levrault
Limonest, France
0000-0002-4588-2698

Anne Etien
Univ. Lille, CNRS, Inria, Centrale Lille
UMR 9189 CRIStAL
F-59000 Lille, France
0000-0003-3034-873X

***Abstract*—Legacy software systems often lack extensive testing, but are assumed to behave correctly after years of bug fixes and stable operation. Migrating or modernizing these systems is challenging because there is little support for preventing regressions. Test carving addresses this problem by generating unit tests based on the current behavior of the system, treating it as an implicit oracle. In this paper, we present MODEST, a multi-language tool that generates unit tests by carving them from execution traces. MODEST processes method calls, including their receivers, arguments, and results, to recreate these invocations as unit tests. Its model-based approach allows it to support multiple languages. We detail how it can be extended to handle additional languages. MODEST aims to generate tests that are human-readable and maintainable over time. To achieve this, it reconstructs values as source code rather than relying on deserialization. We evaluate MODEST by generating tests for both Java and Pharo applications.**

***Index Terms*—Test Carving, Unit Test Generation, Execution Traces, Multi-language, Regression Testing**

I. INTRODUCTION

Unit tests are commonly used by developers to verify code functionality and prevent bugs, ensuring that the software behaves as expected. However, despite the benefits, creating and maintaining unit tests can be time-consuming, especially for legacy systems that often lack comprehensive test coverage. Legacy software systems have typically been in use for decades. Over time, bugs are gradually discovered and patched, resulting in a relatively stable system. However, when such systems need to be migrated or modernized, development teams are faced with the challenge of preventing regressions without the benefit of comprehensive test suites.

Test carving provides a solution to this problem by generating unit tests that capture the behavior of the system during execution. This technique uses execution traces to replay method calls as unit tests that verify that the expected behavior is maintained.

In this paper, we present MODEST, a test carving tool that generates unit tests by treating the current system behavior as an oracle. MODEST aims to be multi-language by using a model-based approach that allows it to work with different programming languages. However, it requires a parser to generate a model for a language, and an exporter to generate tests in the target language.

MODEST reconstructs the captured values as source code, with the goal of improving the readability and maintainability of the generated tests. We expect that this will help developers to better understand and update the tests, ultimately supporting long-term software evolution.

While there are similar approaches to test generation [1]–[3], these often have limitations, such as being language-specific and the tests suffer from a lack of readability. In contrast, MODEST provides a language-agnostic solution that aims to generate human-readable tests. This approach involves the use of metamodels to facilitate the representation and generation of unit tests. The use of metamodels provides a solution that is independent of the programming language and test framework, and allows for automated transformation and code generation. MODEST has already been successfully used to generate tests for Pharo [4], the language in which the tool is implemented[1].

We evaluate MODEST by generating tests for four real-world software systems. To demonstrate the multi-language capabilities of the tool, we chose two Java applications and two Pharo applications. Execution traces were obtained from existing tests or from scenarios provided by the developers. A total of 417 unit tests were generated for the four case studies. Of these, 163 tests were integrated into one of the Java applications.

The remainder of this paper is structured as follows: Section II presents related works, Section III details our approach, Section IV presents how MODEST can be adapted to new languages and test frameworks, Section V evaluates our tool

[1]Pharo is a pure object-oriented and dynamically typed language, see https://www.pharo.org.

by applying it to real-world software systems, and Section VI concludes the paper and presents future work.

II. Related Works

Several approaches to generating unit tests by replaying scenarios have been proposed recently. Most similar to ours, Tiwari et al. [2] propose a solution for production monitoring that generates unit tests by capturing values at runtime, and Alshahwan et al. [3] capture values during end-to-end tests to generate unit tests. Both of these approaches are specialized for Java, and they recreate the captured values by deserializing them within the tests. This deserialization process, while functional, often degrades the readability of the generated tests [5]. Wachter et al. [6] discuss the concept of recreating serialized Java values in plain code, which is consistent with our approach of promoting readability by avoiding deserialization within the tests.

A significant challenge in generating relevant unit tests is navigating the vast search space to identify values that trigger correct or critical behavior. Tools such as Randoop [7], which uses random testing, and EvoSuite [8], which uses evolutionary algorithms, often struggle to generate test cases for specific scenarios due to the inherent complexity of efficiently finding such values. Capturing runtime values during execution helps address this challenge by ensuring that tests are generated with values that are known to exercise meaningful behavior. Jaygarl et al. [1] addressed this issue by capturing desirable values at runtime to assist random testing tools such as Randoop in generating more relevant unit tests. Similarly, Wang et al. [9] introduce Replica, a technique to reduce the gap between tests and real-world behavior by collecting field execution data to cover untested code paths. Artzi et al. [10] propose ReCrash, a technique that captures runtime values to generate unit tests aimed at reproducing software failures. These contributions highlight how incorporating runtime values can significantly improve the relevance and effectiveness of generated tests.

Several recent studies have explored the use of execution traces for software testing, recognizing the valuable insight they provide into the runtime behavior of a program. Paiva et al. [11] propose a web testing approach that generates test cases from user execution traces. To improve the test suite, mutation operators are applied to these test cases to simulate potential real-world failures. Tests that produce different results are retained because they reveal additional behavior in the web application under test. Similarly, Salva et al. [12] focus on testing service compositions by extracting traces from event logs. These traces help identify recurring behaviors and create generic test cases, which are then used to generate test scripts and mock components for each service.

Other approaches to test generation include AI-based techniques and test amplification methods. AI-based test generation, in particular using large language models, is still a relatively new area of research. While they can generate tests from scratch without requiring existing test suites, they face challenges in terms of reliability. For instance, tests generated by GPT-4 fail to compile more than 50% of the time [13], and the generated tests often fail and require extensive human intervention [14]. In addition, there are concerns about AI frugality, where the resources required to train and run AI models can be substantial [15].

Test amplification, on the other hand, enhances and extends existing tests [16]. This inherently requires an existing set of tests to be amplified, which can be a limitation in scenarios where there are insufficient initial tests. While valuable, test amplification methods highlight the need for pre-existing test infrastructure, which MODEST aims to address by generating tests from scratch based on execution traces.

Our approach is not intended to replace test-driven development or traditional practices where developers write tests during the development process. We aim to generate unit tests on legacy software systems where tests are partially or completely missing.

III. Approach

Test carving uses execution traces to automatically generate unit tests by extracting relevant parts of the execution history. This technique replays methods with the same arguments and receiver, and ensures that the output matches what was recorded in the trace. This assumes that the current version of the software system under test is correct, allowing execution traces to serve as an oracle for expected behavior.

MODEST is a test carving tool based on the Moose platform[2]. Moose is a framework dedicated to software analysis [17]. It relies on the use of models to abstract, analyze, query, and navigate code. Abstracting from the specifics of any target system allows us to be multi-language by generalizing our test carving approach across different programming languages and frameworks.

Our process relies on five steps and three interconnected metamodels to generate unit tests, as shown in Fig. 1. The Code metamodel represents the codebase under test, and elements related to it are shown in green throughout this paper. The Value metamodel specifies the values used to test the codebase, with related elements shown in orange. The Unit Test metamodel captures the structure of unit tests, with related elements shown in blue. These metamodels abstract the key aspects of our approach, allowing us to apply MODEST to a wide range of contexts.

We will first describe the metamodels used by MODEST, and then explain the process steps in detail.

A. Description of the Metamodels

In this section, we describe the Unit Test and Value metamodels, and how they interact to represent unit tests. While the Code metamodel is an important component of our approach, it is already established within the Moose platform and has been discussed in previous research [17].

1) Unit Test Metamodel: The *Unit Test metamodel*, shown in Fig. 2, is a structured representation of the components that make up unit tests in object-oriented programming. It provides

[2]https://www.modularmoose.org

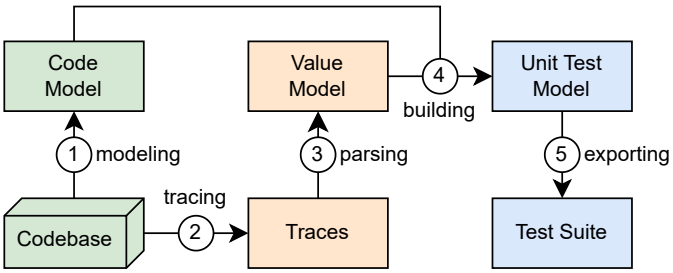

Fig. 1. The 5 steps of our approach. Elements representing the code under test are shown in green (left column), elements representing runtime information in orange (middle column), and elements representing the generated tests in blue (right column).

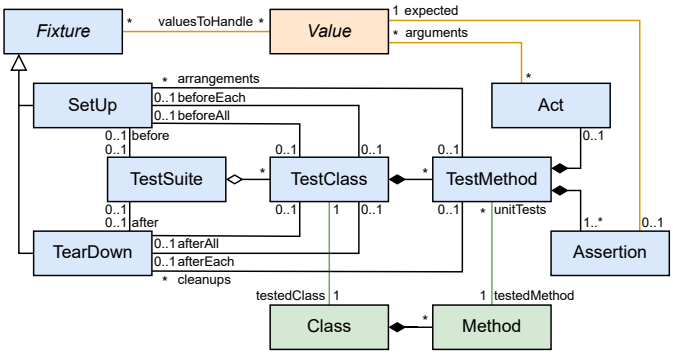

Fig. 2. Class diagram of the Unit Test Metamodel. Test entities are shown in blue, Code entities in green and Value entities in orange.

a way to represent, define, and generate unit tests by specifying their inputs, expected outputs, and any necessary setup and teardown fixtures.

The metamodel is built around the *Arrange Act Assert* (AAA) pattern [18], a widely used approach to structuring unit tests. The Arrange phase involves setting up the necessary preconditions for the test, usually initializing the receiver, arguments and expected values. The Act phase involves executing the method under test, using the values defined in the Arrange phase if any. Finally, the Assert phase verifies that the results of the Act phase are correct, typically by comparing them to expected values.

The *Act* entity represents the "Act" phase of the AAA pattern. This phase is responsible for executing the code under test and producing the actual result. In the philosophy of unit testing, each test method should verify one and only one behavior of the system under test. Therefore, a test method should have at most one *Act* to exercise the code under test and produce the actual result for that specific behavior. This ensures that each test method is focused on verifying a single behavior and simplifies the process of identifying and fixing issues that arise during testing.

An *Assertion* is a fundamental concept in unit testing. It is a condition that must be true for the test to pass and is used to verify that the behavior under test produces the expected outcome. Assertions are used to verify the outcome of a test by comparing expected and actual results, often manifested as return values. They can also be used to verify that a particular exception is thrown by the code under test. This can be particularly useful for testing error-handling code and ensuring that the system responds appropriately to unexpected conditions.

A *Fixture* ensures that the tests run in a consistent and isolated environment. When associated with a *TestMethod*, it corresponds to the code that is part of the "Arrange" phase of the AAA pattern. This is where values from the execution traces are reconstructed.

Finally, the *Value* entity is used to represent runtime data in unit tests. The relationship between the *Value* and *Assertion* entities represents the expected value of the assertion. When a *Value* is associated with an *Act*, it represents the arguments that will be passed to the method under test. When associated with a subclass of *Fixture*, it represents the values to set up or tear down.

Overall, the *Value* entity plays an important role in connecting the various elements of the unit test framework. In other approaches [2], [3], the setup of generated tests is achieved by deserializing data directly, which makes it difficult for developers to understand the tests. In this paper, we will explain how we use the Value metamodel to recreate values as code. The details of the *Value* metamodel are presented in the following section.

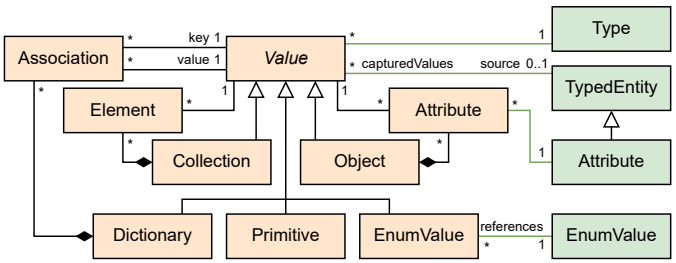

Fig. 3. Class diagram of the Value metamodel. Value entities are shown in orange and Code entities in green.

2) Value Metamodel: The *Value metamodel*, shown in Fig. 3, is a representation of runtime data independent of the programming language. It is inspired from the JSON representation format, representing types of values such as objects, primitives, collections, and dictionaries.

The *Value* entity serves as the root entity for all value representations in the metamodel. It is associated with a *TypedEntity*, which is a Code entity that represents a code element such as a variable or a method.

The *Value* entity is also in a relationship with the *Type* code entity, which is used to represent the type of the value. Even when a value is associated with a typed entity, its runtime type may differ from the static type due to polymorphism.

In the next section, we present how these metamodels are used in the test generation process.

B. Test Generation Process

We now detail the five steps involved in generating unit tests with MODEST.

Step 1: Obtain a model of the application. Using the capabilities of the Moose platform, we create a model of the application for which we want to generate tests. This model captures the structural aspects of the application, such as its classes and methods, and their relationships (invocations, accesses, references, inheritances). This step must be completed before test generation can begin, and needs to be done only once for any given version of the software. The following steps can be repeated as many times as desired.

Step 2: Produce traces of the application. Data about the execution of the software system is recorded as execution traces. Each trace corresponds to a specific method execution and must contain the following information: method identifier, arguments, return value, and the receiver object. Any of those can be omitted if there is no ground for it, for example procedures do not return values, and functions do not have a receiver. The method identifier is a way to know exactly which method was executed. For example, in the case of Java, this identifier consists of the fully qualified class name and the method signature, including parameter types. Each method execution results in a generated test. Thus, a method can have multiple tests generated that differ in the value of the receiver, arguments, or the return value.

Step 3: Import and parse trace data. The execution traces are imported into MODEST for parsing. During this step, the tool identifies and ignores duplicate traces that would end up generating identical tests by comparing the values. The Value metamodel, as described in Section III-A2, is instantiated. When dealing with serialized data, an importer parses the representation of method arguments, return value, and receiver.

The importer traverses the captured data, instantiates the appropriate Value entities and associates them with their corresponding Code entities. The method identifier is used to determine the origin of the trace, which corresponds to the method to test. Using the definition of the method, arguments are bound to their respective parameter based on their index. The return value is bound to the method. For methods, the receiver object must be present in the trace along with its actual type which can differ from the method class. For all object values, their attributes are bound to their code definition using the indications contained in the trace, such as their identifier.

Step 4: Build a unit test model. This step is the core of the approach and is independent of the language and the test framework as it is based on metamodels. During this step, the Unit Test metamodel, as described in Section III-A1, is instantiated. This is when the tests are created. Given the trace of a method execution, MODEST creates a *TestMethod* for the executed method. Eventually, a *TestClass* is created to hold the *TestMethod*. The *SetUp*, *Act* and *Assertion* entities are created and associated with the *TestMethod*. The *Value* entities of the arguments are associated with the *SetUp* and *Act*. Similarly, the *Value* entity of the receiver is also associated with the *SetUp* and *Act*. The *Value* entity of the return value is associated with the *SetUp* and *Assertion*.

Step 5: Export the unit test model into concrete tests. This step translates the unit test model into executable code. This requires knowledge of the target language and framework.

Within the test method, we follow the AAA pattern to organize the test logic clearly. First, the arrange phase recreates the receiver, arguments, and return value, and stores them in separate variables. The return value is stored in a variable explicitly named "expected". The exporter maps the *Value* entities to their variable names. Then, the act phase exercises the method under test using the variables for the arguments and receiver. The return value is stored in a variable named "actual". Finally, the assert phase checks that the actual and expected values are equal. The way to achieve this can vary depending on the test framework or assertion library.

The *Unit Test* and *Value* entities each have their own exporter. The test exporter is responsible for the test structure and calls the value exporter when necessary, as illustrated in Fig. 4. They can both generate an Abstract Syntax Tree (AST) as a medium which will be transformed into source code by a common AST exporter.

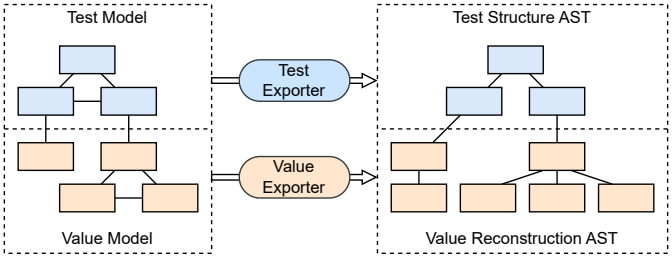

Fig. 4. Representation of interconnected test and value models and their export to AST

To promote readability and shorter tests, we propose to generate helper methods to recreate objects, collections, and dictionaries. Each helper method has no parameters and is specialized for a specific object, collection, or dictionary. Helper methods follow a naming pattern based on the role of the value they recreate and the associated test method. The name consists of:

- the prefix "given";
- if for an argument, the name of the associated parameter followed by "argument"; if for a receiver, the string "receiver"; if for the expected value, the string "expected";
- the string "for";
- the name of the test method

For example, the name of the helper method could be "given_lastName_argument_for_testSaveUser()". We also propose to export these methods into a dedicated class, one for each test class. This organization keeps the test methods clean and focused on the test logic, while the helper class handles the complexity of object creation and initialization.

We have described the five steps of our test generation process. We now detail the requirements for this process to work.

Listing 1
EXAMPLE INPUT FOR MODEST.

```
[
  {
    "class": "com.example.User",
    "method": "hasActiveSession()",
    "receiver": "...",
    "arguments": "[]",
    "result": "true"
  }
]
```

Listing 2
EXAMPLE JAVA OBJECTS WITH A CIRCULAR DEPENDENCY SERIALIZED WITH JACKSON.

```
{
  "@type": "com.example.User",
  "@id": 1,
  "name": "John Doe",
  "session": {
    "@type": "com.example.Session",
    "@id": 2,
    "active": true,
    "user": 1
  }
}
```

C. Process Requirements

We now detail the requirements for the input given to MODEST. This section lists the requirements of each step.

Step 1: Obtain a model of the application. When modeling the application, MODEST requires information about user-defined types. Specifically, we require their structural definition: namespace, modifiers (especially visibility), inheritance, attributes and method signatures. Most of those information are needed for step 5, see below. To create the tests, the namespace of the class is required to put the test class in the same namespace. A method signature is used as a unique identifier needed to associate a trace to the method that produced it.

Step 2: Produce traces of the application. The second step in our process is to collect the runtime execution data necessary to generate tests that accurately replay methods. The actual mechanism is irrelevant to MODEST, as long as the required data is accurately extracted. For example, in Java, bytecode instrumentation is a convenient approach that allows instructions to be injected at runtime, enabling the output of execution data without requiring any changes to the source code. Alternatively, instrumentation can be hard-coded directly into the target methods, although this is a more intrusive approach.

While the mechanics of data collection can vary widely between languages, the essential requirement remains the same: for each invocation of a target method, we capture its identifier and serialize its receiver, arguments, and result. An example input for MODEST is shown in Listing 1. When capturing runtime values, the traces are typically exported in a serialized format. Any serialization format can be used, as long as a suitable importer can deserialize it and instantiate the Value model. The serialized values must contain two key data points to recreate structured objects:

- the runtime type, if it differs from the static type;
- field identifiers that are mapped to their value.

Let us illustrate a possible serialization format that is supported by MODEST. JSON with metadata is a format produced using configurable serialization features of the Jackson library for Java. This format is designed to handle cyclic references and class identification, ensuring that complex data structures can be properly serialized and reconstructed. An example of two serialized Java objects with a circular dependency is shown in Listing 2. The *@id* field is used to uniquely identify objects, and the *@type* field specifies their runtime type. The example shows an instance of the *User* class with the id 1, and a *Session* instance with the id 2. The session references the user in its *user* field by its id. In a statically typed language, we can infer that an object is expected for this field, so finding an integer indicates that we are dealing with a reference.

As input, MODEST expects a list of execution traces. MODEST is flexible: in case a trace is missing any information among the receiver, arguments or result, it is still able to generate a test. Only the method identifier is absolutely required. For example, functions do not have a receiver, and procedures do not return a result. Currently, if the method does not return a value (declared void), or if the result is missing from the trace, MODEST generates a test without an assertion. Such tests are still useful as smoke tests, which focus on verifying basic functionality and stability. If the receiver is missing, MODEST will create a new empty instance of the tested method class by default.

In case the serialized data does not specify runtime type information for non-primitive values, MODEST will try to use the declared type of the value to reconstruct the object. For collections, it is only possible to infer the type of their elements if the collection is parameterized (eg. “List<User>”).

Step 5: Export the unit test model into concrete tests. To promote readability, MODEST recreates values as code. This is a difficult problem [6], and therefore has deep implications for the requirements. To reconstruct objects, MODEST looks for constructors and setters. Class inheritance information can be used to look up these methods, which may be declared in superclasses. However, there may be no obvious constructor or setter. It is also possible that the visibility of a method (for example, “private” in Java) prevents its use in test code. In both cases, a reflexive language helps to get around these restrictions.

In the next section, we will explain how to adapt MODEST to a new language or test framework.

IV. Adapting Modest to New Contexts

Modest is designed to be adaptable to the language, test framework, and serialization format. The import process is affected by the language and the serialization format. The export process is affected by the language and the test framework. To achieve this, Modest provides the adaptation points necessary to translate between specific constructs and the metamodels. These adaptation points allow it to map concrete language constructs, such as method signatures and object states, to the generalized metamodels. They are also used when the models are exported back into concrete test code.

Modest can be adapted for the modeling of the application, import of the execution traces, and export of the tests. The fourth step, building the test model (the core of our approach), is based on models which are agnostic to the language and test framework. Moose's modeling capabilities are based on a system of traits [19] that serve as abstractions for common programming concepts. For example, an entity representing a method in any language would always use the generic concept of a *Method*, which encapsulates common properties such as being named, having parameters, and so on. The use of traits is essential to our approach because it allows us to manipulate entities based on their roles rather than their language-specific implementations. By associating concrete language constructs with abstract traits, we can process and analyze different programming languages in a unified way. This trait-based system allows the generalization of our test carving approach across languages.

In the following, we outline the possible adaptations for the other steps. These adaptations show how Modest can be extended to support new contexts.

Step 1: Modeling an application. The first step in our test generation process is to model the target application with the Code metamodel. Modest is part of the Moose platform which models programming languages in a generic way. Languages currently supported by Moose include C/C++, Java, Pharo, Python, and TypeScript.

In practice, Modest does not need a complete model of the application. The metamodel only needs to contain the requirements listed for the first step in Section III-C.

Step 3: Importing and parsing execution traces. The third step is to import the execution traces obtained in the previous step into Modest. Before processing the traces, Modest performs an initial pass to filter out duplicates, ensuring that no equivalent tests are generated by comparing the recorded values. The remaining execution traces are then processed to create new tests.

Currently, Modest supports importing data in the Jackson format (JSON with metadata) and the XStream format (XML). Other formats could be used, provided an appropriate importer is implemented.

Step 5: Exporting the unit test model. The final extensible step in our process is the export of the unit test model into actual tests. At this stage, two key extension points come into play to handle new target languages and environments: the unit test exporter and the value exporter. These components work together to generate the final output in the form of executable tests.

Listing 3
Code recreating two Java objects.

```
User user = new User();
user.setName("John Doe");
Session session = new Session();
session.setActive(true);
session.setUser(user);
user.setSession(session);
```

The unit test exporter is responsible for generating the structure of the test cases. It is dependent on the language and test framework. This includes creating the test class, defining methods, and following the AAA pattern that is standard in unit testing. During this process, the unit test exporter interacts with the value exporter, asking it to recreate values for each step in the test that requires them. This includes the initialization of the receiver, the arguments, and the expected result.

The role of the value exporter is to generate the source code necessary to recreate each value. It is only dependent on the language. This ensures that the values used in the tests are the same as those captured during execution, to faithfully reproduce the recorded behavior. If the execution trace captured a structured object, the value exporter must produce the code that would generate an equivalent object in the target language. As explained in the requirements (Section III-C), this means finding the constructors and setters, or using the reflective capabilities of the language. Listing 3 shows an example of the Java code to recreate the serialized value of Listing 2.

It is interesting to note that some objects may have a representation in the trace that differs from their implementation. This may require knowledge of specific reconstruction schemes. For example, we may have a date represented by a "yyyy-mm-dd" string, and the only way to reconstruct it from the string is to use a factory. For such cases, the exporter can be specialized to include the necessary knowledge to reconstruct these special cases.

This modular design allows our approach to remain flexible and adaptable to new languages. By implementing new exporters tailored to the specific requirements of the target language and test framework, Modest can be extended to generate unit tests in new contexts.

V. Evaluation

In this section, we present the validation experiment for Modest. We first describe the experimental protocol we used, before presenting and commenting on the results of the experiments.

A. Protocol

We want to apply MODEST on software systems in multiple languages to demonstrate the multi-language capabilities of the tool. The software systems should be used in the real world, not toy projects, to show the applicability of MODEST in real scenarios. We do not limit ourselves to open-source projects because some industrial projects are closed-source, and they may have different characteristics. The study subjects should be of varied size and application domains.

Execution traces are a type of dynamic analysis that extracts runtime data during an execution scenario. This affects which methods are executed, thus traced, and consequently tested. We do not want to introduce a bias in the choice of the scenarios. Therefore we will rely on existing scenarios for each project.

We will evaluate the number of generated tests, the percentage that pass, and the readability of the generated tests. We want to test public methods in public classes that are not constructors or accessors.

To apply MODEST to different languages, we chose Java and Pharo. Both languages are object-oriented, Java is statically typed and Pharo is dynamically typed. In Java, developers can declare specific class and method visibility. In Pharo, developers have no control over visibility: all classes and methods are public, and fields are protected. The dynamic nature of Pharo highlights the importance of including type information in the execution traces. This is because the lack of declared types does not allow us to infer the types of parameters, return values, and object fields.

We selected two software systems developed in each language as our study subjects:

- Omaje, a customer subscription management project with a three-tier architecture used internally by an industry partner;
- Traccar[3], an open-source GPS tracking system, that also uses a three-tier architecture. It has been studied by Verhaeghe et al. [20], a multi-language GUI migration approach;
- Cormas[4], a standalone agent-based modeling platform with a graphical user interface. It is used by CIRAD for modeling real-world problems [21];
- DataFrame[5], a tabular data structure library for data analysis in Pharo.

The number of classes and methods in these projects is given in Table I. The two Pharo projects are smaller than the two Java projects, especially in the number of classes. However, the number of methods in each project is still significant.

To generate the necessary execution traces, we used an agent built with OpenTelemetry[6]. All values are serialized using the Jackson format. For Traccar and DataFrame, we did not select which methods to trace: we instrument all public methods, except constructors and accessors, in public classes. For Omaje and Cormas, we consulted with the developers who provided a list of classes and methods to target for testing.

[3]https://github.com/traccar/traccar/tree/v6.5

[4]https://github.com/cormas/cormas/tree/v0.95

[5]https://github.com/PolyMathOrg/DataFrame/tree/pre-v3

[6]https://opentelemetry.io/

TABLE I
CHARACTERISTICS OF THE STUDY SUBJECTS

PROJECT	LANGUAGE	#CLASSES	#METHODS
DataFrame	Pharo	29	1161
Cormas	Pharo	59	2036
Traccar	Java	1346	4688
Omaje	Java	1198	7006

To trace Traccar and DataFrame, we ran the existing unit tests. Using existing unit tests to generate the traces does not make much sense in real-world conditions, since MODEST should be used to generate tests where coverage is lacking. However, this approach allowed us to collect traces without prior knowledge of the projects, since execution was driven by the tests themselves, independent of our input.

For Omaje and Cormas, there is a notable lack of unit tests, so traces had to be obtained through scenarios provided by the developers. For Cormas, we executed a demonstration of the project. For Omaje, the developers obtained traces by running functional tests and using the tool to generate the tests themselves. Using functional tests to generate traces makes perfect sense compared to using unit tests, because it allows generating tests with a lower granularity.

For both trace generation and test generation, we followed a defensive strategy, stopping the process when a problem occurred to avoid generating tests that would not compile. During trace generation, errors can occur for example due to serialization. During test generation, errors can occur when trying to recreate objects that require special handling that is not possible or has not yet been implemented, such as recreating a private inner class in Java.

B. Results

We show the results of our evaluation in Table II. For each study subject, we show the methods for which tests were generated. For each tested method, we show how many tests were generated, and how many passed or failed. Note that since Omaje is proprietary software, we cannot disclose any information about the tested methods.

We have successfully generated 417 tests for all projects, and most of them pass (86%). For DataFrame and Omaje, 100% of the tests pass.

Looking at the failing tests, we can see that there are 50 for Cormas and 8 for Traccar. These failing tests are for four different methods, and all the tests generated for them fail. This indicates a problem with the methods themselves. We investigated the cause of these failures. It is due to incomplete serialization, which results in recreated objects that are missing necessary information. This can be due to transient fields or null fields that were ignored but were actually different from the default values. Without complete data, MODEST cannot accurately reproduce the state of the objects required by the test. This is the case for the tests in rows 16, 17, 22 and 27.

TABLE II
EXPERIMENTAL RESULTS OF TEST GENERATION WITH MODEST ON THE STUDY SUBJECTS. FOR THE TOTALS, WE GIVE THE PERCENTAGE OF PASSING AND FAILING TESTS.

#	CLASS_METHOD	#TESTS	#PASSING	#FAILING
1	DataFrame_asArrayOfColumns	4	4	0
2	DataFrame_calculateDataTypes	4	4	0
3	DataFrame_columnNames	27	27	0
4	DataFrame_columns	4	4	0
5	DataFrame_dataTypes	10	10	0
6	DataFrame_initialize	2	2	0
7	DataFrame_initializeColumns	1	1	0
8	DataFrame_initializeRows	1	1	0
9	DataFrame_numberOfColumns	10	10	0
10	DataFrame_numberOfRows	5	5	0
11	DataFrame_privateRowNames	5	5	0
12	DataFrame_rowNames	1	1	0
13	DataFrame_setDefaultRowColumnNames	4	4	0
	DATAFRAME TOTAL	**78**	**78** (100%)	**0** (0%)
14	CMEntity_delete	9	9	0
15	CMEntity_init	5	5	0
16	CMEntity_initId	27	0	27
17	CMSpatialEntity_addOccupant	5	0	5
18	CMSpatialEntity_allOccupants	18	18	0
19	CMSpatialEntity_destroyed	9	9	0
20	CMSpatialEntity_initOccupants	18	18	0
21	CMSpatialEntity_neighbourhood	18	18	0
22	CMSpatialEntity_neighbourhoodWithNils	18	0	18
	CORMAS TOTAL	**127**	**77** (60.6%)	**50** (39.4 %)
23	GeofenceCircle_containsPoint	2	2	0
24	GeofenceCircle_distanceFromCenter	2	2	0
25	GeofenceCircle_fromWkt	1	1	0
26	GeofenceCircle_toWkt	1	1	0
27	GeofencePolyline_containsPoint	8	0	8
28	GeofencePolyline_fromWkt	4	4	0
29	GeofencePolyline_toWkt	1	1	0
30	Network_addCellTower	10	10	0
31	Network_addWifiAccessPoint	10	10	0
32	Position_addAlarm	10	10	0
	TRACCAR TOTAL	**49**	**41** (83.7%)	**8** (16.3%)
	OMAJE TOTAL	**163**	**163** (100%)	**0** (0%)
	TOTAL	**417**	**359** (86.1%)	**58** (13.9%)

Another known cause that may arise is when a method depends on external state that is not captured in the execution traces. In such cases, MODEST cannot recreate the required state. For example, if execution depends on the state of a global variable that is not referenced by the receiver, arguments, or result, that state will not be recreated, usually resulting in a test failure.

These causes are consistent with the findings of Tiwari et al. [2]. However, they found two additional causes that did not affect us in this case study. One is that a class can define its own serialization behavior, which can result in an incomplete or incorrectly serialized object, similar to the issue with transient fields. The other cause is due to the use of ASSERTEQUALS, which relies on the EQUALS method. By default, this method uses object identity unless it is redefined. This can lead to assertion failures when two objects are structurally equivalent but not identical. We can avoid this problem by using an assertion library to enable structural equality. It is worth noting that since we have the data necessary to reconstruct objects, we could generate methods to verify structural equality without relying on external libraries.

Regarding the readability of generated tests: some of the tests generated for Omaje were modified by the developers. For example, one modification was to change a test to check only the filename in an absolute path string instead of the entire path. These modifications hint at the good readability and maintainability of the generated tests. We show examples of generated tests in Listing 4, Listing 5 and Listing 6, as well as in our replication package[7]. These can be compared to tests that recreate values using deserialization, for example in listings 7 and 8 of Tiwari et al. [2].

For the Traccar test shown in Listing 4, we also show the generated helper method in Listing 5. The helper uses the simplest constructor it could find, which is the empty constructor. Then MODEST tried to find setters for the fields that are not set by the constructors but did not find any. For this reason, it uses reflection (using our SETFIELD method) to initialize the fields. In this particular case, a different constructor would have been more appropriate. Reconstructing objects as plain code is a difficult problem discussed by Wachter et al. [6]. Improving MODEST on this topic is part of our future work.

[7]TO BE PROVIDED

The Pharo test for DataFrame shown in Listing 6 also follows the AAA structure, with clear separation of phases. Again, helper methods are used to make the arrangement phase concise and readable. The assertion is handled by a Pharo library for structural equality to support test reliability.

C. Threats to Validity

Internal validity is the extent to which we can draw a causal link between the treatment in the experiment and the response.

We have used a consistent methodology across all of the study subjects to ensure that differences in results are not due to different experimental procedures. All execution scenarios used to generate traces are provided by the projects themselves. There may be a bias in what the scenarios exercise, because they are created by the developers who are influenced by their own familiarity with the projects. We believe that this bias is toward what is important to test. We also believe that this bias is consistent with real-world use of MODEST.

External validity is the extent to which we can apply the findings of the study to a broader context.

We tried to ensure a diversity of study subjects by selecting projects projects in different languages, domains, sizes, closed or open source. A possible bias is that we only experimented with object-oriented languages. In the future, we would like to experiment with languages in other paradigms, especially procedural.

Construct validity discusses the extent to which the results really measure what they are supposed to measure.

We acknowledge that our evaluation of readability is somewhat subjective. We are not aware of any automated solution for evaluating readability. In the future, we would have to ask developers to evaluate it manually. Despite of all this, the developers of the Omaje application were still willing and able to modify the generated tests themselves. We see this as an indication that they were able to read and understand the tests.

Reliability considers the extent to which the results can be reproduced when the research is repeated under the same conditions.

We made sure there was no bias in the scenarios we ran by using existing scenarios from each of them. Only for Omaje we are unable to provide the source code and scenarios due to its proprietary nature. We provide a replication package with the generated tests and instructions on how to replicate the experiment, detailing which versions of the study subjects to use.

VI. CONCLUSION AND FUTURE WORK

In this paper, we present MODEST, a language-agnostic test carving tool designed to generate unit tests from execution traces. Our primary goals are to ensure broad applicability across different programming environments, and to focus on the readability and maintainability of the generated tests. Recreating values as code is pivotal to achieving this goal.

To validate MODEST, we applied it to four real-world software systems. The multi-language aspect was validated by applying it in the context of the Java and Pharo programming languages. We have also demonstrated the ability of MODEST to recreate values as code, which improves readability. Furthermore, the tool was used by the developers of Omaje, an industrial Java software system, to generate 163 tests that were then integrated. Some of the tests were edited manually, confirming that the developers found the tests human-readable and maintainable.

To acknowledge some of the limitations of MODEST, it currently does not implement mechanisms to handle most side effects, making methods without side effects the preferred targets for testing. Side effects include file system interactions, network operations, global state changes, elements of randomness, and so on.

Looking ahead, several avenues for future work have been identified. We plan to explore object pruning techniques based on call graphs to minimize test size without sacrificing coverage. Given our access to the application and its values, it is possible to prune unused values in objects, which is especially critical for industrial use cases that can involve enormous amounts of data. By minimizing the size of the tests, we aim to improve both their readability and their debugging capabilities. Additionally, we intend to evaluate developer feedback regarding the readability of generated tests, specifically comparing the approaches of deserializing values versus recreating them as source code. Finally, we intend to present the use of MODEST in collaboration with our industry partner to improve their testing processes and support the integrity of their migration to newer technologies.

Listing 4
JAVA TEST GENERATED FOR TRACCAR

```java
@Test
public void testDistanceFromCenter() {
  /* ARRANGE */
  GeofenceCircle receiver = given_receiver_for_testDistanceFromCenter();
  /* ACT */
  double actual = receiver.distanceFromCenter(55.75545, 37.61921);
  /* ASSERT */
  assertThat(actual).isEqualTo(163.77736255543593);
}
```

Listing 5
JAVA HELPER GENERATED FOR THE TEST OF LISTING 4

```java
public static GeofenceCircle given_receiver_for_testDistanceFromCenter() {
  GeofenceCircle geofenceCircle32 = new GeofenceCircle();
  setField(geofenceCircle32, GEOFENCECIRCLE_CENTERLATITUDE, 55.75414);
  setField(geofenceCircle32, GEOFENCECIRCLE_CENTERLONGITUDE, 37.6204);
  setField(geofenceCircle32, GEOFENCECIRCLE_RADIUS, 100.0);
  return geofenceCircle32;
}
```

Listing 6
PHARO TEST GENERATED FOR DATAFRAME

```
testRowNames
        | actual expected anArray receiver |
        "ARRANGE"
        expected := self given_expected_for_testRowNames.
        anArray := self given_anArray_argument_for_testRowNames.
        receiver := self given_receiver_for_testRowNames.
        "ACT"
        actual := receiver rowNames: anArray.
        "ASSERT"
        self assert: actual deepEquals: expected
```

References

[1] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "OCAT: object capture-based automated testing," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 159–170. [Online]. Available: https://doi.org/10.1145/1831708.1831729

[2] D. Tiwari, L. Zhang, M. Monperrus, and B. Baudry, "Production monitoring to improve test suites," *IEEE Transactions on Reliability*, vol. 71, no. 3, pp. 1381–1397, Sep. 2022. [Online]. Available: http://dx.doi.org/10.1109/TR.2021.3101318

[3] N. Alshahwan, M. Harman, A. Marginean, R. Tal, and E. Wang, "Observation-based unit test generation at Meta," 2024.

[4] G. Darbord, F. Vandewaeter, A. Etien, N. Anquetil, and B. Verhaeghe, "Modest-Pharo: Unit test generation for Pharo based on traces and metamodels," in *IWST 2024: International Workshop on Smalltalk Technologies*, Jul. 2024. [Online]. Available: https://hal.science/hal-04622256

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

[6] J. Wachter, D. Tiwari, M. Monperrus, and B. Baudry, "Serializing java objects in plain code," 2024.

[7] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2007.

[8] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[9] Q. Wang and A. Orso, "Improving testing by mimicking user behavior," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 488–498.

[10] S. Artzi, S. Kim, and M. D. Ernst, "ReCrash: Making software failures reproducible by preserving object states," in *ECOOP 2008 – Object-Oriented Programming*, J. Vitek, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 542–565.

[11] A. C. R. Paiva, A. Restivo, and S. Almeida, "Test case generation based on mutations over user execution traces," *Software quality journal*, vol. 28, no. 3, pp. 1173–1186, 2020.

[12] S. Salva and J. Sue, "Automated test case generation for service composition from event logs," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2023, pp. 127–134. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ASEW60602.2023.00022

[13] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No more manual tests? Evaluating and improving chatgpt for unit test generation," *arXiv preprint arXiv:2305.04207*, 2023.

[14] R. A. Poldrack, T. Lu, and G. Beguš, "Ai-assisted coding: Experiments with gpt-4," *arXiv preprint arXiv:2304.13187*, 2023.

[15] A. S. George, A. H. George, and A. S. G. Martin, "The environmental impact of AI: A case study of water consumption by Chat GPT," *Partners Universal International Innovation Journal (PUIIJ)*, vol. 01, no. 02, Apr. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7855594

[16] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A snowballing literature study on test amplification," *Journal of Systems and Software*, vol. 157, Nov. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219301736

[17] N. Anquetil, A. Etien, M. H. Houekpetodji, B. Verhaeghe, S. Ducasse, C. Toullec, F. Djareddir, J. Sudich, and M. Derras, "Modular Moose: A new generation of software reengineering platform," in *International Conference on Software and Systems Reuse (ICSR'20)*, ser. LNCS, no. 12541, Dec. 2020, pp. 119–134.

[18] K. Beck, *Test Driven Development: By Example*. Addison-Wesley Longman, 2002.

[19] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, "Traits: Composable units of behavior," in *Proceedings of European Conference on Object-Oriented Programming*, ser. LNCS, vol. 2743. Springer Verlag, Jul. 2003, pp. 248–274.

[20] B. Verhaeghe, A. Shatnawi, A. Seriai, A. Etien, N. Anquetil, M. Derras, and S. Ducasse, "From GWT to Angular: An experiment report on migrating a legacy web application," *IEEE Software*, 2021.

[21] P. Bommel, N. Becu, C. Le Page, and F. Bousquet, "Cormas: an agent-based simulation platform for coupling human decisions with computerized dynamics," in *Simulation and gaming in the network society*. Springer, 2016, pp. 387–410.