Modest-Pharo: Unit Test Generation for Pharo Based on Traces and Metamodels

Gabriel Darbord, Fabio Vandewaeter, Anne Etien, Nicolas Anquetil, Benoit Verhaeghe

► **To cite this version:**

Gabriel Darbord, Fabio Vandewaeter, Anne Etien, Nicolas Anquetil, Benoit Verhaeghe. Modest-Pharo: Unit Test Generation for Pharo Based on Traces and Metamodels. IWST 2024: International Workshop on Smalltalk Technologies, Jul 2024, Lille, France. hal-04622256v2

HAL Id: hal-04622256

https://hal.science/hal-04622256v2

Submitted on 27 Aug 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Modest-Pharo: Unit Test Generation for Pharo Based on Traces and Metamodels

Gabriel Darbord[1], Fabio Vandewaeter[1], Anne Etien[2], Nicolas Anquetil[2] and Benoit Verhaeghe[3]

[1]*Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France*
[2]*Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France*
[3]*Berger-Levrault, France*

Abstract

Unit testing is essential in software development to ensure code functionality and prevent the introduction of bugs. However, challenges such as time constraints and insufficient resource allocation often impede comprehensive testing efforts, leaving software systems vulnerable to regression. To address this issue, we introduce Modest, a language-agnostic approach to unit test generation that uses metamodels and execution traces. This method ensures non-regression by replaying scenarios captured from real-world executions. We demonstrate the application of Modest to Pharo codebases by generating unit tests for two projects. A total of 188 tests were generated and compared to existing tests based on mutation coverage, and we found that combining existing and generated tests increased coverage.

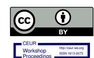
Keywords

test generation, unit tests, regression testing, trace-based, metamodels, Pharo

1. Introduction

Unit testing is an essential part of software development, serving as a critical mechanism for verifying code functionality and mitigating the risk of introducing bugs. Despite its importance, time constraints and inadequate resource allocation often prevent the widespread adoption of unit testing practices. This can result in codebases that lack proper testing, leaving software systems vulnerable to bugs, issues, and regressions [1, 2].

While existing approaches [3, 4, 5] showed promising results in test generation, they have some limitations, such as being specific to a particular programming language or testing framework. To address this issue, we propose Modest, a language-agnostic approach to test generation. 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 testing framework. It enables automated transformation and code generation. Specifically, we use three metamodels: the unit test metamodel, which represents unit test elements; the code metamodel, which represents the codebase; and the

IWST 2024: International Workshop on Smalltalk Technologies, July 9-10, 2024, Lille, France
gabriel.darbord@inria.fr (G. Darbord); fabio.vandewaeter.etu@univ-lille.fr (F. Vandewaeter); anne.etien@inria.fr (A. Etien); nicolas.anquetil@inria.fr (N. Anquetil); benoit.verhaeghe@berger-levrault.com (B. Verhaeghe)
0000-0001-7364-7567 (G. Darbord); 0000-0003-3034-873X (A. Etien); 0000-0003-1486-8399 (N. Anquetil); 0000-0002-4588-2698 (B. Verhaeghe)


© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

value metamodel, which specifies the values used to test the codebase. Our approach is not intended to replace test-driven development or classic development practices where tests are written during the development phase. Our approach aims to generate unit tests on legacy software systems where tests are partially or completely missing. The generated tests help manage regression and identify new software bugs in existing areas of a system after changes have been made.

Furthermore, we aim to generate maintainable test code that is easy for humans to understand. Human-readable and maintainable tests make it easier for developers to understand how the code works and to make changes to the codebase with confidence [6]. In addition, human-readable tests can be helpful when onboarding new developers to a project, or when maintaining code written by others. Ultimately, maintainable tests can reduce the amount of time spent debugging and fixing issues in the codebase.

To generate realistic tests, we use application traces consisting of method arguments and return values to leverage values from real business scenarios. Traces refer to the sequential recording of actions or operations in a system during its execution. This information is critical because it provides an accurate representation of how the software behaves at runtime.

In our previous work [7], we introduced two metamodels: the Value metamodel for representing runtime values, and the Unit Test metamodel. Our approach is based on Moose, a platform for software and data analysis[1]. This infrastructure allows us to extract knowledge from software systems and to apply our approach across programming languages.

In this paper, we present our five-step approach in Section 2. In Section 3 we explain the implementation of some steps in the case of test generation in Pharo. Section 4 presents some results on concrete Pharo applications. We discuss related works in Section 5 In Section 6, conclusions are drawn and perspectives are proposed.

2. Modest: a Unit test Generation Approach

Modest uses method execution traces to generate unit tests. This approach assumes that the current version of the software system for which tests are being generated is correct, allowing execution traces to be used as an oracle. The process relies on five steps to generate unit tests, as shown in Figure 1.

2.1. Prerequisites

There are two independent requirements that must be met before the test generation process can begin.

Step 1: Obtain a model of the application. Using the capabilities of the Moose platform, we create a comprehensive model of the application for which tests are to be generated. This model captures the structural aspects of the application, such as its classes and methods, and their relationships.

Step 2: Produce traces of the application. Data about the execution of the current version of the software system is recorded as a trace. Each trace corresponds to a specific method

[1] https://moosetechnology.org/

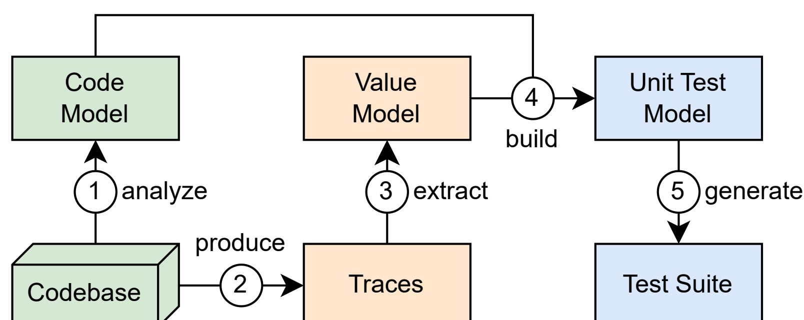


Figure 1: The 5 steps of the Modest approach. Entities representing the code to be tested are shown in green (left column), entities representing runtime information are shown in orange (middle column), and entities representing the generated tests are shown in blue (right column).

execution and must contain the following information: method identity, arguments, return value, and the receiver object. The method identity is a way to know exactly which method was executed. This is critical because multiple methods in the system can have the same signature due to polymorphism. This identity consists of the fully qualified class name and the method signature, including parameter types in the case of statically typed languages.

For a given project, any method that has no side effects and returns a value is a candidate for instrumentation. Side effects include use of the file system, graphical interfaces, network, global states, and randomness. Each execution of an instrumented method can result in a generated test. Thus, for a given executable comment or existing test, multiple tests can be generated that differ in the value of the arguments and the return value.

2.2. Test Generation Process

Once the prerequisites are met, the following steps are performed iteratively for each test generation cycle.

Step 3: Import and parse trace data. Traces are imported into Modest and reified to conform to a specific format. This ensures that the imported traces are represented consistently, regardless of the original storage format. The serialized data contained in each trace is parsed to extract relevant information. This parsed data is then reified using our Value metamodel, transforming it into a standard format for further processing. The Value metamodel bridges static code elements, such as method parameters, with dynamic runtime values, such as method arguments.

Step 4: Build a unit test model. The Unit Test metamodel presented in [7] is agnostic to the language and the testing framework used. It is built around the *Arrange Act Assert* (AAA) pattern, a widely used approach to structuring unit tests. We use the trace of a particular method execution to determine the test class and method, as well as the arrange, act, and assert phases of a unit test. The executed method determines the test method, while its class determines the test class. The method arguments determine the arrange and act phases of the test, where they are set up, used, and torn down. Finally, the result obtained from the trace determines the assert phase. We use the result as a test oracle, and the actual return value obtained in the act phase is

compared to the expected value from the trace.

Step 5: Export the unit test model into concrete tests. The unit test model is translated into executable test code specific to the target language and the specific testing framework. This translation involves converting the model into Abstract Syntax Tree (AST) nodes. Finally, the AST nodes are used to generate the actual unit tests.

3. Adapting the Modest Approach to Pharo

In this section, we outline our methodology for generating unit tests for Pharo software systems. The Moose platform is used for application modeling, and a Pharo implementation of OpenTelemetry is used to produce execution traces. Consequently, details are given for language-dependent steps, i.e. steps 2, 3, and 5.

Step 2: Produce traces of the application. We use a Pharo implementation[2] of OpenTelemetry to generate execution traces of the application. OpenTelemetry[3] is an open-source observability framework and standard designed to generate, collect, and export telemetry data such as traces. It provides tools and APIs for instrumenting applications to monitor and analyze their behavior. Our implementation uses MetaLinks [8, 9], which allows the execution of instrumentation code before and after the method on which it is installed. We use this mechanism to record the method identity, arguments, return value, and the receiver. The instrumentation does not propagate to outgoing calls, only the targeted methods are traced.

This preliminary step is only concerned with generating trace data. These traces will be fed into Modest in the following step, which will take place at a later date and possibly in a different Pharo image. Thus, the recorded data must be serialized for storage. In addition, we require that the serialized objects contain enough information to be correctly represented by the value metamodel, such as their runtime type.

The STON library[4] encodes the runtime type data we need, but it is not able to serialize all types of objects. In addition, STON allows developers to define a custom serialization format for their class. While this customization is useful, it makes the object encoding opaque to external tools such as Modest. Consequently, we developed a custom library inspired by Jackson[5], called PharoJackson[6], with the goal of being able to serialize any object to JSON in a consistent way. Similar to STON and Jackson, our library includes metadata to express the object type and handles circular references.

Step 3: Import and parse trace data. When the execution traces are imported into Modest, the data they contain is parsed to extract the relevant information. First, the method identity is used to determine the origin of the trace, corresponding to the method to be tested. For Pharo, this consists of the method selector and the name of the defining class.

Then, the serialized data containing the method arguments, return value, and receiver is deserialized from JSON to basic data structures: dictionaries, arrays, strings; and primitive data

[2] https://github.com/Gabriel-Darbord/opentelemetry-pharo
[3] https://opentelemetry.io/
[4] https://github.com/svenvc/ston
[5] https://github.com/FasterXML/jackson
[6] https://github.com/Modest-Project/PharoJackson

types: numbers, booleans, and null values. Except for dictionaries, all these types represent instances of their corresponding class in Pharo. For example, a JSON array corresponds to an instance of a Pharo Array.

Listing 1: User and Session objects serialized with PharoJackson

```
1  {
2    "@type": "User",
3    "@id": 1,
4    "name": "John Doe",
5    "session": {
6      "@type": "Session",
7      "@id": 2,
8      "active": true,
9      "user": { "@ref": 1 }
10   }
11 }
```

Dictionaries are a special case because they are used to represent objects. Their key-value pairs correspond to attribute names and values (e.g. in Listing 1, the name attribute on line 4). In addition, metadata is added by PharoJackson: the `@type` value indicates the class of the object (e.g. line 2 indicates it is an instance of the User class), and the `@id` value is an identifier for handling circular dependencies (e.g. line 3). If the same object is referenced more than once, it is subsequently represented by a dictionary with a `@ref` value indicating the identifier of the corresponding object (e.g. line 9).

Thus, deserializing the trace data returns a graph of basic data structures. The importer of the Value metamodel is designed to interpret this specific format. It traverses the graph and instantiates the corresponding Value entities into a model.

Step 5: Export the unit test model into concrete tests. The unit test model is translated into executable test code specific to the Pharo language and the SUnit testing framework. Each element of the model is systematically visited.

Test classes are created using Pharo's built-in class creation API. For clarity and separation from existing tests, newly created test classes are named by appending `ModestTest` to the name of the tested class, e.g. in Listing 2. As part of the class creation process, each test class is then assigned to an appropriate package. Following Pharo's naming conventions, the test package is named after the package of the tested class, with the suffix `-Tests` added. If the specified test package does not exist, it will be created automatically.

Listing 2: Definition of the generated test class for the DataFrame class, from the package of the same name.

```
TestCase << #DataFrameModestTest
    slots: {};
    package: 'DataFrame-Tests'
```

After visiting a test class within the unit test model, the process moves on to exporting its test methods along with their associated arrange, act, and assert entities. These three entities are linked to value entities, which are visited by a specialized visitor responsible for generating

the AST to recreate the values as code, e.g. in Listing 3. Both visitors work together to generate the AST of the test method, which is then materialized and installed in the test class.

Listing 3: Generated code recreating the object from Listing 1.

```
1 (user := User new)
2   name: 'John Doe';
3   session: (Session new
4     active: true;
5     user: user;
6     yourself);
7   yourself
```

4. Modest in Action on Pharo Projects

In this section we evaluate our approach on real Pharo projects. First, relevant projects were selected. Then they were instrumented to generate traces. Finally, we present the generated test cases and the benefits of our approach.

4.1. Selection of Projects

As explained earlier, our approach is based on execution traces. There are several ways to get them in Pharo, such as manually executing the software to be tested. However, this can be difficult if we are not a user or developer of the software system; it requires expert knowledge. Therefore, alternative ways to generate execution traces are needed. As it happens, there are other ways to run valid execution scenarios: tests and examples, such as executable comments or class-side examples using the `<example>` pragma. Such examples are very common in kernel packages and graphical projects. However, since our approach uses metalinks to generate the trace, it is not possible to select projects from the kernel that are used by the instrumentation itself, such as the Boolean or Collection packages, as this would break the image. Also, as explained in Section 2, our approach does not currently deal with graphical applications, as it requires that the tested method returns a value. Side effects and randomness are also not handled yet, which limits the choice of projects.

Two projects were selected: DataFrame[7] and LabelContractor[8]. DataFrame is a tabular data structure for data analysis in Pharo. It organizes and represents data in a tabular format, similar to a spreadsheet or database table. It also provides several algorithms for data manipulation. For our evaluation we only considered the `DataFrame` class. The LabelContractor project is used to reduce the size of labels for graphical interfaces using different strategies. It currently provides 13 different contraction strategies and two ways to combine them. For our evaluation, we considered the project's main class, a tokenizer class, a helper class, and seven strategies. We report information about the selected classes in Table 1.

In the case of Dataframe, traces result from running existing tests. In the case of Label-Contractor, traces result from running existing tests and executable comments. In both cases,

[7]https://github.com/PolyMathOrg/DataFrame
[8]https://github.com/moosetechnology/LabelContractor

Project	Tested Classes	Methods	Existing Tests	Executable Comments	Covered Methods	Mutation Coverage
DataFrame	1	187	275	0	144	59%
LabelContractor	10	64	31	18	44	56%

Table 1
Selected Pharo projects and the number of evaluated classes. The table shows the number of methods, existing tests, and executable comments for the selected classes. It also shows the number of methods covered by tests and comments, representing the methods for which tests were generated.

our approach generates tests from these traces. Since tests already exist for these projects, it is possible to compare them with our generated tests in terms of mutation coverage. To obtain these measurements, we used the MuTalk[9] library.

4.2. Results

We generated tests for the previously introduced projects and classes. To reduce the number of generated tests, we recorded only the first execution of each instrumented method. An example of an existing test is shown in Listing 4, and the test that was generated from its execution is shown in Listing 5.

We now evaluate how the mutation coverage of the existing tests compare to our generated tests. We also look at how the coverage evolves when both existing and generated tests are considered. Our results are reported in Table 2.

Listing 4: Existing test from the tokenizer class of the LabelContractor project.

```
1 testTokenize
2
3   self
4     assert: (LbCTokenizer new tokenize: 'CK123J')
5     equals: #( 'C' 'K123' 'J' ) asOrderedCollection
```

Listing 5: Test generated from the execution trace of Listing 4.

```
1 testTokenize
2
3   | expected aString lbCTokenizer actual |
4   expected := OrderedCollection withAll: { 'C'. 'K123'. 'J' }.
5   aString := 'CK123J'.
6   lbCTokenizer := LbCTokenizer new.
7   actual := lbCTokenizer tokenize: aString.
8   self assert: actual equals: expected
```

The reason for failed tests is that there are still some objects that are not serializable by our library, such as closures. The DataFrame project has a higher number of failed tests compared to LabelContractor. This difference can be attributed to the greater complexity of the DataFrame project, which uses more objects that are currently not serializable.

[9] https://github.com/pharo-contributions/mutalk

Project	Generated Tests	Passes	Fails	Mutation Coverage	Combined Mutation Coverage
DataFrame	144	114	30	43%	64%
LabelContractor	44	42	2	43%	59%

Table 2
Generated tests for selected Pharo projects and their results. The table shows the number of generated tests, the number of tests that passed and failed, and the mutation coverage achieved by these tests. The combined mutation coverage indicates the coverage when both existing and generated tests are evaluated together.

For DataFrame, the mutation coverage achieved by the generated tests is lower than that of the existing tests (43% compared to 59%). However, when both existing and generated tests are combined, the mutation coverage improves to 64%. For LabelContractor, the mutation coverage achieved by the generated tests is 43%, lower than the existing test coverage of 56%. When combined, the mutation coverage also improves to 59%.

We can see that more mutants are killed by existing tests than by generated tests. This can be explained by the fact that existing tests often use auxiliary methods to initialize test values during the setup phase. In contrast, the generated tests rely on a structural reconstruction approach based solely on constructors (new in Pharo) and accessors, or on reflection. During the experiment, mutations were generated for entire classes rather than for specific methods, so existing tests were more likely to encounter and kill a mutation because they execute methods more often and with different arguments.

These results indicate that the combination of generated and existing tests leads to higher mutation coverage for both projects. The increase can be attributed to the use of structural equality between actual and expected results in the generated tests. This exhaustive recursive comparison helps to identify and kill more mutants than a standard equality check.

A threat to the validity of the generated tests is their reliance on execution traces. These traces are derived from specific scenarios, and the coverage and effectiveness of the generated tests are inherently tied to the completeness of those scenarios. If the execution traces do not cover relevant code paths or edge cases, the generated tests will also lack coverage in these areas.

5. Related Works

EvoSuite [3] is characterized by its ability to generate JUnit test cases using evolutionary algorithms, with a specific focus on Java. One of its strengths is achieving high levels of code coverage, including branch and line coverage. However, its generated unit tests often have a distinct style that differs from human-written tests, which can affect their readability [10]. SmallEvoTest [11] generates unit tests for dynamically typed programming languages, specifically Pharo and GToolkit, by using a type-profiling mechanism and a genetic algorithm to evolve the unit tests. In contrast to these language-specific, evolutionary algorithm-driven approaches, our approach aims to be language-agnostic and uses execution traces to generate

tests. We also focus on generating code that is more comprehensive for humans.

Several research studies have explored the use of execution traces for software testing, recognizing the valuable insight they provide into the behavior of a program at runtime. One web testing approach generates test cases from user execution traces [12]. To improve the test suite, mutation operators were applied to these test cases, simulating potential real-world failures. Tests that yielded different results were kept because they revealed additional behavior in the web application being tested. Techniques such as Daikon's invariant inference, which identifies likely invariants from execution traces, demonstrate the effectiveness of trace-based testing [13]. In the future, we could use similar methods to identify interesting test scenarios from traces.

In recent years, test generation tools using deep learning have attracted considerable interest. Among these tools, AthenaTest [4] stands out for its ability to generate unit test cases for Java programs by learning from actual methods and developer-written tests. Developer surveys indicate that AthenaTest outperforms other tools such as EvoSuite in both test coverage and readability. Building on AthenaTest, A3Test [5] introduces improvements by integrating assertion knowledge and ensuring consistency in naming and test signatures, resulting in improved correctness and method coverage. CodeT [14] presents a method that uses pre-trained language models to automatically generate test cases to evaluate the quality and correctness of code solutions. Despite these advances, deep learning-based tools still face notable challenges because they require extensive training data and significant computational resources.

6. Conclusion

In this paper, we introduced Modest, a language-agnostic approach to test generation that uses metamodels to generate unit tests. This approach ensures non-regression by replaying scenarios captured by execution traces. Finally, we showed how Modest can be applied to Pharo by generating unit tests for two projects.

Looking ahead, several avenues for further development of Modest are possible. These include experimenting with trace selection and mutation [12], mining for invariants [13], optimizing the generated test suite through coverage modeling, and pruning recreated objects to focus on relevant data. In addition, we plan to evaluate our approach on a larger scale to better understand its effectiveness and applicability. A key aspect of future work will be the criteria for selecting relevant scenarios or traces, which are currently determined by the user. By addressing these areas, we aim to further refine Modest and increase its utility in managing regression in software systems.

References

[1] P. Runeson, A survey of unit testing practices, IEEE Software 23 (2006). doi:10.1109/MS.2006.91.

[2] F. Trautsch, J. Grabowski, Are there any unit tests? an empirical study on unit testing in open source python projects, in: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017, pp. 207–218. doi:10.1109/ICST.2017.26.

[3] G. Fraser, 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, ESEC/FSE '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 416–419. URL: https://doi.org/10.1145/2025113.2025179. doi:10.1145/2025113.2025179.

[4] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, N. Sundaresan, Unit test case generation with transformers and focal context, CoRR abs/2009.05617 (2020). URL: https://arxiv.org/abs/2009.05617. arXiv:2009.05617.

[5] S. Alagarsamy, C. Tantithamthavorn, A. Aleti, A3test: Assertion-augmented automated test case generation, 2023. arXiv:2302.10352.

[6] E. Daka, J. Campos, G. Fraser, J. Dorn, W. Weimer, Modeling readability to improve unit tests, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA, 2015, pp. 107–118. URL: https://doi-org.ressources-electroniques.univ-lille.fr/10.1145/2786805.2786838. doi:10.1145/2786805.2786838.

[7] G. Darbord, A. Etien, N. Anquetil, B. Verhaeghe, M. Derras, A unit test metamodel for test generation, in: Proceedings of the 2023 International Workshop on Smalltalk Technologies, CEUR Workshop Proceedings, 2023. URL: https://hal.science/hal-04219649.

[8] M. Denker, Sub-method Structural and Behavioral Reflection, PhD thesis, University of Bern, 2008.

[9] S. Costiou, V. Aranega, M. Denker, Sub-method, partial behavioral reflection with reflectivity: Looking back on 10 years of use, The Art, Science, and Engineering of Programming 4 (2020). doi:10.22152/programming-journal.org/2020/4/5.

[10] G. Grano, S. Scalabrino, H. C. Gall, R. Oliveto, An empirical investigation on the readability of manual and generated test cases, in: Proceedings of the 26th Conference on Program Comprehension, ICPC '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 348–351. URL: https://doi.org/10.1145/3196321.3196363. doi:10.1145/3196321.3196363.

[11] A. Bergel, G. Galindo-Gutiérrez, A. Fernandez-Blanco, J.-P. Sandoval-Alcocer, SmallEvoTest: Genetically created unit tests for smalltalk, in: Proceedings of the International Workshop on Smalltalk Technologies, CEUR Workshop Proceedings, 2023.

[12] A. C. R. Paiva, A. Restivo, S. Almeida, Test case generation based on mutations over user execution traces, Software quality journal 28 (2020) 1173–1186.

[13] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, C. Xiao, The Daikon system for dynamic detection of likely invariants, Science of Computer Programming 69 (2007) 35–45. URL: https://www.sciencedirect.com/science/article/pii/S016764230700161X. doi:https://doi.org/10.1016/j.scico.2007.01.015, special issue on Experimental Software and Toolkits.

[14] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, W. Chen, CodeT: Code generation with generated tests, 2022. URL: https://arxiv.org/abs/2207.10397. doi:10.48550/ARXIV.2207.10397.