TORI: A Tool for Oracle Quality Assessment

Facundo Molina
Complutense University of Madrid
Madrid, Spain
facundom@ucm.es

Nazareno Aguirre
University of Río Cuarto and CONICET
Río Cuarto, Argentina
naguirre@dc.exa.unrc.edu.ar

Alessandra Gorla
IMDEA Software Institute
Madrid, Spain
alessandra.gorla@imdea.org

Abstract

Oracle quality assessment is a crucial aspect of software testing, as it directly impacts the effectiveness of testing efforts. However, existing approaches for assessing oracle quality often rely on computationally expensive dynamic analyses, which can limit their applicability in large-scale software projects or scenarios where quick feedback is essential.

In this tool demo paper, we present Tori, a static analysis tool that seeks to help software testers to efficiently assess the quality of their test oracles. To achieve this goal, Tori identifies assertion oracles in test cases and provides a report on their quality based on the state field coverage metric, which measures the extent to which the assert statements in a test case cover the state fields of the software under test. A demo video showcasing the use of Tori is available at https://youtu.be/neb3B-Dyacw.

Keywords

Oracle Quality Assessment, Static Analysis.

ACM Reference Format:
Facundo Molina, Nazareno Aguirre, and Alessandra Gorla. 2026. TORI: A Tool for Oracle Quality Assessment. In . ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Software testing is a fundamental activity in software development, aimed at ensuring the quality and reliability of software systems. For testing to be effective, it is not enough to have test inputs that cover relevant scenarios, it is also essential to have high-quality test oracles that can accurately determine the correctness of software behavior. The quality of test oracles directly impacts the effectiveness of testing efforts, as poor-quality oracles can lead to undetected faults and reduce confidence in the testing process.

While the problem of producing high-quality test oracles, known as the *oracle problem* [3], has been widely recognized in the software testing community, the problem of automatically assessing the quality of test oracles has received comparatively less attention. A very common approach is to indirectly evaluate the quality of test oracles by assessing their ability to detect artificially injected faults, through mutation testing [2, 7]. More direct metrics for evaluating oracle quality have also been proposed. Two notable examples are checked coverage [8] and the search-based oracle deficiency detection method OASIs [4]. Checked coverage measures how thoroughly test assertions cover those statements of the software under test (SUT) that influence test outcomes. OASIs quantifies assertion quality by identifying false positives (correct executions incorrectly flagged as erroneous) and false negatives (undetected faults) caused by the oracles. However, these approaches rely on computationally expensive dynamic analyses, limiting their applicability in large-scale software projects or scenarios where quick feedback is essential. Furthermore, the feedback for improving oracles provided by these techniques is indirect (e.g., surviving mutants or unassessed SUT statements), whose translation into oracle improvements is non-trivial.

To alleviate these limitations, we propose Tori, a tool that aims to help software testers efficiently assess the quality of their test oracles without relying on expensive dynamic analyses. Tori achieves both efficiency and the provision of actionable feedback by employing static analysis techniques to identify and analyze the test cases assert statements, the commonly used mechanism to express test oracles. Our tool measures the quality of test oracles by resorting to *state field coverage* [6], a metric that assesses the extent to which the assert statements in a test case cover the state fields of the SUT. One of the main benefits of this metric is that it provides direct feedback on the quality of test oracles, as it can identify state fields that are not covered (i.e., queried or observed) by any assert statement, which can indicate potential weaknesses in the test oracles.

This paper extends our previous work [6], which introduced state field coverage, our metric for assessing oracle quality, by integrating the initial prototype into Tori. Unlike the earlier prototype, Tori is designed to be extensible and customizable, allowing users to define additional metrics for oracle quality assessment beyond state field coverage. Furthermore, this paper provides practical guidance on using Tori, as well as details regarding its availability and implementation.

2 Tool Availability

Tori is publicly available at Zenodo [1], and it is source code is available on GitHub at the following URL:

https://github.com/facumolina/tori

The repository provides detailed instructions on how to install and use Tori, including a Dockerfile to facilitate the installation process. A running example illustrating the tool's usage, instructions on how to configure Tori and how to extend some of its components are also provided. A demo video showcasing the use of Tori is available at https://youtu.be/neb3B-Dyacw.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

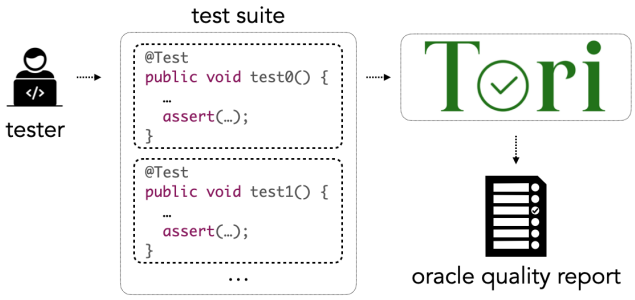

Figure 1: Usage of Tori.

```
class ByteArrayInputStreamWithPos extends MemorySegmentInputStreamWithPos {

    private static final byte[] EMPTY = new byte[0]
    ...

class MemorySegmentInputStreamWithPos extends InputStream {

    private MemorySegment segment;
    private int position;
    private int count;
    private int mark;
    ...

    @Override
    public int read() {
        return (position < count) ? 0xFF & (segment.get(position++)) : -1;
    }
```

(a) Byte array input stream implementation in Apache Flink.

```
class ByteArrayInputStreamWithPosTest {

    private final byte[] data = new byte[] {'0', '1', '2', '3', '4', '5', '6
       ', '7', '8', '9'};

    private final ByteArrayInputStreamWithPos stream = new
       ByteArrayInputStreamWithPos(data);

    @Test
    void testGetMoreThanAvailable() {
        int read = stream.read(new byte[20], 0, 20);
        assertThat(read).isEqualTo(10);
        assertThat(stream.read()).isEqualTo(-1); // exhausted now
    }

    ...
```

(b) Byte array input stream test in Apache Flink.

Figure 2: A target class and its test case in Apache Flink, which oracles will be analyzed by Tori.

3 Usage

Tori's envisioned users are researchers or Java practitioners that may want to assess the quality of the test oracles in a software testing context. A typical scenario in which Tori may be used is illustrated in Figure 1. During testing, the software tester writes or automatically generates test cases for the SUT, which include test oracles (assert statements) to check the correctness of the software behavior. As part of the testing process, the tester may perform different analyses to assess the quality of test suites, such as code coverage, mutation analysis, among others. In this situation, Tori can be used to efficiently get feedback on the quality of the oracles present in the tests.

To properly illustrate the use of Tori, let us consider as a running example a byte array input stream implementation and one of its test cases, taken from the Apache Flink project[1], a popular open source stream processing framework. Figure 2 shows the relevant code snippets. The `ByteArrayInputStreamWithPos` class shown in Figure 2(a) is intended to implement an un-synchronized stream, similar to Java's ByteArrayInputStream, that also exposes the current position. The test case shown in Figure 2(b) is intended to test the `read()` method of the class, which reads a byte of data from the input stream, and has two assert statements (test oracles) that check the correctness of the method's behavior.

In this situation, one can use Tori to analyze the quality of the test oracles. When executed, Tori will perform a static analysis of the test case, identify the assert statements, and compute the state field coverage metric to assess their quality. Intuitively, the state field coverage metric measures the proportion of the state fields (non-static fields) of the SUT that are covered by the assert statements [6]. Whether a state field is covered by an assert statement is determined by statically inspecting the code reachable from the assert statement and checking if it accesses the state field.

In our example, the `ByteArrayInputStreamWithPos` class has four state fields: `segment`, `position`, and `count` and `mark`. The test case shown in Figure 2(b) has two assert statements. The first assertion, `assertThat(read).isEqualTo(10)`, just checks the value of the local variable `read`, which is not a state field of the SUT, and therefore does not contribute to the state field coverage. The second assertion, `assertThat(stream.read()).isEqualTo(-1)`, checks that the value returned by the `read()` method is equal to -1, which indicates that the end of the stream has been reached. By invoking the `read()` method, this assertion indirectly checks the value of the fields `position`, `count` and `segment`, which are used by the `read()` method to determine its return value.

Tori can be executed by running the following command:

```
java tori-1.0.0-all.jar
  -t <path/to/test-class-src-file>
  -m <test-method>
  -metric org.tori.metrics.StateFieldCoverage
  -metric-config <path/to/metrig-config-file>
```

The first two parameters specify the test class source file and the test method to analyze, from where the assert statements will be identified. For our example, these parameters would be set as follows:

```
-t path/to/ByteArrayInputStreamWithPosTest.java
-m testGetMoreThanAvailable
```

The `metric` parameter specifies the metric to compute, which in this case is the state field coverage metric implemented in Tori. Finally, the `metric-config` parameter specifies the path to a properties file that contains the configuration for the metric, which allows users to provide parameters that are specific to the metric being computed. In our example, the metric configuration file contains the following state field coverage-specific parameters:

```
target_class=path/to/ByteArrayInputStreamWithPos.java
exec_level=test_method
```

[1] https://github.com/apache/flink

The `target_class` parameter specifies the source file of the SUT class, which is needed to identify the state fields of the SUT and determine whether they are covered by the assert statements. The `exec_level` parameter specifies the execution level at which to compute the metric, which in this case is set to `test_method`, indicating that the metric should be computed at the level of individual test methods (all assert statements within the test method will be analyzed together to compute the metric).

Running the above command will execute Tori and compute the state field coverage metric for the test method's assert statements. The output will include the computed metric value and will also report the uncovered state fields, if any. For our example, Tori reports the following output:

```
state_field_coverage_score: 0.75
uncovered_fields: mark
```

The state field coverage is 0.75, as 3 out of the 4 state fields of the SUT are covered by the assert statements, and the state field `mark` is reported as uncovered, as it is not covered by any assertion.

Overall, the results provided by Tori can help the tester identify potential weaknesses in the test oracles, such as state fields that are not covered by any assert statement, which can indicate that certain aspects of the software behavior are not being adequately checked by the tests.

4 Tori

Let us now discuss the implementation details of Tori. Figure 3 shows an overview of our tool, which is implemented in Java. Tori works in two main steps: the oracles identification step and the assessment step, where a specified metric is computed. Given a test suite $\mathcal{T}$ containing a set of test cases, Tori first identifies the test oracles present in the test cases, which are essentially the assert statements. Then it computes the specified metric to assess the quality of the identified test oracles. Below we describe these two steps in more detail.

4.1 Target Oracles Identification

To identify the target oracles, Tori performs a static analysis process to load the given test suite and collect assert statements from the test methods. For this process, Tori relies on a static analysis component that uses the `tree-sitter` library[2] to parse the test class source file and build its abstract syntax tree (AST). Tori then traverses the AST to collect the assert statements.

The assert statements collected to be analyzed depend on whether a target method is specified and on the execution level at which the analysis is performed, which is specified by the user. If a target method is specified, Tori will only focus on the assert statements present in that method, while if no target method is specified, Tori will analyze all assert statements present in the test class. Then, the execution level determines how the identified assert statements are grouped together for the analysis. There are three execution levels available in Tori: `assert`, `test_method`, and `test_class`. In the `assert` mode, all assert statements are analyzed individually, treating them as separate oracles. In the `test_method` mode, all assert statements present in the test method being analyzed are considered together as a single oracle, while in the `test_class` mode, all assert statements present in the test class are considered together. These different levels allow users to analyze the quality of test oracles at different granularities, which can provide different insights into the quality of the test oracles at different levels.

Finally, to actually identify assert statements, Tori inspects all method calls present in the test cases and checks whether the name of the invoked method starts with `assert`. In this way, Tori not only identifies assert statements that use the standard Java `assert` keyword, but also statements that use assertion methods provided by testing frameworks, such as JUnit [3] and AssertJ [4].

4.2 Oracle Assessment

Once the target oracles are identified, Tori computes the specified metric to assess their quality. While it currently provides two metrics, Tori is designed to be extensible and customizable, allowing users to define and implement additional metrics for oracle quality assessment. Below we describe the metrics currently implemented in Tori.

4.2.1 State Field Coverage. The state field coverage metric [6] assesses the quality of test oracles by measuring the proportion of the SUT's state definition (i.e., class fields, in the context of object-oriented code) referenced by the oracle. A field is considered "covered" if it is directly or indirectly referred to in oracle expressions. Being statically computed, the state field coverage metric provides an efficient way to assess oracle quality, as it does not require executing the tests or performing dynamic analyses.

Additionally, the metric provides actionable feedback by explicitly identifying state fields omitted from oracles. Every time a field is reported as uncovered, testers can analyze the reasons behind this omission and determine whether it is a consequence of an oversight in the test design or if it is intentional. This feedback can help testers to easily identify potential weaknesses in their test oracles, and guide them in improving their test suites by adding new assertions that cover the uncovered state fields.

To compute the state field coverage, Tori requires the source file of the SUT class, which is used to identify the state fields of the SUT and determine whether they are covered by the assert statements. Tori employs `tree-sitter`-based static analyses to identify target fields and to inspect the code reachable from the identified assert statements. To identify target fields, Tori inspects the AST of the SUT class and collects all non-static fields reachable from the SUT class, i.e., all non-static fields defined in the SUT class, its superclasses and the classes of its fields, recursively. To determine whether a field is covered by an assert statement, Tori recursively inspects the code reachable from the assert statement, which includes the code of the test method and the code of any method invoked from the test method. If any of the reachable code accesses the field, then the field is considered covered by the assert statement. The state field coverage is computed as the proportion of state fields that are covered by the assertions, and reported depending on the specified execution level.

[2] https://tree-sitter.github.io/

[3] https://junit.org/

[4] https://assertj.github.io/

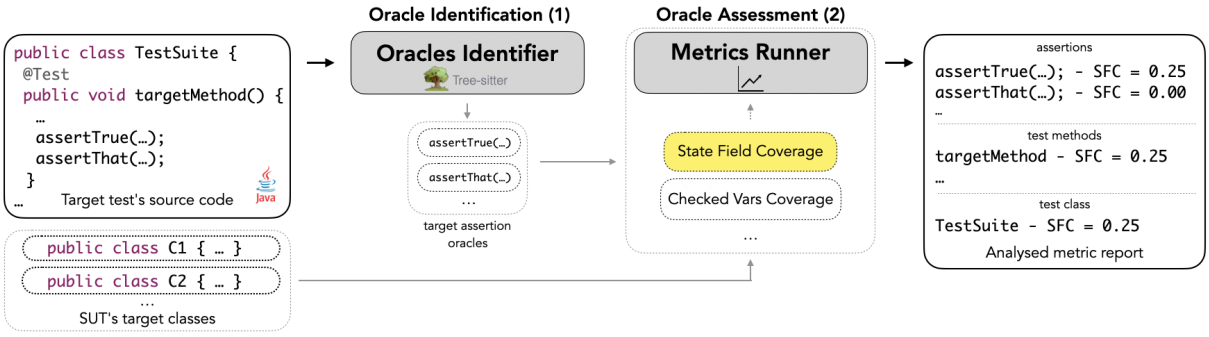

Figure 3: An overview of Tori.

4.2.2 Checked Variable Coverage. Tori also includes a second and more straightforward metric, called checked variable coverage, which measures the proportion of variables, i.e., local variables defined in the test method, that are checked by the assert statements. This metric is also statically computed and provides a more coarse-grained assessment of oracle quality,as it only considers the local variables defined in the test method, which are typically used to store intermediate results or values obtained from the SUT, and checks whether they are "used" by the assert statements.

Tori's current implementation allows to analyze oracles in Java test cases. In the future, we plan to extend Tori to support the analysis of test oracles in other programming languages. That's is the reason why we use `tree-sitter` for the static analysis of test cases, as it supports a wide range of programming languages.

5 Evaluation

In our previous work [6], we evaluated the *state field coverage* implementation available on Tori. Our evaluation focused on whether state field coverage is an effective metric for assessing oracle quality, by analyzing its correlation with the fault-detection ability of test oracles, measured as the detection of artificially injected faults. Our experiments, involving 83,032 test assertions taken from the Defects4J benchmark [5] (version 2.0.1), which includes developer-written test assertions for real-world projects, show that state field coverage is an effective metric for assessing oracle quality, as it strongly correlates with fault-detection ability measured by mutation score (proportion of mutants detected by the test assertions).

Our evaluation also showed that static metrics like state field coverage address a key limitation of dynamic techniques, by avoiding their computational costs while maintaining strong correlation with fault detection. Moreover, our experiments demonstrate that state field coverage can effectively guide oracle improvement, as its ability to identify uncovered state portions directly supports targeted assertion generation for improved fault detection.

In the future, we plan to conduct a more comprehensive evaluation of Tori, including a user study to evaluate the usefulness of the feedback provided by Tori in guiding oracle improvement.

6 Conclusion

Tori is a tool that employs static analysis techniques to efficiently assess the quality of test oracles in Java test cases. To do so, Tori identifies assertion oracles in test cases and provides means to compute different metrics for assessing their quality. Given a metric to compute, Tori performs a static analysis of the test cases to compute the metric and provide feedback on the quality of the test oracles. Tori reports the quality of test oracles based on the computed metric, and allows to do so at different execution levels, which can provide different insights into the quality of test oracles at different granularities (e.g., at the level of individual assertions, test methods, or test classes). Thanks to its modular design, Tori can be easily extended to support additional metrics for oracle quality assessment, and to analyze test oracles in other programming languages.

Acknowledgments

This work was funded by the Madrid regional government program S2018/TCS-4339 (BLOQUES-CM) and by the Spanish Governement MCIN/AEI/10.13039/501100011033/ERDF through grants TED2021-132464B-I00 (PRODIGY) and PID2022-142290OB-I00 (ESPADA). Those projects are co-funded by European Union ESF, EIE, and NextGeneration funds.

References

[1] 2026. Tori's implementation. https://doi.org/10.5281/zenodo.20132516
[2] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing.* Cambridge University Press. https://doi.org/10.1017/CBO9780511809163
[3] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785
[4] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 247–258. https://doi.org/10.1145/2931037.2931062
[5] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055
[6] Facundo Molina, Nazareno Aguirre, and Alessandra Gorla. 2025. State Field Coverage: A Metric for Oracle Quality. In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025.* IEEE, 2707–2719. https://doi.org/10.1109/ASE63991.2025.00222
[7] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. *Adv. Comput.* 112 (2019), 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015
[8] David Schuler and Andreas Zeller. 2011. Assessing Oracle Quality with Checked Coverage. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011.* IEEE Computer Society, 90–99. https://doi.org/10.1109/ICST.2011.32