Toward Just-in-time and Language-agnostic Mutation Testing

Stefan Reschke
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
stefan.reschke@student.hpi.uni-potsdam.de

Toni Mattis
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

Fabio Niephaus
Oracle Labs
Potsdam, Germany
fabio.niephaus@oracle.com

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
robert.hirschfeld@hpi.uni-potsdam.de

ABSTRACT

Mutation Testing is a popular approach to determine the quality of a suite of unit tests. It is based on the idea that introducing faults into a system-under-test (SUT) should cause tests to fail, otherwise, the test suite might be of insufficient quality. In the language of mutation testing, such a fault is referred to as "mutation", and an instance of the SUT's code that contains the mutation is referred to as "mutant". Mutation testing is computationally expensive and time-consuming. Reasons for this include, for example, a high number of mutations to consider, interrelations between these mutations, and mutant-associated costs such as the cost of mutant creation or the cost of checking whether any tests fail in response. Furthermore, implementing a reliable tool for automatic mutation testing is a significant effort for any language. As a result, mutation testing is only available for some languages.

Present mutation tools often rely on modifying code or binary executables. We refer to this as "ahead-of-time" mutation testing. Oftentimes, they neither take dynamic information that is only available at run-time into account nor alter program behavior at run-time. However, mutating via the latter could save costs on mutant creation: If the corresponding module of code is compiled, only the mutated section of code needs to be recompiled. Additional run-time information (like previous execution results of the mutated section) selected by an initial test run, could also help to determine the utility of a mutant. Skipping mutants of low utility could have an impact on mutation testing efficiency. We propose to refer to this approach as *just-in-time mutation testing*.

In this paper, we provide a proof of concept for just-in-time and language-agnostic mutation testing. We present preliminary results of a feasibility study that explores the implementation of just-in-time mutation testing based on Truffle's instrumentation API. Based on these results, future research can evaluate the implications of just-in-time and language-agnostic mutation testing.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Runtime environments**; Software performance.

KEYWORDS

mutation testing, mutation coverage, GraalVM, Truffle, language-agnostic, polyglot

ACM Reference Format:
Stefan Reschke, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. 2022. Toward Just-in-time and Language-agnostic Mutation Testing. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Unit testing is a method often employed as the basis of software validation processes. A good suite of unit tests covers all features of a specific software component (as well as its non-functional requirements) and is typically fast to execute. This test suite enables stable and maintainable development of additional features in a module. Developers often determine the test suite's strength by measuring its coverage of application code. A most basic approach is to measure which lines of code the tests have been executing (*line coverage*). If the code is viewed as an *Abstract Syntax Tree* (AST), another approach would be to measure covered nodes of this AST (*node coverage*).

Among the more sophisticated coverage criteria, *mutation coverage* is a "high-end" coverage criterion [1]. It relies on the expectation that every change to the *tested* code should cause a test to fail. Every encountered violation (a change that caused no test to fail) is considered a defect of the test suite. The ratio of violations to all considered changes determines the suite's *mutation coverage*. The process of measuring *mutation coverage* is referred to as *mutation testing*.

Common mutation testing tools (or just *mutation tools*) such as MuJava [11] carry out the following operations: (1) determining viable changes to the tested code by a set of rules (*mutation operations*), (2) the creation of code-versions that each contains a specific change (*mutants*), (3) running all tests on these code-versions (check if mutant/mutation is *alive*). Following this approach, measuring mutation coverage on larger codebases is usually time-consuming and computationally expensive, even if the test suite itself runs fast. As a result, without applying more sophisticated techniques, mutation testing is (if at all) done infrequently. Enabling research addressing the scalability problem of mutation testing has been proposed repeatedly, e.g. in [3]. A central focus on research towards mutation testing has been in eliminating these problems.

Mutation testing has been researched since the 1970s [5, 8]. Statically-typed languages have been popular targets for mutation testing as valid mutations can be determined easier, but dynamic languages increasingly moved into focus [2], albeit with less information available to mutation tools. Nevertheless, mutation testing still largely follows a static approach by analyzing and mutating the program *before* compilation or execution, thus not leveraging valuable run-time information. We call this approach *ahead-of-time mutation testing*.

Papadakis et al. emphasize that mutants are inherently language-specific because the method of creating a specific mutant varies between the languages [13]. However, the way similar mutation operations are described in different languages is often the same. Some of these mutation operations can be easily expressed in different languages, for example deleting statements or exchanging conditional/math operators. The underlying concept of these operations might be language-agnostic. If they are, these mutation operation's implementations for different languages are partly redundant. Building a module of these mutation operation's concepts enables the reuse of these implementations among mutation tools of different languages.

Currently, approaches to measuring mutation coverage are often repeatedly evaluated in different languages. Each mutation tool is specifically designed to only generate mutants of a specific programming language or its intermediate representation (e.g. JVM-bytecode). However, language implementation frameworks such as Truffle [14], [15] enable language implementers to implement languages based on a common language-agnostic intermediate representation. In the case of Truffle, these representations are Truffle's ASTs. Truffle also supports developing language-agnostic tools through its instrumentation API [4]. These tools are implemented as what is referred to as Truffle *instruments*. In consequence, with the introduction of language implementation frameworks such as Truffle, an opportunity to evaluate whether mutation testing approaches can capture different languages within a language-agnostic mutation tool arises.

In this paper, we provide a proof of concept for just-in-time and language-agnostic mutation testing. We describe a mutation tool that uses Truffle's instrumentation API to mutate Truffle's AST just-in-time. Our initial findings indicate that Truffle enables just-in-time and language-agnostic mutation testing. Based on these findings, we describe research opportunities in section 5 that future research can address. We do not yet provide an evaluation of the concept of just-in-time and language-agnostic mutation testing.

2 APPROACH

Our objective is to provide a proof of concept for just-in-time and language-agnostic mutation testing. Thus, we present preliminary results of a feasibility study that explores the implementation of just-in-time mutation testing based on Truffle's instrumentation API. Based on these results, additional research can evaluate the implications of just-in-time and language-agnostic mutation testing.

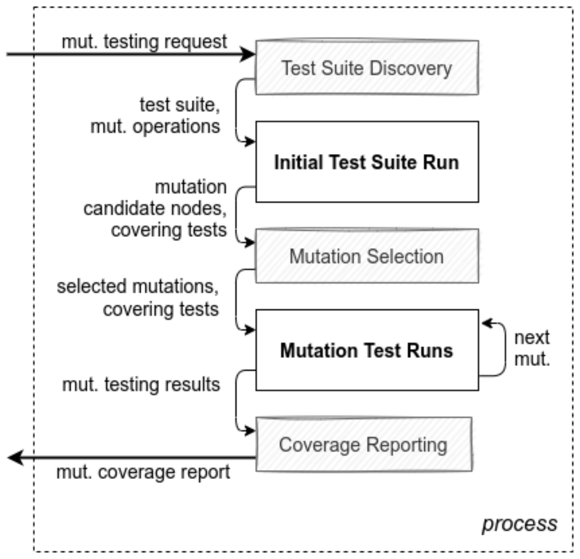

Figure 1: Algorithm for just-in-time mutation testing based on AST manipulations. The algorithm consists of five steps: 1) Setting up the mutation tool by collecting test suite and available mutation operations, 2) an initial run of the whole test suite to gather all mutation candidate nodes and the tests covering them, 3) selecting mutations to proceed with, 4) checking if these mutations are alive or not, and 5) building the mutation coverage report.

First, we considered adapting existing mutation tools to support using Truffle's instrumentation API. For this feasibility study, we wanted to avoid using an existing mutation testing API to reduce complexity. Therefore, we decided to develop a standalone mutation tool as a Truffle instrument. This mutation tool employs the following algorithm for just-in-time mutation testing: (Step 1) Discover the SUT's test suite, (Step 2) Perform an initial run of the entire test suite to determine all mutation candidate nodes (nodes that available mutation operations could be applied on) and the tests that cover them. Then, from these candidate nodes: (Step 3) Select mutations (node and mutation operation to apply on this node) to evaluate. This offers the possibility of discarding (Step 4) For each of these mutations, check if they are alive or not. This involves running the tests that cover the mutations node. Once the node is reached on test execution, the mutation operation is applied. Then, using the gained knowledge about the mutations: (Step 5) Provide a mutation coverage report that the user can interpret for his purposes.

We focused our efforts on steps 2 and 4 because they are most important for the feasibility study. For steps 1, 3, and 5, we made simplifying assumptions. Demonstrating the mutation testing instrument on an example project requires selecting a language to focus our support on. We decided to employ both *SimpleLanguage*[1] and *GraalJS*[2]. SimpleLanguage is a Truffle demonstration language implementation provided by Oracle. We chose SimpleLanguage

[1]https://github.com/graalvm/simplelanguage/
[2]https://github.com/oracle/graaljs

because it is the most compact Truffle language implementation and is also extensively documented. GraalJS is a Truffle language implementation of JavaScript and is also provided by Oracle. We chose GraalJS, because: (1) It is, compared with SimpleLanguage, a language that is used in real software projects, (2) it is the only fully supported Truffle language implementation by Oracle and thus, (3) promises to be more stable than e.g. GraalPython[3].

3 IMPLEMENTATION

In this section, we describe how we implemented mutation operations and each of the steps of the just-in-time mutation testing algorithm displayed in Figure 1. Here, we focus on (Step 2) *Initial Test Suite Run* and (Step 4) *Mutation Test Runs*. Before, we briefly describe the assumptions we made to reduce implementation effort for steps 1, 3, and 5. Afterward, we sum up the implementation effort that we require Truffle language implementers to do if they want their language to be supported by our mutation tool.

AST node manipulation mutation operations. Our feasibility study requires us to implement a few mutation operations. We chose to implement three simple mutation operations that mutate the result of AST nodes that are boolean expressions. These are (MO1) overriding the boolean-expression with *true*, (MO2) overriding the boolean–expression with *false*, and (MO3) negate the result of the boolean expression (true to false, false to true).

Working assumptions. For the proof of concept, we considered two approaches for (Step 1) Test Suite Discovery: (a) Assuming that top-level functions whose name starts with "test" and are within the SUT's source file, are tests and collecting handles to these functions with Truffle's instrumentation API, and (b) Implementing an adapter for a popular JavaScript test framework (e.g. Mocha[4]). For this proof of concept, we want to avoid dealing with interfaces of test frameworks for the sake of simplicity. Thus, we went with (a).

Since the initial test run provides all candidate nodes for available mutation operations, (Step 3) Mutation Selection gives an opportunity to filter these candidates. For our proof of concept, this is not required. Our mutation selection strategy is "pick them all".

Mutation testing without valuable output (Step 5) is not helpful. For the proof of concept, we decided that a simple console output suffices.

Initial test suite run. Discovering mutation candidates requires us to identify nodes that are interesting for our mutation operations. Truffle's instrumentation API allows us to attach listeners to node execution events [4]. These listeners can only listen for nodes that have a specific *tag*. In this context, a tag is a label of an AST node that is used to communicate the purpose of this node. All Truffle language implementations are supposed to provide an implementation of a set of standard tags. However, these standard tags do not provide enough information to easily select the nodes that are required by certain mutation operations. For example, (MO1) requires all nodes that are boolean-expressions. Standard tags, however, only communicate purpose on the level expression/statement, not the respective type returned by those. We decided to add an additional set of tags which we refer to as *mutation tags*. We assume that the languages implement these mutation tags just as they do with Truffle's standard tags. For the proof of concept, we changed both SimpleLanguage and GraalJS source code accordingly, overriding one method per node class. With these mutation tags, we were able to configure a listener to listen to only those node execution events that were of interest for our mutation operations.

With the handles discovered by the Test Suite Discovery step and the configured listener, we are able to perform the initial test suite run in order to discover mutation candidates and the tests that cover them. It also enabled us to collect run-time information from the context of the registered events. Most notably, this run-time information included the evaluation results of these nodes.

Mutation test runs. In order to apply a mutation just-in-time, we again listen for node execution events, just as during the initial test run. However, instead of collecting information about the node from the event, we alter the node's evaluation result. Optionally, the mutation operation could use the actual result the node would have returned. An example mutation operation that does this is (MO3), which requires the actual result of the node (true, false) to determine the correct negated value (false, true). However, if a mutation operation does not require an evaluation result, executing the children of the node can be avoided.

In fact, the capabilities of the node execution event listener best describe the interface that our tool enables language implementers to implement mutation operations in. Language-agnostic mutation operations such as (MO1), (MO1), and (MO3) require Truffle-language nodes to cooperate with the host-languages (Java) primitive types (in this case Booleans). However, language-specific mutation operations can be implemented with the same interface. These kinds of mutation operations might involve using a language concept that is only available in the language implementation itself (e.g. by upcasting a node). Therefore, we added this interface for operations to Truffle, which enables language implementers to supply their own, language-specific mutation operations.

For each of the selected mutations, determining if the mutation is alive or not involves running the tests that cover the mutated node until either a test failed or necessary tests have been run. During these test runs, the listener is configured to mutate only the requested node. For the next mutation, the listener is reconfigured.

Implementation effort for language implementers. There are three aspects language implementers need to attend to. These are displayed in Figure 2.

Aspect a) is necessary, because present language implementations only supply Truffle's standard AST nodes tags. These tags do not provide enough information for our mutation tool to easily determine the purpose of a node.

Furthermore, our mutation tool relies on the aspects b) and c) to be supplied by language-specific modules. Our assumption for Test Suite Discovery ("test"-prefix, same file) enabled us to easily implement this step. However, in order to support common test frameworks, we require a module that utilizes the framework to provide our mutation tool with a handle to all individual tests of a test suite implemented in this framework. If additional language-specific AST-based mutation operations should be available for a

[3]https://github.com/oracle/graalpython
[4]https://mochajs.org/

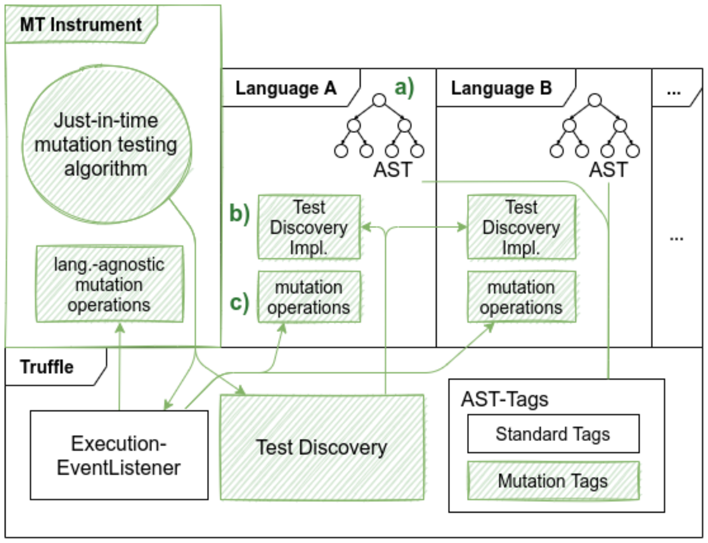

Figure 2: System overview: Just-in-time and language-agnostic mutation testing on Truffle. New modules are highlighted in green. There are three aspects language implementers need to attend to: a) Implement our mutation tags on Truffle language nodes, b) Contribute code to provide our tool with a projects test suite, and c) (if necessary) contribute code for language-specific Mutation Operations. The interface for these language-specific mutation operations is determined by the instrumentation API's ExecutionEventListener.

set of tags, we require a module that provides these mutation operations to our tool. As described, the interface for these operations is determined by the capabilities of the listener used to listen for node execution events.

4 PROOF OF CONCEPT

This section demonstrates an end-to-end mutation testing example that utilizes the presented mutation tool. Currently, our language-agnostic mutation operations and Test Suite Discovery support both SimpleLanguage and GraalJS. We chose to demonstrate a JavaScript example.

The example source file (example.js) displayed in Listing 1 contains both an example SUT and its tests. It contains a binary predicate *calc* and tests that verify that *calc* returns true for some example input values.

To run our mutation tool on this example, we only have to call any Truffle executable with the option --mt, and the source file whose test suite's mutation coverage should be determined. If our mutation tool is installed, this option instructs the executable to enable our mutation tool.

The test suite in Listing 1 covers all lines of the function under test. It also covers all branches, because both if-statements evaluate to true and false in the test suite.

Following the execution of the mutation testing run, our mutation tool displays a simple coverage report to the terminal (see Listing 2). Programmers can learn from this output which test requirements their tests have missed. In this case, a good first step to improve the quality of the test suite would be to write a test that specifies the condition under which the predicate can also evaluate to false.

Listing 1: Example JavaScript source file to demonstrate the difference between line-, branch- and mutation-coverage. It contains the binary predicate function "calc" and two tests. Note the two alternatives (b_i and b_j) that each depend on only one of two input-parameters

```
function calc(i, j) {
    if (i < 10) {  // b_i
        i = i + 100;
    }
    if (j < 100) {  // b_j
        j = j + 10;
    }
    return i < j;
}

console.log("GraalJS requires not-empty main to make our
    Test Suite Discovery work");

function test_lineCoverage() {
    if (!calc(5, 99)) {  // b_i: true, b_j: true
        throw Error();
    }
}

function test_additionalBranchCoverage() {
    if (!calc(11, 111)) {  // b_i: false, b_j: false
        throw Error();
    }
}
```

5 OPPORTUNITIES

This section gives an overview of what future research can focus on, based on the presented results. Our next steps will expand on the presented proof of concept. To enable a comparison with

Listing 2: Mutation Coverage Report of mutation testing run on Listing 1. Our three mutation operations were applicable on three nodes, resulting in nine available mutations. Of these nine mutations, only three mutations are alive. While this test suite had a line and branch coverage of 1, its mutation coverage is <1.

```
Mutation coverage score: 0.67
Tests run per mutation: 1.6
Total number of tests run: 14
Considered tests in test-suite: 2

alive mutations: 3
 on i < j @(example.js [8:12-8:16]) override with true
 on j < 100 @(example.js [5:9-5:15]) override with true
 on i < 10 @(example.js [2:9-2:14]) override with false
killed mutations: 6
 on i < j @(example.js [8:12-8:16]) override with false
    was killed by # test_additionalBranchCoverage
 on i < j @(example.js [8:12-8:16]) negate result
    was killed by # test_additionalBranchCoverage
 on j < 100 @(example.js [5:9-5:15]) override with false
    was killed by # test_lineCoverage
 on j < 100 @(example.js [5:9-5:15]) negate result
    was killed by # test_lineCoverage
 on i < 10 @(example.js [2:9-2:14]) override with true
    was killed by # test_additionalBranchCoverage
 on i < 10 @(example.js [2:9-2:14]) negate result
    was killed by # test_additionalBranchCoverage
```

other JavaScript mutation tools, we need to implement additional JavaScript mutation operations. We also need to add additional configuration options for our mutation tool, e.g. to enable or disable sets of mutation operations. We also need to make sure that our Test Suite Discovery for Mocha is fully functional, to easily compare mutation coverage results based on real-world example projects. With the presented mutation tool, we enable further research on the following three aspects:

Just-in-time mutation testing. Continue research on the implications of the just-in-time approach to mutation testing. (1) *Just-in–time mutation operations.* Discover which additional mutation operations can be implemented with this just-in-time mutation approach. We expect that some of these mutation operations could replace common operations known in ahead-of-time mutation testing. In order to measure if these operations could replace known operations, they need to be compared separately. We expect these operations to be able to replace each other if they enable creating the same mutants. Running mutation tools with them should result in equal mutation coverage reports. If they are qualitatively different from related known operations, these differences need to be discussed. (2) *Performance evaluation.* We expect that our approach leads to a reduced number of mutations considered. Because we rely on run-time AST node manipulation, we expect to have a reduced cost of mutant creation. We expect these reductions to reduce the end-to-end time of mutation testing runs. We plan to evaluate this with benchmarks, comparing our tool's end-to-end mutation testing time with other tools, making sure that employed mutation operations are comparable, as described above.

Mutation testing dynamic languages. Employing our just-in-time mutation approach, we hope to reduce the number of available mutation operations in dynamic languages. Just-in-time mutation testing enables using (previously not used) type information available at run-time and supports taking other specific run-time aspects into account. However, to evaluate our mutation tool to other mutation tools for JavaScript, we need to implement further mutation operations. A candidate mutation tool we would like to compare our mutation tool is *StrykerJS*[5], an open-source mutation tool for JavaScript. Future work can then deal with such a comparison with regard to supportable mutation operations and end-to-end mutation testing costs.

Language-agnostic framework for mutation testing. Our mutation tool relies on manipulating nodes of the language-agnostic AST employed by Truffle. This enables us to find out if these language-agnostic manipulations are available for different Truffle languages. Language-specific mutation operations form another category of supported mutation operations. If our mutation tool provides an interface that enables language authors to add language-specific mutation operations while re-using our mutation testing approach, we could reduce the cost of maintaining a mutation tool for these languages. Instead of these separate tools, more lightweight additional modules to our mutation tool could be maintained. A map of how different languages support simple mutation operations (based on the realizable AST node value manipulations) could be valuable for mutation testing research.

Future work. We hope to contribute the mutation tool to the Oracle Graal repository[6]. Through this, our mutation tags, interfaces for Test Suite Discovery, and implementing additional (language-specific or language-agnostic) mutation operations could become part of Truffle's API.

Future work can focus on enabling other languages like for example GraalPython or TruffleSqueak [12] to benefit from our mutation tool or expanding on the existing support for GraalJS. This involves changing these languages to implement our mutation tags. It also involves developing additional implementations for (Step 1) Test Suite Discovery to support more test frameworks of each of these languages. If required, language-specific mutation operations could also be contributed.

Future work on the mutation tool can also focus on the other steps of the applied just-in-time mutation testing algorithm that the presented feasibility study neglected (steps 3, 5, see Figure 1). The (Step 2) Initial Test Suite Run (see Figure 1) enables collecting run-time information for mutation candidate nodes, e.g. previous evaluation results of these nodes. Additional research can explore which additional run-time information from the context of accessed node execution events can be used to further specify mutations. An idea here would be the position of the node in its tree. Next steps can also explore how this run-time information can be used to further specify mutations. An idea here would be to enable a mutation operation on a node only on the node's n-th execution. We also see the chance to explore how this run-time information can be used to guide (Step 3) Mutation Selection, e.g. to assume

[5]https://github.com/stryker-mutator/stryker
[6]https://github.com/oracle/graal

the likely utility of a mutation. Next could also investigate how the intermediate results of the (Step 4) Mutation Test Runs provide additional information that can contribute to reducing or re-ordering the mutations selected in (Step 3).

GraalVM and Truffle's language-agnostic ASTs enable multiple languages to be used within a single application (polyglot programming). Future work can evaluate how our mutation tool could be used to perform mutation testing on polyglot code. We expect that language-agnostic mutation operations, as well as language-specific mutation operations of the employed languages, can be applied to evaluate the test suite of a polyglot application. Future research could also evaluate if there are additional, polyglot-specific mutation operations.

6 RELATED WORK

Papadakis et al. gave an invaluable overview of advances in mutation testing research [13]. Using the AST location of a mutation to determine if the mutant is useful, was proposed in [10]. Before our tool, other tools have employed just-in-time mutation testing:

SMutant, a mutation tool for the Smalltalk programming language was among the first tools to apply mutations at run-time during test execution [7]. Instead of relying on an AST representation of Smalltalk code, it directly manipulated the source code.

MutPy, a mutation tool for Python, employed an AST representation of Python code to increase the performance of its mutation testing capabilities [6].

7 CONCLUSION

In this paper, we provided proof of concept for just-in-time and language-agnostic mutation testing. The presented preliminary results of a feasibility study indicate that Truffle's instrumentation API supports implementing just-in-time mutation testing. We presented a mutation testing tool that is implemented as a standalone Truffle instrument. It employs a just-in-time mutation testing algorithm that relies on an initial test suite run to collect mutation candidate AST nodes, and then mutation test runs to check if these mutations are alive or not. A mutation is applied at run-time while the tests that cover the mutated node are running. Next steps include further development of this mutation tool to enable a comparison with established JavaScript mutation tools like StrykerJS [9].

Future research can evaluate the implications of this approach. This includes exploring mutation operations for just-in-time mutation testing and how they relate to classic ahead-of-time mutation operations. This also includes evaluating the performance of described mutation testing approach compared to ahead-of-time mutation testing.

Our mutation tool allows for previously unused run-time information (e.g. previous evaluation results of an AST node or the position of a node in its AST) to be employed in mutation testing efforts. We see the necessity to explore how useful this run-time information is. It could be used to specify when to apply a mutation operation during the test run (e.g. only on a node's n-th execution). However, it could also help to determine relations between mutants or determine a mutant's utility.

Because ahead-of-time mutation testing approaches have problems in dynamic languages, we see the chance to investigate if just-in-time mutation testing could be a modern standard for mutation testing in dynamic languages.

Because our approach relies on the manipulation of Truffle's ASTs, which are language-agnostic, it can be used to do mutation testing on multiple languages. Further research can explore if maintaining an adapter to our mutation tool by implementing a Truffle interface is easier than maintaining a standalone mutation tool for this language.

ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of Oracle Labs[7], HPI's Research School[8], and the Hasso Plattner Design Thinking Research Program[9].

REFERENCES

[1] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing.* Cambridge University Press.

[2] Leonardo Bottaci. 2010. Type sensitive application of mutation operators for dynamically typed programs. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops.* IEEE, 126–131.

[3] Thierry Titcheu Chekam. 2017. Automated and Scalable Mutation Testing. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST).* IEEE, 559–560.

[4] Michael L. Van de Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. 2018. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools. *Art Sci. Eng. Program.* 2, 3 (2018), 14. https://doi.org/10.22152/programming-journal.org/2018/2/14

[5] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.

[6] Anna Derezinska and Konrad Hałas. 2015. Improving mutation testing process of python programs. In *Software Engineering in Intelligent Systems.* Springer, 233–242.

[7] Milos Gligoric, Sandro Badame, and Ralph Johnson. 2011. SMutant: a tool for type-sensitive mutation testing in a dynamic language. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.* 424–427.

[8] Richard G. Hamlet. 1977. Testing programs with the aid of a compiler. *IEEE transactions on software engineering* 4 (1977), 279–290.

[9] Nico Jansen and Simon de Lang. 2022. *StrykerJS.* https://github.com/stryker-mutator/stryker

[10] René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring Mutant Utility from Program Context. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017).* Association for Computing Machinery, New York, NY, USA, 284–294. https://doi.org/10.1145/3092703.3092732

[11] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. 2006. MuJava: a mutation system for Java. In *Proceedings of the 28th international conference on Software engineering.* 827–830.

[12] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2019. GraalSqueak: Toward a Smalltalk-Based Tooling Platform for Polyglot Programming. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019).* Association for Computing Machinery, New York, NY, USA, 14–26. https://doi.org/10.1145/3357390.3361024

[13] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers.* Vol. 112. Elsevier, 275–378.

[14] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity.* 13–14.

[15] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming amp; Software* (Indianapolis, Indiana, USA) *(Onward! 2013).* Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[7] https://labs.oracle.com/
[8] https://hpi.de/en/research/research-school.html
[9] https://hpi.de/en/dtrp/