This is the peer reviewed version which has been published in final form at https://doi.org/10.1002/stvr.1838. This article may be used for non-commercial purposes in accordance with Wiley Terms and Conditions for Use of Self-Archived Versions.

SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2022; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr.1838

JUGE: An Infrastructure for Benchmarking Java Unit Test Generators

Xavier Devroey[1][6]*, Alessio Gambi[2][8], Juan Pablo Galeotti[3], René Just[4], Fitsum Kifetew[5], Annibale Panichella[6], and Sebastiano Panichella[7]

[1] *NADI, University of Namur, rue de Bruxelles 61, 5000 Namur, Belgium*
[2] *University of Passau, Innstrasse 41, 94032 Passau, Germany*
[3] *Universidad de Buenos Aires, Argentina*
[4] *University of Washington, Seattle, USA*
[5] *Fondazione Bruno Kessler, via Sommarive 18, 38123 Trento, Italy*
[6] *Delft University of Technology, Postbus 5, 2600 AA Delft, The Netherlands*
[7] *Zurich University of Applied Sciences, Steinberggasse 13, 8400 Winterthur, Switzerland*
[8] *IMC University of Applied Sciences, Am Campus Krems, Trakt G, 3500 Krems an der Donau, Austria*

SUMMARY

Researchers and practitioners have designed and implemented various automated test case generators to support effective software testing. Such generators exist for various languages (*e.g.,* Java, C#, or Python) and various platforms (*e.g.,* desktop, web, or mobile applications). The generators exhibit varying effectiveness and efficiency, depending on the testing goals they aim to satisfy (*e.g.,* unit-testing of libraries vs. system-testing of entire applications) and the underlying techniques they implement. In this context, practitioners need to be able to compare different generators to identify the most suited one for their requirements, while researchers seek to identify future research directions. This can be achieved by systematically executing large-scale evaluations of different generators. However, executing such empirical evaluations is not trivial and requires substantial effort to collect benchmarks, setup the evaluation infrastructure, and collect and analyze the results. In this software note paper, we present our *JUnit Generation Benchmarking Infrastructure* (JUGE) supporting generators (search-based, random-based, symbolic execution, *etc.*) seeking to automate the production of unit tests for various purposes (validation, regression testing, fault localization, *etc.*). The primary goal is to reduce the overall benchmarking effort, ease the comparison of several generators, and enhance the knowledge transfer between academia and industry by standardizing the evaluation and comparison process. Since 2013, several editions of a unit testing tool competition, co-located with the Search-Based Software Testing Workshop, have taken place where JUGE was used and updated. As a result, an increasing amount of tools (over ten) from academia and industry have been evaluated on JUGE, matured over the years, and allowed the identification of future research directions. Based on the experience gained from the competitions, we discuss the expected impact of JUGE in improving the knowledge transfer on tools and approaches for test generation between academia and industry. Indeed, the JUGE infrastructure demonstrated an implementation design that is flexible enough to enable the integration of additional unit test generation tools, which is practical for developers and allows researchers to experiment with new and advanced unit testing tools and approaches. Copyright © 2022 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: JUGE; unit test generation; evaluation infrastructure; benchmarking

*Correspondence to: Xavier Devroey, NADI, University of Namur, rue de Bruxelles 61, 5000 Namur, Belgium. E-mail: xavier.devroey@unamur.be

1. INTRODUCTION

Over the last decades, researchers have come up with various techniques to automate the generation of test cases. In particular, unit test generators seek to automate the production of tests for various purposes (*e.g.,* validation, regression testing, fault localization, *etc.*) using different techniques, including random search (*e.g.,* [?, ?]), search-based software testing (*e.g.,* [?, ?, ?]), and symbolic (*e.g.,* [?, ?]) and concolic execution (*e.g.,* [?, ?]).

Juristo *et al.* [?] identified three essential features each empirical evaluation should contribute to the software testing empirical body of knowledge. First, the evaluation should be fully defined, and the data should be analyzed with appropriate techniques to interpret the results. Second, the programs used for the evaluation and the setup and variables considered should represent the reality of the practice. Third, an evaluation should be replicable and come with a replication package to confirm previous results and reach an acceptable level of confidence in the hypothesis.

Similarly, to bridge the gap with industry, automated test case generators must come with solid evidence that the approach can also be applied in practice. For instance, evidence-based software engineering [?] can help practitioners make informed decisions about the choice of a generator based on the current best evidence from research. Those pieces of evidence come from empirical evaluations identifying the strengths and weaknesses of various generators.

In the case of unit test generators, conducting an empirical evaluation is not trivial. It requires an extensive manual effort to collect benchmarks (*i.e.,* Java classes for which to generate test cases), setup the evaluation and the evaluation infrastructure, collect and analyze the produced unit tests, and compare the results with the state-of-the-art. The primary goal of our JUnit Generation Benchmarking Infrastructure (JUGE) [?] is to reduce the overall effort and ease the comparison of several generators by standardizing the evaluation process. This standardization allows researchers to meet the requirements, enabling an effective contribution to the empirical body of knowledge in software testing.

JUGE is suited for evaluating and comparing *fully automated* black, white, and gray-box unit test generators. For instance, in previous editions of the tool competition, JUGE has been applied to evaluate various types of tools relying on a variety of approaches, including search-based [?, ?, ?], random-based [?, ?, ?], and symbolic execution [?, ?]. In a nutshell, the generator takes the source code or the binaries of a Java project as input and generates unit tests for a given class or set of classes. A time budget limits the generation, and the generated tests are compared *w.r.t.* their structural coverage and mutation score. The benchmarks, the tests, and the intermediate results can be saved and archived to be added to a replication package and enable future comparisons without re-executing all the generators, thanks to the standardized evaluation process implemented in JUGE.

JUGE has been initially developed in the context of the tool competition, co-located with the Search-Based Software Testing (SBST) workshop. It has been built to be *extensible* and configurable to be used with different test case generators and deployed in different environments. It relies on standardized processes, including an *adapter* mechanism to run a tool and process the generated tests. *Isolation* of the generator and test executions are handled through containerization (using Docker). Similarly, *scalability* of the evaluations relies on the parallelization of different containers, which allows relying on standard technologies (*e.g.,* Docker commands and dashboards) to handle the overall evaluation smoothly.

Since 2013, ten editions of the tool competition have taken place and used the JUGE infrastructure to evaluate and compare automated unit test generators [?, ?, ?, ?, ?, ?, ?, ?, ?, ?]. Consequently, JUGE has been improved and evolved over the years to integrate the latest advances from academia to enhance the comparison and best practices from industry to achieve high automation. Several tools have entered the competition [?, ?] and matured over the years by fixing bugs evidenced by the evaluations using the JUGE infrastructure, but also by confronting the various approaches to different benchmarks to discover areas for improvement and future research directions. The current implementation is openly available on GitHub[†] and on Zenodo for long-term storage [?].

The remainder of this paper is structured as follows: Section 2 discusses the background and related work of empirical evaluation and comparison of test case generators. Section 3 presents the challenges and requirements for building JUGE, as well as the design and implementation choices made to address those challenges. Section 4 provides guidelines for setting up an evaluation with JUGE (examples of evaluations setups using JUGE are discussed in Appendix A). Section 5 presents the impact of JUGE so far, while Section 6 discusses the lessons learnt from building JUGE and running evaluation with it, as well as the selection of suitable benchmarks (*i.e.,* classes under test), and the future work. Finally, Section 7 concludes the paper.

2. BACKGROUND AND RELATED WORK

When designing a new test case generation technique, conducting empirical evaluations is paramount to position this new technique in the current software testing body of knowledge [?, ?]. When the technique gains in maturity, developers will also rely on those empirical evaluations to make informed decisions about choosing a tool relevant to their industrial context [?, ?]. For instance, Melo *et al.* [?] designed a recommender for concurrent software testing techniques based on the characteristics of the software under test and the current body of knowledge in concurrent software testing.

2.1. *Empirical evaluation guidelines*

Over the years, several guidelines, benchmarks, and infrastructures have been developed to design, execute, and assess test case generators. For instance, Arcuri and Briand [?] defined guidelines for using statistical tools when evaluating and comparing randomized algorithms, which is the case with many automated test case generators. In their systematic review of the empirical evaluation of search-based test case generation, Ali *et al.* [?] identify the elements that should be reported in study designs. They found that search-based software testing has been focused on structural coverage and unit testing and that empirical studies should adopt a more rigorous and standardized execution and reporting approach. In particular, studies should account for random variation in the results using appropriate statistical hypothesis testing. They should then compare the techniques with other baselines to conclude that it brings any advantage.

More recently, in a significant effort to improve the review process in software engineering, Ralph *et al.* [?] defined *Empirical Standards* listing specific attributes expected when conducting empirical evaluation following a given research methodology. The empirical evaluation of automated test cases generation is classified under the umbrella of *Optimization Studies in Software Engineering*: *i.e., research studies that focus on the formulation of software engineering problems as search problems and apply optimization techniques to solve such problems* [?]. Among the essential characteristics of such studies, the standards require comparing the approach to an appropriate baseline and the distribution of the dataset (*i.e.,* the benchmarks used for the evaluation, if possible, and the results).

JUGE contributes to the general effort of improving the quality and reproducibility of empirical evaluations for unit test generators by (i) standardizing the evaluation process, using appropriate data analysis techniques, and (ii) enabling easy distribution of the benchmarks (*i.e.,* classes under tests used for the evaluation) and results, including the test cases, coverage, and mutation analysis, and statistical analysis for future comparisons and reproductions. Section 4 discusses the guidelines to design, execute, and report the results of an empirical evaluation using our infrastructure.

Similar efforts have been pursued in other areas of automated testing. Recently, FuzzBench [?], an open fuzzer benchmarking platform-as-a-service, has received much attention after its

[†]https://github.com/JUnitContest/JUGE

deployment, thanks to the support of Google. Similarly to JUGE, FuzzBench provides a standard procedure to benchmark fuzzer, using coverage and fault coverage on representative program datasets. It has been designed to be scalable and ensure fair resource allocation and platform independence. JUGE achieves the same goals by relying on containerization via Docker. It allows running the different generators in isolation while relying on standard technologies and tools (*e.g.,* Docker dashboards) to manage scalability, resource allocations, and independence from the underlying platform (more details about the infrastructure will be provided in Section 3).

2.2. *Comparison of test case generators*

Besides structural coverage, like lines or branch coverage, empirical evaluations rely on mutation analysis to compare different test case generators [?]. Mutation analysis [?] applies mutation operators, *e.g.,* replacing an arithmetic operator, to a program under test to produce faulty variants (*i.e.,* mutants) and executes a test suite on those variants. If a test fails on a particular mutant, this mutant is considered as *killed*. The mutation score, *i.e.,* the ratio of killed mutants to the total number of mutants, is used to measure the faults detection capabilities of the test suite [?].

For now, JUGE supports structural coverage and mutation analysis of the generated tests. Other kinds of automated analysis can be plugged into the infrastructure's extendable architecture. Additionally, all the generated test suites are saved using a unique identifier and can be collected for additional manual inspection.

2.3. *Benchmarks for software testing*

Empirical evaluations can be performed on various benchmarks (*i.e.,* classes under test). For instance, Fraser and Arcuri built SF110 [?], a corpus of 23,886 classes from 110 open-source projects used to evaluate and compare unit test generators. Other benchmarks follow a different approach by using actual bugs extracted from Java software systems. For instance, Defects4J [?] is a collection of reproducible bugs and a supporting infrastructure widely used for evaluating software testing and debugging approaches. In its latest version (v2.0.0), Defects4J contains 835 bugs from 17 Java software systems [?]. Similarly, BugSwarm [?] is a toolkit designed to mine reproducible failures and corresponding fixes to evaluate fault-detection, localization, and repair approaches.

JUGE supports the definition of customized benchmarks. For instance, previous editions of the tool competition have used classes from Defects4J's projects and classes collected from open-source projects. Section 6.2 discusses guidelines to select classes under test for an empirical evaluation based on our experience in the tool competition and related work.

3. JUGE INFRASTRUCTURE

We describe hereafter the challenges and requirements for building an automated infrastructure like JUGE. Based on those requirements, we present the architecture and implementation of JUGE.

3.1. *Challenges and requirements*

Building an evaluation infrastructure like JUGE poses several challenges. Indeed, to be reusable and have an impact on automated unit test generation, such infrastructure has to meet several requirements. We present and discuss the most important ones in the following paragraphs.

Extensible and configurable (C1). JUGE has to be extensible to adapt to different research contexts. For instance, JUGE has been used for the unit testing competition (described in Section 5) for which the overall execution time is limited. It should also be practical for large-scale evaluations running over several weeks. For this, adding new generators to the infrastructure should be easy to compare them to the state-of-the-art. Similarly, it should be easy to configure JUGE to use a given set of classes under test (*i.e.,* benchmarks) for an evaluation. Finally, it should be easy to extend

JUGE to use different measures on the generated tests and add new ones to adapt to the different research questions and hypotheses driving the empirical evaluation.

Isolation (C2). The execution of automated test case generators may have side effects on the rest of a system [**?**]. For instance, a generator using all available resources by default can negatively impact other generators running in parallel. Additionally, unexpected behavior, such as failures caused by faults in the generator, can cause damage to the system. Moreover, evaluating the generated tests requires executing them, which might also cause undesirable side effects. JUGE has to include mechanisms to isolate the execution of the generator and the generated test from the rest of the host system to avoid any undesirable side effects or damages.

Performance and scalability (C3). JUGE should be able to run on various platforms depending on the resources available. For instance, it should run on a standalone machine but also on a computationally intensive server or in a distributed setting on several servers in parallel. It should also scale to extensive empirical evaluations involving several tools and hundreds of classes under test.

Standardization (C4). Finally, evaluations relying on JUGE should be standardized to ease replication. This includes configuration of JUGE for a given evaluation, recording, analysis, and reporting of the data, and archiving to provide a companion artifact for the evaluation.

3.2. Implementation of JUGE

Given the identified requirements, we developed JUGE. JUGE is *extensible* and can be *configured* for evaluating and comparing *fully automated* black, white, and grey-box unit test generators (**C1**). The generator expects as input the source code or the binaries of a Java project and generates unit tests for a given class or set of classes. The generation is limited by a time budget provided as input to the generator. The overall execution is limited by a global timeout (*i.e.,* to twice the time budget) to take the pre and post-processing of the generator into account. For each benchmark (*i.e.,* class under test), JUGE runs the test case generator with the given time budget. JUGE can be configured to repeat the executions a given amount of times to balance *performance* and *scalability* (**C3**), depending on the available resources and the required statistical power of the results. Once the generation is completed, JUGE can measure structural coverage, perform mutation analysis of the generated tests, and compare the different generators using sound statistical analysis in a *standard* manner (**C4**). Moreover, the benchmarks, generated tests, and other data can be collected and stored in an artifact.

JUGE is open-source, available on GitHub[‡] and packaged as a Docker image to ensure *isolation* from the host system (**C2**). It contains scripts and tools supporting (i) the *generation of unit tests* for a given set of classes under test and time budget; (ii) the *coverage and mutation analysis* of the generated tests; and (iii) the *statistical analysis* and comparison of different unit test generators.

As illustrated in Figure 1, JUGE relies on an *adapter*, called `runtool`, to wrap specific calls to a unit test generator (MyTool in Figure 1). This adapter offers an interface to the `benchmarktool`, in charge of orchestrating the evaluation of the unit test generator. The communication between the host and the `JUnitcontest` container (*B* in Figure 1) is done via a common folder (*A* in Figure 1), mounted in the file tree structure of the image. This folder contains the executable binaries of the unit test generator and its `runtool` adapter. The generated tests, the metrics, and the statistical analysis results are saved in a subfolder (`results/`) to be made available to the host. The classes under test and the corresponding configuration file are saved in the Docker container (`benchmarks/`). Hence, to evaluate multiple tools, one can reuse the same container and only has to mount different folders, each containing the unit test generator and its `runtool` adapter.

[‡]`https://github.com/JUnitContest/JUGE`

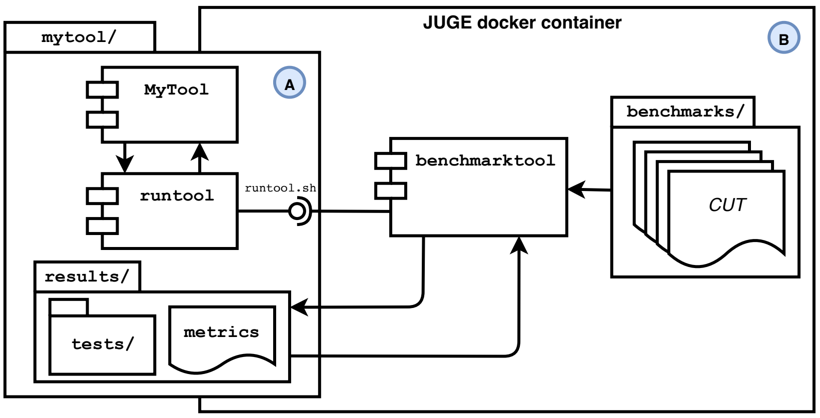

Figure 1. JUGE architecture overview

3.3. *Unit test generation*

One of the main challenges when building JUGE was to define a generic protocol for the generation of unit tests able to handle various unit test generators. We rely on a set of conventions and a generic communication protocol between the `benchmarktool` and the `runtool` adapter.

Conventions. By convention, the common folder (*A* in Figure 1) has to be named after the generator (`mytool/` in our example) and mounted in the `/home/` directory of the Docker container. For any generator, unit tests have to be generated in `/home/mytool/temp/testcases/`. Unit tests have to be stored as one or more Java test files containing JUnit tests for each class under test. Each Java test file has to declare a public class with a zero-argument public constructor, annotate test methods with `@Test`, and declare test methods public. Additional files may be saved to `/home/mytool/temp/data/` for later offline analysis (*e.g.,* for debugging of the generator).

Adapter. To make the liaison between the `benchmarktool` and the `runtool` adapter, the folder `/home/mytool/` must contain a `runtool` executable script or binary (*e.g.,* `runtool.sh` in Figure 1) that will be called by the `benchmarktool` to start the generation of unit tests. Typically, `runtool.sh` contains a single command launching the adapter. A customizable implementation of *runtool* is provided in the source code repository of our infrastructure.[§] The create an adapter for a new tool, one has to implement the different methods of the `ITestingTool.java` interface described in Listing 1.[¶] This effort should be minimal for any reasonably well-implemented tool. For instance, the implementation for RANDOOP is 102 lines long, against around 150 lines of code for EVOSUITE-based implementations. The methods are then called, as specified by the protocol described in the following paragraph.

Communication protocol. The adapter has to support the protocol described in Figure 2. In the first part (*A*), the `benchmarktool` signals the start of a new evaluation by sending the ’`BENCHMARK`’ message, followed by the paths to the source code and binaries of the software under test, the `CLASSPATH`, and the number of classes under test in the evaluation. Based on that information, the `runtool` adapter initializes the generator (in this case, MYTOOL).

After the initialization, the generator can signal that it will use additional `CLASSPATH` entries for its execution. The adapter notifies the `benchmarktool` of those additional entries (*B* in Figure 2). In the third part (*C*), the adapter notifies the `benchmarktool` that the generator is ready to start the

[§] `https://github.com/JUnitContest/JUGE/tree/master/runtool`
[¶] `https://github.com/JUnitContest/JUGE/blob/master/runtool/src/main/java/sbst/runtool/ITestingTool.java`

Listing 1: `ITestingTool` adapter interface

```
public interface ITestingTool {

    /**
     * List of additional class path entries required by a testing tool.
     * @return List of directories/jar files.
     */
    public List<File> getExtraClassPath();

    /**
     * Initialize the testing tool, with details about the code to be tested (
       SUT).
     * Called only once.
     * @param src       Directory containing source files of the SUT.
     * @param bin       Directory containing class files of the SUT.
     * @param classPath List of directories/jar files (dependencies of the SUT)
       .
     */
    public void initialize(File src, File bin, List<File> classPath);

    /**
     * Run the test tool, and let it generate test cases for a given class.
     * @param cName      Name of the class for which unit tests should be
       generated.
     * @param timeBudget How long the tool must run to test the class (in
       miliseconds).
     */
    public void run(String cName, long timeBudget);

}
```

evaluation by sending the `'READY'` message. The `benchmarktool` then sends the time budget allocated for the generation and the class under test of the first *run* to the adapter that, in its turn, calls the generator. After the generation, the adapter notifies the `benchmarktool` that the generator is ready for the next class under test.

3.4. Data collection

Once the test cases have been generated, JUGE can compute the different `metrics` for each test suite (*A* in Figure 1). Those metrics include: (i) the number of *flaky and non-compiling* tests, (ii) the *line and branch coverage*, and (iii) the *mutation score* of the generated tests. The JUGE infrastructure can be extended to support other kinds of metrics.

Flaky and non-compiling tests. First, if the test suite (one per Java file) does not compile, it is tagged and ignored in the subsequent steps of the analysis. Once compiled, the test suite is executed five times. Test methods (identified using the `@Test` annotation) producing different results between different executions are marked as flaky and ignored for the remainder of the analysis.

Line and branch coverage. JUGE relies on JACOCO [?] for statement and conditions coverage of the generated tests. Coverage information is furthermore used to reduce the subsequent mutation analysis time by restricting the execution of the tests against a given mutant to the tests effectively covering the lines modified by the mutant.

Mutation analysis. In the early versions of JUGE, we relied on PITEST [?] to generate and execute the mutants. However, it raised several issues for unit test generators relying on a dedicated test execution environment. For instance, test cases generated using EVOSUITE require running with

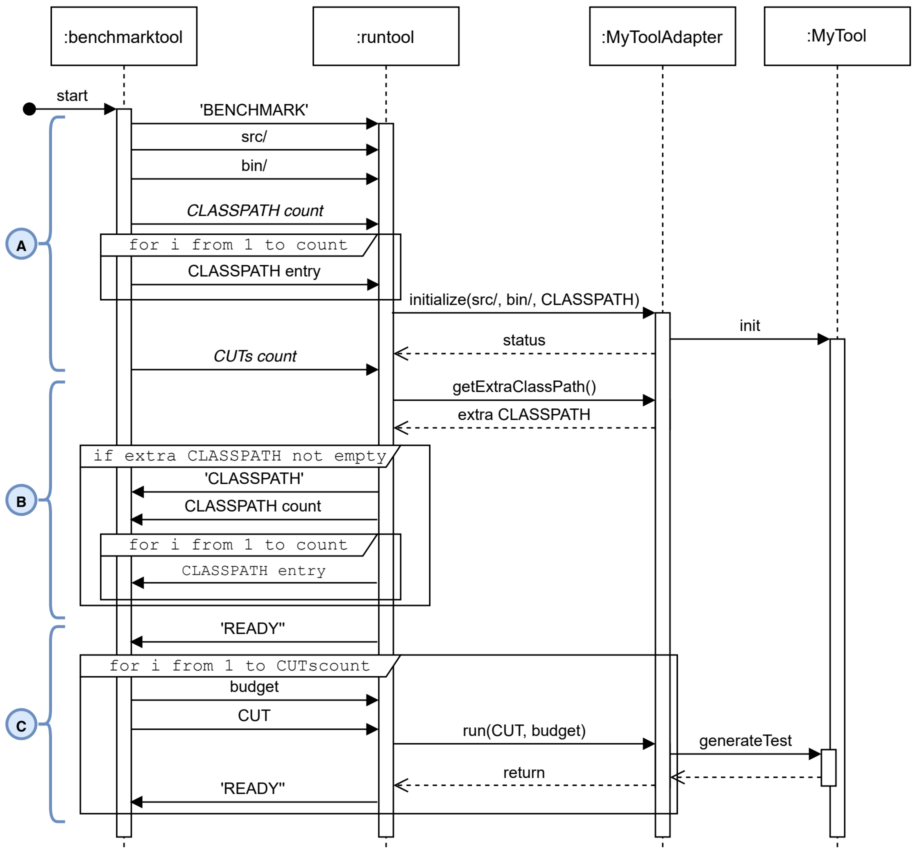

Figure 2. Communication protocol of the `runtool` adapter

a dedicated runner to avoid flakiness, handle inputs and outputs, *etc.*, preventing JUGE from using the PITEST environment for test execution. To solve this issue, we refined the mutation analysis to use the default test execution environment, supporting ad-hoc test runners. We use PITEST to generate the various mutants and the results of the line coverage to reduce the analysis time by executing only tests reaching the mutated lines against each mutant. Additionally, we set a hard deadline (5 minutes by default) for the mutation analysis to avoid infinite executions.

3.5. *Data analysis*

The generators can be compared based on quantitative analysis and the different measures collected during the analysis of the generated tests. For that, JUGE relies on Friedman's and post-hoc Conover's tests for multiple pairwise comparison [**?**] (available as an R script in JUGE). The former is a non-parametric test for significance, and it is widely used for multiple-problem analysis, where the problems correspond to the CUTs in our case. A significant p-value for this test indicates that the evaluated tools statistically differ *w.r.t.* to the overall performance score (*alternative hypothesis*). While Friedman's test does indicate whether the tools in the comparison are statistically different or not, it does not indicate for which pairs of tools such significance holds. Hence, the statistical analysis is complemented by using the post-hoc Conover's test for the pairwise comparison. Notice that the p-values produced by the post-hoc test are further adjusted with the Holm-Bonferroni procedure. This procedure corrects the statistical significance level (p-value=0.05) in case of multiple comparisons [**?**].

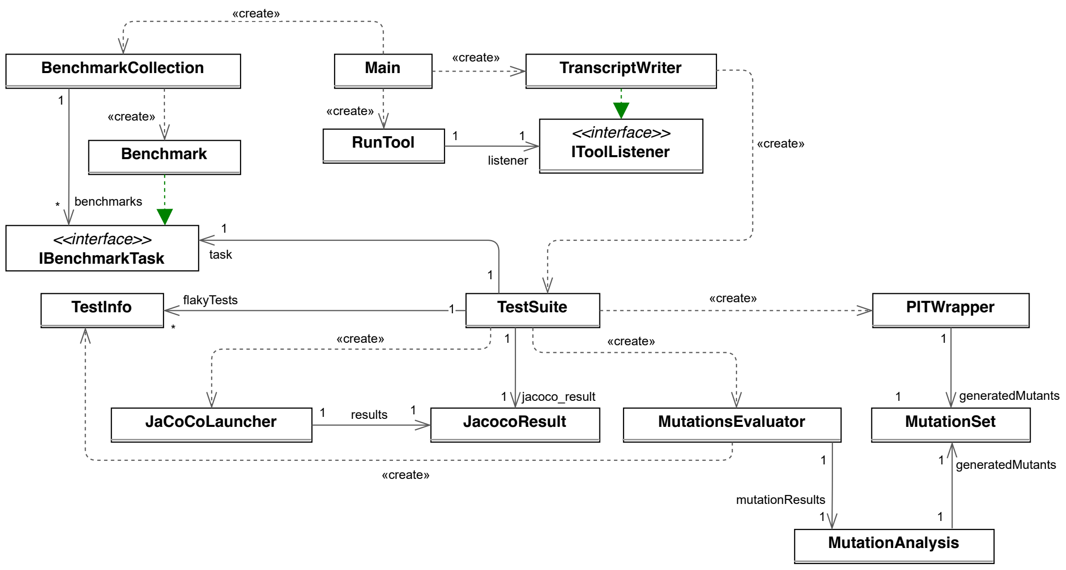

Figure 3. Internal architecture of the `benchmarktool` module

3.6. *Internal architecture*

The module `benchmarktool` (in Figure 1) is responsible for the orchestration and execution of the different steps of an evaluation. Figure 3 presents the main classes of that module. From a configuration file, the `Main` class loads the list of the benchmarks (*i.e.,* the classes under test) to use for the evaluation in a `BenchmarkCollection`. Each `Benchmark` object contains information about the class under test, classpath, and binary and source directory to use.

Once the benchmarks are loaded, the main class creates a `RunTool`, which is responsible for handling the execution and communication with the test case generator (the `sbst.benchmark.RunTool` class was renamed to `:benchmarktool` in Figure 2 to avoid confusion with the `:runtool` adapter module). This class opens an `SBSTChannel`, implemented as print and write streams on standard input and output, to communicate with the adapter (`:runtool` in Figure 2) of the generator following the protocol described in Section 3.3.

For each generated test suite, the `TranscriptWriter` class is responsible for collecting the different data (encapsulated in `TestInfo` objects) described in Section 3.4. The `JaCoCoLauncher` and `PITWrapper` encapsulate calls to (resp.) JACOCO, for code coverage information, and PITEST, for mutation analysis. As explained in 3.4, it was not possible to use PITEST to execute the full mutation analysis. The `PITWrapper` is therefore used to create the mutants, while a `MutationEvaluator` is in charge of executing the test suite on them and collecting the corresponding information.

4. SETTING UP AND RUNNING AN EVALUATION WITH JUGE

This section provides general guidelines for evaluating and comparing unit test generation tools with JUGE.

4.1. *Evaluation setup*

JUGE can be used to evaluate automated unit test generators that do not require human intervention during the generation process. It relies on the source code or binaries of a set of projects. JUGE is primarily designed to help carry out quantitative studies. It comes with support for structural coverage and mutation analysis and can be extended to support other measures.

Listing 2: Excerpt of a `benchmarks.list` configuration file

```
{
  BCEL-1= {
    src=/var/benchmarks/projects/bcel-6.0-src/src/main/java
    bin=/var/benchmarks/projects/bcel-6.0-src/target/classes
    classes=(org.apache.bcel.classfile.Utility)
    classpath=(/var/benchmarks/projects/bcel-6.0-src/target/classes)
  }
  BCEL-2= {
    [...]
  }
}
```

Nevertheless, JUGE also allows qualitative analysis as all the tests generated during the evaluation are saved and can be inspected or reused. Additionally, as explained in Section 3.3, JUGE allows the unit test generators to save any additional data for later analysis. For instance, a search-based unit test generator can save intermediate fitness values to analyze the fitness landscape evolution.

Generator meta-parameters. Many unit test generators can be configured through meta-parameters (*e.g.,* mutation and crossover probabilities for search-based approaches). To ease the evaluation and processing of the results, we recommend considering each configuration as an individual generator with its adapter in a dedicated folder (*A* in Figure 1) and an explicit name reflecting the configuration. Configuring the generators with the right parameters to answer the research questions and reporting those configurations in the empirical study is of paramount importance to reduce the threats to validity and enable the replicability of the results.

JUGE meta-parameters. The infrastructure has two meta-parameters: the *time budget* and the *number of repetitions*. The time budget corresponds to the budget allocated to generate a set of test cases for a given benchmark (*i.e.,* a class under test). JUGE also uses the time budget to set a global timeout for each execution equal to twice the time budget. The time budget depends mainly on the type of approach used by the test case generator. For instance, previous research indicates that a time budget of three minutes is suited for a search-based generator like EvoSuite [**?**, **?**] but is not enough for symbolic execution approaches like Tardis or Sushi [**?**].

Similarly, the number of repetitions varies if the generator relies on an exact approach or uses randomness. For exact approaches, one execution is enough (unless one of the research questions considers the execution time, in which case, several repetitions are necessary). For randomized approaches (*e.g.,* search-based and random approaches), several repetitions are necessary to ensure the statistical power of the results. Arcuri and Briand [**?**] estimated that the number of repetitions is a compromise between the number of benchmarks used in the evaluation, the execution time of the generators, and the overall budget available to perform the evaluation. They concluded that each randomized generator should be executed 1,000 times and, if it is not possible, report the reasons and the total execution time of the entire evaluation. However, the number of repetitions (for a more significant number of benchmarks) should be at least 10.

4.2. Benchmarks

The selection of the benchmarks (*i.e.,* sets of classes under test) should follow a systematic approach and ensure that the benchmarks are diverse enough to reduce the threats to the validity of the research questions [**?**]. For instance, by considering projects from different application domains. Those projects (and classes under test) can come from existing benchmarks: *e.g.,* Defects4J [**?**] or the previous editions of the tool competition relying on JUGE [**?**, **?**, **?**, **?**, **?**].

The benchmarks are described in a dedicated configuration file (`benchmarks.list`). Listing 2 provides an excerpt of `benchmarks.list` configuration file from the JUGE example benchmarks. Each benchmark has a unique identifier (line 2), the path to the root folder of the source files of the project (line 3), the path to the root folder containing the compiled classes (line 4), the list of classes under test (line 5), and the classpath with all the dependencies to use for the generation and coverage and mutation analysis (line 6). Once the benchmarks are defined, JUGE allows building a new Docker image (*B* in Figure 1) that can be instantiated multiple times in different containers to run the different tools.

4.3. Evaluation execution and results processing

Once the benchmarks and meta-parameters are defined, JUGE can start the evaluation by running different commands from the home directory of the tool in the Docker image (*e.g.,* `/home/mytool` in the example of Figure 1). We summarize hereafter the main steps and commands to use during the evaluation.[‖] If the available hardware allows it, it is possible to run several Docker containers in parallel (instantiated from the same JUGE Docker image), each responsible for executing a different generator. One should, however, be cautious to avoid overloading the machine as it could impact the execution of the generators and provoke timeouts. Different Docker containers should be run on independent machines with the same hardware configuration. Practically, if this is not possible, we strongly recommend doing some initial tests to determine the adequate number of parallel Docker containers to avoid undesirable side effects.

Unit test generation. JUGE allows running multiple *rounds* of the tool's execution on the same benchmarks and with the same budget in one command. Each execution's results are placed in a folder named after the tool and the time budget (*e.g.,* `/home/mytool/results_mytool_10` for a time budget of 10 seconds). For each benchmark and each round, JUGE creates a folder with the tests generated by the tool (*e.g.,* `BCEL-1_1`, `BCEL-2_1`, *etc.* for the first round of executions). Those different folders also contain text files with the logs and additional data produced by the generator.

Data collection and analysis. After the generation of the unit tests, JUGE can perform a coverage and mutation analysis using JaCoCo and PITest. The different results are stored in a dedicated CSV file (`transcript.csv`) for analysis. In our future work, we intend to extend JUGE to consider other types of measures, for instance, test case readability [**?**].

Once the different metrics have been computed for the different tools and budgets, the different results can be grouped in a single `results.csv` for a global quantitative analysis. JUGE can perform statistical analysis, as explained in Section 3.5, and produce a report with the results of the comparison.

4.4. Reporting, archiving, and reproducibility

One of the goals of the JUGE infrastructure is to enhance *repeatability* and *reproducibility* of both the results and statistical and qualitative analysis. For that, we strongly recommend submitting an *artifact* containing the following elements:

- the benchmarks, the generated tests, and additional data, if any,
- the files produced by the coverage and mutation, as well as any additional analysis,
- the results of the statistical analysis, together with any other data analysis scripts used for the evaluation.

[‖]Details on how to start the Docker container, and the different commands available in JUGE are available in the documentation at https://github.com/JUnitContest/JUGE/blob/master/docs/.

Suppose some of the benchmarks are under a non-disclosure agreement. In that case, we strongly recommend adding benchmarks from open source systems to the evaluation and releasing those in the artifact. The design of such an artifact must be considered early in the study. We recommend, for instance, to fork the JUGE repository and update the benchmarks configuration and files to generate a Docker image used to perform the evaluation. The fork can then be easily saved in a data repository (like Zenodo,[**] which has a GitHub integration) for long-term storage with a dedicated DOI.

In addition to the artifact, the reporting of the evaluation setup should mention the following elements:

- the randomized (or not) nature of the generators used in the evaluation;
- the meta-parameter configuration(s) of each generator;
- the meta-parameter configuration of JUGE (including the number of repetitions in the case of randomized generators) with a justification for those values;
- the total number of independent executions and the total execution time taken by the evaluation;
- the specifications of the hardware and the number of Docker containers running in parallel;
- the benchmarks selection procedure and the characteristics of the selected benchmarks relevant to the goals of the evaluation (*e.g.,* the number of lines of code of the projects and classes under test, the average McCabe's cyclomatic complexity of the benchmarks, *etc.*);
- any additional data collected and statistical analysis performed on the evaluation results with a proper justification (*e.g.,* see Arcuri and Briand [**?**] for a discussion on statistical analysis for randomized algorithms).

5. IMPACT OF JUGE

The JUGE infrastructure played a significant role in the *replication* of previous results regarding the structural coverage and mutation score achieved by automated unit test generators. The configurability of the infrastructure through the meta-parameters and the benchmarks considered for the various editions of the tool competition allowed us to assess the generated tests under various conditions. It independently confirmed that (i) search-based unit test generation (as implemented in EvoSuite) achieves a better coverage and mutation score [**?**, **?**, **?**, **?**, **?**, **?**]; and (ii) automatically generated tests can compete with manually written ones *w.r.t.* coverage and mutation score [**?**, **?**].

5.1. *Ten editions of the tool competition*

The JUGE infrastructure and the tool competition also helped to push the boundaries of unit test generation by confronting industrial generators to academic ones and showcasing how research can contribute to the industrial practices [**?**, **?**, **?**]. Also, selecting various benchmarks from open source systems helped to improve the academic generators by confronting them with new classes under test, thereby increasing the generalisability of the underlying approaches. For instance, EvoSuite has entered the competition multiple times with several algorithms (*whole suite approach* [**?**], MOSA [**?**], DynaMOSA [**?**], *etc.*) and in 2019, the results of the competition lead to the fix of a major bug [**?**]. The results of EvoSuite have also been recently independently confirmed using JUGE by Herlim *et al.* [**?**].

Table I describes the main characteristics of the different editions of the tool competition. Over the years, various tools have entered the competition and evolved. Among the different tools, Randoop

[**] https://zenodo.org

Table I. Editions of the tool competitions relying on the JUGE infrastructure with the generators, the time budgets (in seconds), the number of classes under test (#C), and the projects considered for the edition. Industrial tools are indicated by a start (⋆).

Edition	Generators	Budgets (in sec.)	#C	Projects
2013 [?]	RANDOOP, EVOSUITE [?], T2 [?]	-	77	Apache Commons Lang, Apache Lucene, Barbecue, Joda Time, sqlsheet
2014 [?]	RANDOOP, EVOSUITE [?], T3 [?]	-	63	Async Http Client, eclipse-cs, GData Java Client, Guava, Hibernate, JMLL, JWPL, Scribe, Twitter4j
2015 [?]	RANDOOP, EVOSUITE (whole-suite) [?], EVOSUITE (MOSA) [?], GRT [?], JTEXPERT [?], T3 [?], undisclosed Commercial Tool (CT)⋆	-	63	Async Http Client, eclipse-cs, GData Java Client, Guava, Hibernate, JMLL, JWPL, Scribe, Twitter4j
2016 [?]	RANDOOP, EVOSUITE (whole-suite) [?], JTEXPERT [?], T3 [?]	60, 120, 240, 480	68	DEFECTS4J [?]
2017 [?]	RANDOOP, EVOSUITE (whole-suite) [?], JTEXPERT [?]	10, 30, 60, 120, 240, 300, 480	69	Apache Commons BCEL, Imaging, and Jxpath, Freehep, Gson, Re2J, LA4J, Okhttp
2018 [?]	RANDOOP, EVOSUITE (whole-suite) [?], T3 [?]	10, 60, 120, 240	59	Dubbo, FastJason, JSoup, Okio, Redisson, Webmagic, Zxing
2019 [?]	RANDOOP, EVOSUITE (DYNAMOSA) [?], SUSHI [?], TARDIS [?], T3 [?]	10, 60, 120, 240	38	Antlr4, AuthzForce, Dubbo, Fescar, FastJason, Imixs-Workflow, Okio, Spoon, Webmagic, Zxing
2020 [?]	RANDOOP, EVOSUITE (DYNAMOSA) [?]	60, 180	70	Fescar/Seata, Guava, PdfBox, Spoon
2021 [?]	RANDOOP, EVOSUITE [?], EVOSUITE (dynamic symbolic execution) [?], KEX [?]⋆, UTBOT [?]⋆	30, 120	98	Seata, Guava, FastJSON, Spoon, Weka, Okio
2022 [?]	RANDOOP, EVOSUITE [?], BBC [?], KEX and KEX (reflection) [?]⋆, UTBOT and UTBOT (mocks) [?]⋆	30, 120	65	Seata, Guava, FastJSON, Spoon

is used as a baseline, and EVOSUITE has joined every year since the first edition. In 2015, 2021, and 2022, different industrial test case generation tools entered the competition.

The different editions have also tried different configurations *w.r.t.* to the execution of the tools and the time budget allocated for the generation. Before 2016, the time budget was left to the participants to decide (marked as - in Table I). Since 2016, the organizers have tried various time budgets to assess how the different tools react under a minimal budget: 10 seconds in 2017 and 2018 and 30 seconds in 2017 and 2021.

Similarly, the different editions have used classes under tests from various open-source projects to allow the distribution of the benchmarks after the competition. It allows one to replicate the results and the participants of the next edition to try their `runtool` adapter before submitting their tool to the competition. In 2016, the organizers used DEFECTS4J to generate regression tests and assess the tools' capability to expose real-world faults. Also, in 2019, 78 classes under test were initially selected. However, due to issues with metrics computation (that have been fixed), the number of classes considered for the final ranking dropped to 38.

Running the tool competition every year is not trivial. One of the main challenges the different organizers face is the hardware infrastructure required due to the limited time between the submission of the different tools and the limit for providing the results (around two weeks). Both the generation of the tests and their evaluation using coverage and mutation analysis are time-taking, requiring a powerful server or a cluster.

5.2. *Overview of the results of the competition*

As illustrated in Table I, several tools have entered the competition over the years. Figure 4 presents the averaged instruction and branch coverage, and averaged mutation score of the different tools

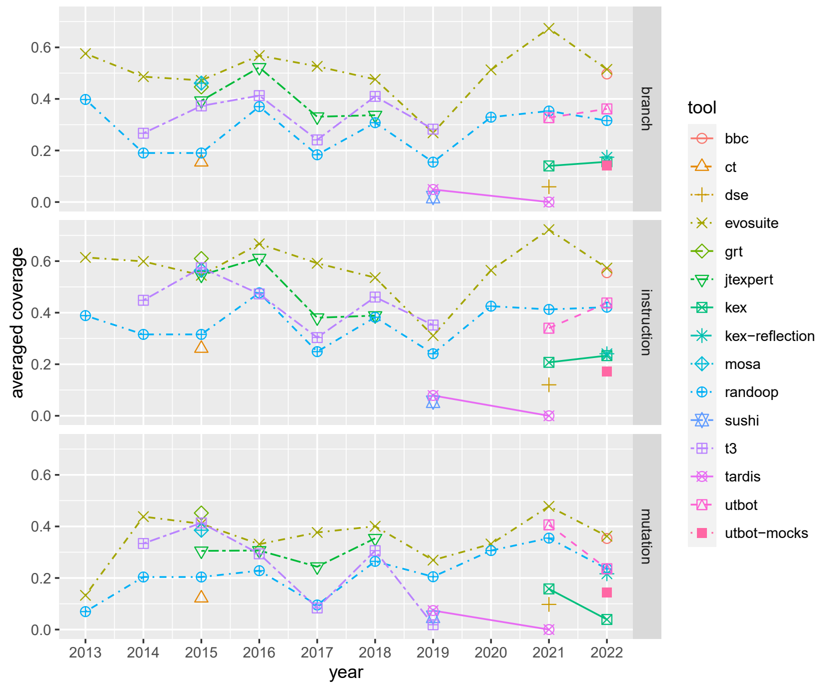

Figure 4. Averaged coverage evolution per tool over the years.

per year, collected from the reports of the past editions [?, ?]. As can be seen from the Figure, EvoSuite has the best averaged structural coverage and mutation scores over the years. The evolution from one edition to another can be attributed to several factors, including bug correction and improvements of the underlying algorithms, selection of different time budgets and benchmarks. The full dataset providing averaged coverage, mutation score, and execution times per benchmark and time budget is available on Zenodo [?].[††]

Among the different tools, Randoop, relying on feedback-directed random test generation [?], and EvoSuite, relying on genetic algorithms [?], have entered the competition every year since 2013. Figure 5 presents the distributions of the averaged coverage evolution per benchmark for each configuration of EvoSuite and Randoop for the different editions of the competition (*i.e.,* each data point represents an averaged value of all the executions of the tool with one time budget on one benchmark). Generally, EvoSuite performs better on individual benchmarks than Randoop for unit test generation. This is confirmed when applying Friedman's non-parametric with post-hoc Nemenyi test [?, ?] on instructions coverage (Randoop ranking 1.77 and EvoSuite 1.23), branch coverage (Randoop ranking 1.77 and EvoSuite 1.23), and mutation score (Randoop ranking 1.73 and EvoSuite 1.27). The rankings are significantly different as their corresponding average ranks differ by at least the given critical distance, here, 0.048, with a Friedman's test p-value below 0.01 for the three cases.

6. DISCUSSION AND LESSONS LEARNED

Any empirical evaluation of automated unit test generation faces several technical and methodological challenges. JUGE seeks to alleviate those challenges by providing a standardized way of designing, running, and reporting such evaluations. Both the development of JUGE and the

[††]Due to incomplete data, the results of T2 from 2013 [?] are not included in Figure 4.

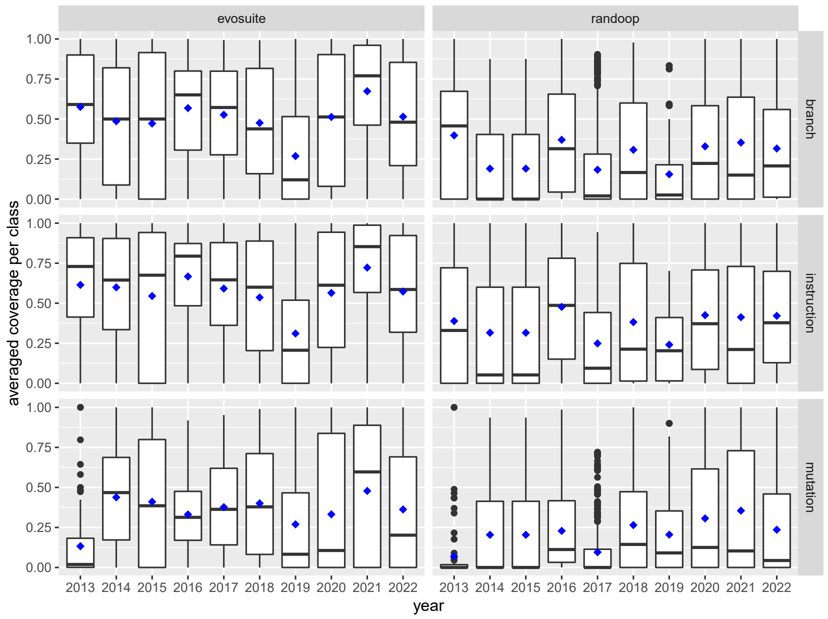

Figure 5. Averaged coverage evolution per benchmark for EvoSuite and Randoop. Diamond (♦) indicates the mean value.

evaluation method reported in Section 4 took several years to develop. We discuss hereafter the main lessons learned and potential new applications of JUGE.

6.1. *Building* JUGE *and evaluating generators*

Adapt to different generators. The main technical challenges for such an infrastructure come from the diversity of the generators that can be considered for an empirical evaluation (*i.e.,* random-based, search-based, concolic/symbolic-based, *etc.*)(**C1**). It requires *isolating* (**C2**) the executions to avoid troubles in case of a bug in the generator (*e.g.,* erasing files from the host file system [**?**]) while still having a *standard communication interface* (**C4**). It is achieved through an adapter with a shared standard set of commands used by JUGE to interact with the generator. JUGE runs in a Docker container to guarantee the isolation of the generator from its environment during test case generation.

Performance, scalability, and statistical power. As for any empirical evaluation with a random-based generator, researchers have to balance the number of classes under test to consider reducing external validity with the number of executions to ensure enough statistical power, giving the external constraints on the overall execution time [**?**]. For instance, in the tool competition, the entire evaluation must be done in around two weeks. To cope with this limitation, organizers use sampling to select a subset of classes under test, limit the time budget (not more than 8 minutes), and the number of repetitions of the executions (between 6 and 10, depending on the year). As explained in Section 4.1, the time budget allocated to the generator depends on the type of approach and the goals of the evaluation.

JUGE supports parallel executions by running several Docker containers in parallel (the number of containers depends on the technical specifications of the host machine), allowing to *scale* while preserving the overall *performances* of an evaluation (**C3**). It allows relying on standard Docker technologies and tools (*e.g.,* dashboards) to handle the overall evaluation.

Configuration of the meta-parameters. In addition to the time budget and the number of repetitions, which can be configured for JUGE, the generators usually come with various meta-parameters that directly influence the generation process (**C1**). As explained in Section 4.1, such parameters should be carefully considered and reported to reduce the threats to the validity and enable the *replicability* of the results.

For instance, many test generators like EvoSuite and Randoop include post-processing mechanisms that can be activated to minimize the generated tests [**?**, **?**]. Such mechanisms are time-consuming and can be deactivated to reduce the overall execution time when evaluating properties such as coverage or the mutation score. However, deactivating test case minimization has a significant impact on other properties, such as structural properties, readability, the execution time of the tests, *etc.* Researchers should be aware of such impacts and carefully consider them when designing their studies.

Analysis of the generated tests. Automated test case generation is a challenging task and requires several mechanisms (*e.g.,* code instrumentation, handling I/O operations on the system under test, *etc.*) to be effective. Among the possible mechanisms is using a specific scaffolding for the generated tests: for instance, EvoSuite controls elements that could be non-deterministic to avoid test flakiness. However, such mechanisms might cause undesired interactions with the infrastructure and, more specifically, with the tools used to analyze the generated tests (**C1**). It has been the case for EvoSuite and the mutation analysis: the test runner (`EvoRunner`) used in the generated tests was not compatible with PITest and required to use the mutated `.class` files directly instead of relying on the optimized PITest infrastructure. In the latest version, JUGE includes options to parallelize the execution of the mutation analysis and reduce the overall execution time of the evaluation (**C3**).

Repeatability and reproducibility. JUGE considers each generator configuration (*e.g.,* EvoSuite using a different generation algorithm) as a generator that will require its `runtool` adapter and corresponding shared folder, like, for instance, the different algorithms used with EvoSuite in the competition. Those generators should be shared in a companion artifact to provide *standard* readily usable implementations to the research community (**C4**). As one of the main goals of JUGE is to provide a common platform where researchers and practitioners can plug their generators and compare them using various benchmarks, sharing generators and their corresponding adapters will greatly improve *repeatability* (*i.e.,* the same evaluation can be performed by the same team in the same conditions and produce the same results) and *reproducibility* (*i.e.,* the same evaluation can be performed by a different same team in similar conditions and produce the same results) of the results.

6.2. *Benchmarks selection*

As explained in Section 4.2, JUGE allows one to define their own set of benchmarks for a given evaluation. Section A.1 provides an example of benchmarks selection process, followed in the eighth edition of the tool competition [**?**]. As illustrated by the example, the competition mainly used an opportunistic approach by considering open-source projects built with Maven to ease the collection of projects' dependencies. The main selection criteria were availability and the complexity of the classes to test. Other criteria can also be taken into account.

In general, selecting representative benchmarks for an evaluation is not an easy task. The selection criteria depend on the goals of the evaluation but also the availability of the benchmarks. We provide some guidelines for benchmark selection based on our experience and related literature.

Representative. A good set of benchmarks should be representative of real-world software. This is a best practice adopted by the software engineering research community to ensure that the evaluation addresses a relevant problem [**?**, **?**, **?**, **?**, **?**]. The benchmarks can come from one or more systems depending on the experimental design. For instance, Almasi *et al.* [**?**] performed an evaluation of unit test generation on a closed-source industrial *case study*. Other evaluations studied unit test generation on various open-source (and openly accessible) systems [?, ?, ?, ?]. In this latter case, one should take care of selecting systems that are popular (*e.g.,* by looking at the number of stars on GitHub or the number of dependant projects in Maven central) and under active maintenance (*e.g.,* by looking at the commit frequency) to ensure that the results of the evaluation will be relevant to the software engineering industrial community.

Diverse. The benchmarks used for an evaluation should be diverse enough to mitigate threats to external validity. This diversity can, for instance, be achieved by considering benchmarks from multiple projects from diverse application domains. Another approach could be, instead of a random sampling like in Section A.1, selecting diverse benchmarks based on the coverage and mutation score a random generator achieves.

Adequate. The benchmarks should be selected adequately *w.r.t.* the goals of the evaluation. For instance, if one of the evaluation goals is to compare the coverage and mutation score of automatically generated and hand-written tests, one has to consider the coverage and mutation score of the existing test suites when selecting the benchmarks. *E.g.,* in 2019, the competition [?] introduced Spoon in the benchmarks as it has a hand-written test suite with high coverage. The results showed that none of the generators used in that edition could achieve a higher branch or line coverage, or mutation score.

Challenging. Depending on the kind of generation technique considered, some benchmarks might not be relevant or do not bring any interesting insights. For instance, Shamshiri *et al.* [?] showed that many classes in the SF110 corpus [?], a corpus of benchmarks widely used for unit test generation, can be covered using random search. Those benchmarks should be filtered out for evaluating search-based unit test generators as they will not provide interesting insights. For instance, the competition and other related work [?, ?] have considered McCabe's cyclomatic complexity (*i.e.,* the number of independent paths in a control flow graph) of the benchmarks to filter out simple benchmarks, easily covered (*i.e.,* classes with a cyclomatic complexity lower than five).

Comparable. Generalizability is hard to achieve and requires several evaluations, preferably performed independently by different research teams. Selecting a subset of benchmarks that have already been used in other studies is interesting to allow comparison between different studies. It enables the meta-analysis of the results and cross-comparisons between different studies. For instance, Campos *et al.* [?], and Panichella *et al.* [?] reused the same benchmarks to perform large-scale empirical evaluations of search-based unit test generation algorithms, allowing a direct comparison and discussion of the results.

Documented. As stated in Section 4.4, both the selection procedure and the characteristics of the benchmarks should be reported together with the evaluation results. It is essential to the reviewing process to ensure that reviewers have enough information about the benchmarks to assess the paper describing the evaluation [?], but also to ensure that future research can compare and replicate the results on other benchmarks with similar characteristics.

Available. Finally, benchmarks coming from open-source projects should be made available to the research community. Building a good benchmark is not trivial and represents a substantial effort [?]. The community should share this effort by encouraging the best open-science practices and distribution of the benchmarks. JUGE provides a standard way of sharing and reusing such benchmarks (described in Section 4.2). [‡‡]

[‡‡]Benchmarks of previous tool competitions are available at `https://github.com/JUnitContest/JUGE/tree/master/infrastructure`.

6.3. Future applications

The method described in Section 4 constitutes a standard that can be applied to unit test generation for other kinds of languages using an infrastructure similar to JUGE. For instance, Lukasczyk *et al.* [?] recently defined an approach to generate unit tests for Python. Of course, dynamically typed languages such as Python face additional challenges than those discussed in Section 3.1. Such challenges must be considered in the design of the infrastructure (*e.g.,* running type inference engines during pre-processing) and the selection of the benchmark (*e.g.,* considering classes with type annotations only, *etc.*), and reported in the description of the empirical evaluation.

Besides comparing unit test generators, the JUGE infrastructure can be used to generate large amounts of tests for various kinds of classes using different tools and configurations. It enables the continuous creation of an openly available corpus of automatically generated unit tests. Such a corpus would (i) directly contribute to the body of empirical evidence on which decision-makers can rely to assess the usage of a unit test generator in their industrial context [?]; and (ii) enables further empirical evaluations on automatically generated tests without configuring and running the generators, which require a certain level of expertise. For instance, in a recent study, Panichella *et al.* [?] revisited previous studies on the presence of test smells in automatically generated tests and found that previous results vastly overestimated their presence. Among the different problems, they pointed out a misconfiguration of EvoSuite and its minimization process, resulting in more prominent test cases more likely to contain certain smells. Building openly available corpora using the appropriate configuration for the generators, with a description of the characteristics applicable evaluations, would avoid such issues. More in general, having the JUGE infrastructure and its associated standards available to industrial and academic research communities can open the road for more systematic testing for other languages (e.g., Python, etc.) as well as the definition of testing pipelines to identify bugs and imperfections of systems in other application domains [?, ?, ?, ?]. The availability of such technologies can also impact computer science education, with available tools that can be used in practical courses. Finally, we also expect future investigation supported by JUGE in the context of test code quality and cost-effectiveness of both automatically and manually generated tests [?, ?].

Finally, regarding the infrastructure itself, we plan to add other types of analysis in addition to coverage and mutation score. Such analyses include performances of the generated tests (*e.g.,* execution time and memory consumption) and readability.

7. CONCLUSION

JUGE sets a standard for properly assessing automated test case generators. It provides an infrastructure and a method to design, set up, and execute an empirical evaluation, collect and analyze the results, and produce a replication package to meet the requirements, enabling an effective contribution to the software testing empirical body of knowledge. It includes recommendations for selecting benchmarks and the parametrization of the generator and the infrastructure, depending on the research questions. JUGE was initially introduced and developed in the context of the tool competition and has been used with several generators and dozens of classes under test from various projects.

Finally, the JUGE infrastructure availability opens several directions for practitioners, who can rely on a large body of empirical evidence to assess automated test case generation usage in their context, and researchers, who can benefit from corpora of automatically generated tests for further empirical evaluations. JUGE also provides guidelines for evaluating unit test generation in other programming languages and for defining similar infrastructures in other domains.

ACKNOWLEDGEMENT

We would like to thank (in alphabetical order) Arthur Baars, Sebastian Bauersfeld, Matteo Biagiola, Ignacio Lebrero, Urko Rueda Molina, and Fiorella Zampetti for their contribution to the implementation of the

JUGE infrastructure. We would also like to thank (in alphabetical order) Azat Abdullin, Marat Akhin, Giuliano Antoniol, Andrea Arcuri, Cyrille Artho, Mikhail Belyaev, Pietro Braione, Nikolay Bukharev, José Campos, Nelly Condori, Christoph Csallner, Giovanni Denaro, Gordon Fraser, Yann-Gaël Guéhéneuc, Masami Hagiya, Mainul Islam, Dmitry Ivanov, Gunel Jahangirova, Kiran Lakhotia, Ignacio Manuel Lebrero Rial, Lei Ma, Alexey Menshutin, Arsen Nagdalian, Gilles Pesant, Simon Poulding, Wishnu Prasetya, Vincenzo Riccio, José Miguel Rojas, Abdelilah Sakti, Hiroyuki Sato, Sebastian Schweikl, Gleb Stromov, Yoshinori Tanabe, Paolo Tonella, Artem Ustinov, Sebastian Vogl, Tanja Vos, Mitsuharu Yamamoto, Fiorella Zampetti, and Cheng Zhang for their participation in previous editions of the competition and the feedback they provided on the infrastructure. Xavier Devroey was partially funded by the EU Horizon 2020 ICT-10-2016-RIA "STAMP" project (No.731529), the Vici "TestShift" project (No. VI.C.182.032) from the Dutch Science Foundation NWO, and the CyberExcellence (No. 2110186) project, funded by the Public Service of Wallonia (SPW Recherche). Alessio Gambi's work was partially supported by the DFG project STUNT (DFG Grant Agreement n. FR 2955/4-1). Sebastiano Panichella and Annibale Panichella gratefully acknowledge the Horizon 2020 (EU Commission) support for the project *COSMOS* (DevOps for Complex Cyber-physical Systems), Project No. 957254-COSMOS. René Just's work is partially supported by the National Science Foundation under grant CNS-1823172.

A. EXAMPLE OF EVALUATION USING JUGE

This appendix illustrates how JUGE can be used in practice. We use as an example the eighth edition of the Java unit testing tool competition [?] in which three of the authors were involved. The competition has occurred since 2013 and is co-located with the Search-Based Software Testing workshop. Participants willing to enter the competition have to provide (i) an executable version of their tool (potentially obfuscated, which enables the participation of industrial tools); and (ii) an implementation of the `runtool` adapter (described in Section 3.3) for their tool. In practice, this implementation consists in cloning the `runtool` project available in the JUGE repository and defining the methods of the `ITestingTool` interface described in Listing 1. The effort is minimal and consists of writing between 100 and 200 lines of code for a reasonably well-implemented tool (*i.e.,* tools relying on configuration parameters, for instance).

The competition aims to compare different tools on a diverse set of classes under test (*i.e.,* benchmarks) for different time budgets. As the organizers run the evaluation, the time budgets and number of executions depend on the computational resources available. For the eighth edition, the tools were executed ten times for 60 and 180 seconds on each benchmark.

A.1. Benchmarks selection

Several ways exist to define a new set of benchmarks for a given set of projects. The projects selected for that edition were a mix of projects from the previous iterations of the competition to allow comparing results across years and popular GitHub projects built using Maven to ease dependency collection. The benchmarks were selected following a two-step procedure for the list of selected projects. In the first step, (i) identifying the packages in the project that contain classes relevant for the evaluation (*e.g.,* packages containing classes with the business logic); (ii) computing the McCabe's cyclomatic complexity for the different classes of those packages and remove classes with a complexity lower than five. This reduces the risk to sample classes with few branches, easily covered by randomly generated tests [?].

In the second step, a random generator (here, RANDOOP [?]) was executed with a low time budget (*e.g.,* ten seconds) on the remaining classes to filter out classes for which the generator could not generate any tests. This reduces the chances of facing technical difficulties while evaluating the different tools. Since the remaining classes were still too high, a subset of classes was randomly sampled for each project.

In addition to the previous steps, one can also use JUGE to perform a coverage and mutation analysis of the tests produced by the random generator and report the results for the candidate and sampled classes. Figure 6 shows the line and branch coverage and mutation score of the candidate and sampled classes of the eighth edition of the tool competition.

A.2. Execution

The eighth edition of the competition received only one submission (*i.e.,* EVOSUITE with DYNAMOSA) that was compared against RANDOOP, used as a baseline. The implementation of the `runtool` adapter is available in JUGE's GitHub repository. EVOSUITE and RANDOOP were executed ten times each against each benchmark. The executions were run in parallel (using Docker) on two servers: one with 40 CPU cores (2.30GHz) with 128 GB memory and one with 8 CPU cores (2.49GHz) with 160 GB memory. The coverage and mutation analysis were performed on the same machines. The total execution time for test generation and analysis took around four days.

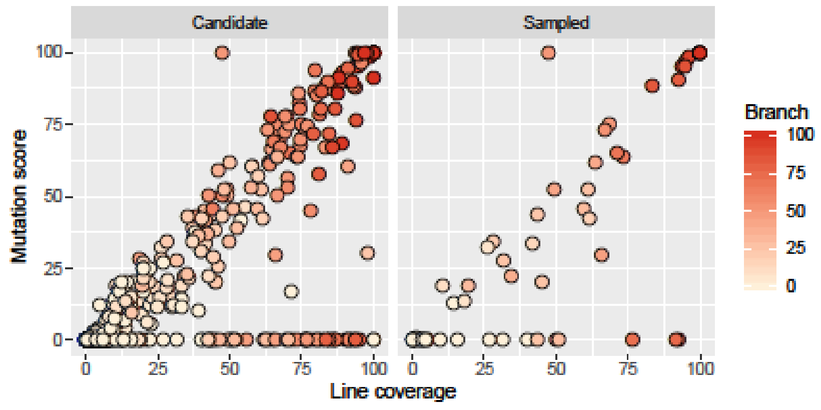

Figure 6. Example of reporting of the line coverage, branch coverage and mutation score for the candidate (382 classes) and selected benchmarks (60 classes, 20 per project) from the 2020 tool competition [?].

Table II. Example of scores and ranking obtained through Friedman's test for the 5th edition of the tool competition [?].

Tool	Score	Std.dev	Ranking
EVOSUITE	1457	192.72	1.55
JTEXPERT	849	102.03	2.71
T3	526	82.43	2.81
RANDOOP	448	34.75	2.92

Table III. Example of results of the post-hoc Conover's test for the 5th edition of the tool competition [?].

	EVOSUITE	JTEXPERT	T3	RANDOOP
EVOSUITE	-	-	-	-
JTEXPERT	< 0.01	-	-	-
T3	< 0.01	0.01	-	-
RANDOOP	< 0.01	< 0.01	0.06	-

A.3. *Data analysis and ranking of the contestants*

The competition combines the different metrics to ease the comparison of different generators. For that, it relies on a *scoring formula* [?]. This formula has been developed and refined during the different editions of the competition and takes into account the line and branch coverage, the mutation score, and the time budget used by the generator, and applies a penalty for flaky and non-compiling tests. In 2020, EVOSUITE ranked first. As a further example, Table II provides the ranking obtained through Friedman's test for the fifth edition of the tool competition [?]. EVOSUITE is ranked first with an average score of 1457, followed by JTEXPERT, T3, and RANDOOP. Table III gives the post-hoc Conover's test results for the same edition of the competition and indicates that the various comparisons are statistically significant, except for T3 and RANDOOP for which the *p*-value is above the confidence level of 0.05.

The scoring formula used in the competition provides an aggregated measure to rank the different tools based on their coverage and mutation analysis performances. Other aspects could be considered, like, for instance, the readability of the generated tests. Considering such aspects and including them in the scoring formula is part of the future work identified in the tenth edition of the competition [?].

Once the coverage and mutation analysis have been performed, the full results for a tool (*i.e.,* benchmarks, generated tests, and collected data) are sent to the tool's authors for further analysis. Authors can further analyze the results for each round of execution of their tool on each benchmark.