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
Guangdong Technion - Israel Institute of Technology
Shantou, China
naguirre@dc.exa.unrc.edu.ar

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

Abstract

Oracle quality is a critical factor in software testing, directly influencing the fault-detection capability and overall effectiveness of test suites. Despite its importance, assessing oracle quality remains challenging: existing techniques predominantly rely on dynamic analysis, which incurs substantial computational overhead and limits adoption in large-scale or time-constrained development settings.

In this tool demo paper, we present Tori, a lightweight static analysis tool designed to help testers evaluate oracle quality efficiently. Tori operates by analyzing assertion-based oracles within test cases and generating a quality report based on the state field coverage metric, a measure of how comprehensively the assertions exercise the internal state fields of the software under test.

A demonstration video showcasing the use of Tori is available at https://youtu.be/neb3B-Dyacw.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; **Automated static analysis**.

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 *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3832783.3834653

1 Introduction

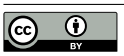
Software testing is a fundamental activity in the software development life cycle, aimed at detecting faults and ensuring the reliability of software systems [2]. A test suite comprises two essential components: test inputs that exercise the system and cover relevant scenarios, and test oracles that determine whether the observed behavior is correct [9]. While substantial research has focused on test input generation and selection criteria, considerably less attention has been devoted to assessing the effectiveness of test oracles and their impact on the testing process [3, 9]. Test oracle assessment is very important, as the quality of test oracles directly affects fault-detection effectiveness: weak or missing oracles can allow faults to go undetected even when the faulty code is executed, reducing 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* oracle quality has received comparatively less attention. A common indirect approach is to evaluate oracle quality through mutation testing, where artificially injected faults serve as proxies for real defects [2, 7]. While this method provides useful insights into fault-detection capability, it is computationally expensive and the connection between mutant survival and oracle deficiency is not always straightforward.

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 technique OASIs [4]. Checked coverage measures how thoroughly test assertions exercise 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). However, both approaches rely on computationally expensive dynamic analyses, which limits their applicability in large-scale projects or fast-paced development settings where quick feedback is essential. Moreover, the feedback they provide, such as lists of surviving mutants or unexercised SUT statements, is indirect and offers little guidance on how to actually improve the oracles.

To alleviate these limitations, we propose Tori, a static analysis tool that enables software testers to efficiently assess the quality of their test oracles without the overhead of dynamic execution. Unlike existing approaches that require running the SUT and instrumenting its execution, Tori operates directly on the source code, analyzing test cases to extract and examine their assertion statements, the predominant mechanism for expressing oracles in modern test suites. By avoiding test execution, Tori provides very efficient feedback, making it suitable for large codebases where even moderate analysis overhead can become prohibitive.

Tori measures oracle quality using *state field coverage* [6], a metric that assesses the extent to which the assert statements in a test case observe or query the state fields of the SUT. Intuitively, an assertion that checks the value of a particular field after a sequence of operations exercises that field; conversely, state fields that are



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834653

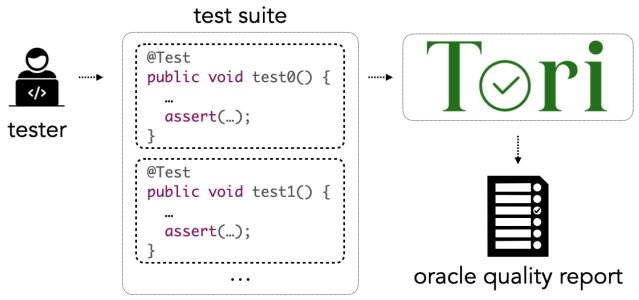


Figure 1: Usage of Tori.

never referenced by any assertion remain unchecked, leaving potential faults in those fields undetected even if the relevant code is executed. The metric thus provides direct, actionable feedback: it produces a report highlighting which state fields are covered by existing assertions and, more importantly, which ones are not. This enables testers to immediately identify gaps in their oracles and take targeted actions, such as adding or strengthening assertions, rather than relying on indirect signals like surviving mutants or uncovered statements that require further interpretation to translate into concrete improvements.

This paper extends our previous work [6], which introduced state field coverage as a metric for assessing oracle quality, by presenting Tori, a tool that implements this metric and makes it better accessible to practitioners. While our earlier prototype demonstrated the feasibility of static oracle quality assessment, Tori goes beyond a proof of concept by offering a more robust, extensible architecture that supports additional metrics beyond state field coverage. This modular design allows researchers and practitioners to plug in custom assessment criteria, making Tori a flexible platform for experimenting with novel oracle quality measures. Beyond the tool itself, this paper provides guidance on interpreting the tool's output, and acting upon the feedback it provides. We also discuss implementation details, as well as Tori's availability as open-source software to facilitate adoption and further research.

2 Tool Availability

Tori is publicly available as open-source software. The source code, along with installation scripts and usage examples, can be accessed on GitHub at:

https://github.com/facumolina/tori

For archival purposes and to ensure reproducibility, a stable snapshot of the tool is also available on Zenodo [1], with a persistent DOI.

The repository includes comprehensive documentation covering installation, configuration, and usage. To streamline the setup process, a Dockerfile is provided, allowing users to run Tori in a containerized environment without manually managing dependencies. The documentation also includes a step-by-step running example that demonstrates the tool's main features, from analyzing a subject project to interpreting the generated quality report. Additionally, users interested in extending Tori will find guidelines on how to implement custom metrics and integrate them into the tool's modular architecture.

A demo video illustrating the tool's functionality and typical usage scenarios is available at:

https://youtu.be/neb3B-Dyacw

3 Usage

Tori is designed for researchers and Java practitioners who wish to assess the quality of test oracles in their software testing projects. A typical usage scenario is illustrated in Figure 1. In this scenario, a software tester writes—or generates automatically—test cases for the software under test (SUT), each of which includes one or more assertion statements that serve as oracles to verify the correctness of the observed behavior. During the testing process, testers commonly employ various quality assessment techniques, such as code coverage analysis to measure test completeness, or mutation testing to evaluate the effectiveness of the test suite. While these techniques provide valuable information about the tests themselves, they offer only indirect insights into the quality of the oracles. In this context, Tori can be invoked to obtain rapid, targeted feedback specifically on oracle quality, complementing existing analyses without requiring additional test execution or instrumentation.

To illustrate the use of Tori in practice, we 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), implements an unsynchronized input stream similar to Java's standard `ByteArrayInputStream`, but with the added capability of exposing the current read position to the caller. This additional functionality is the main feature under test. The test case in Figure 2(b) exercises the `read()` method, which reads a single byte from the stream and advances the position accordingly. The test contains two assert statements that serve as oracles: one checks that the read byte matches the expected value, and the other verifies that the internal position has been correctly updated. These assertions represent the test author's expectations about the method's behavior under the given inputs.

In this situation, one can invoke Tori to analyze the quality of the test oracles. When executed, Tori performs a static analysis of the test case, identifies the assert statements, and computes the state field coverage metric to assess their quality.

Intuitively, state field coverage measures the proportion of state fields (i.e., non-static fields) of the SUT that are referenced by at least one assert statement [6]. The analysis determines coverage by inspecting the code reachable from each assertion—including method invocations and field accesses within those methods—to check whether a state field is accessed, either directly or indirectly. Assertions that only check local variables, literal values, or constants without consulting any SUT state contribute no coverage. Because this analysis is purely static, Tori delivers its quality assessment without requiring test execution, providing feedback that testers can obtain quickly and act upon without additional overhead.

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

```
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.

In our running example, the `ByteArrayInputStreamWithPos` class has four state fields: `segment`, `position`, `count`, and `mark`. The test case shown in Figure 2(b) contains two assertions. The first assertion, `assertThat(read).isEqualTo(10)`, 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, indicating that the end of the stream has been reached. By invoking the `read()` method, this assertion indirectly checks the values of the `position`, `count`, and `segment` fields, which are used by the `read()` method to determine its return value. Thus, Tori would report that three out of the four state fields are covered by the test's oracles, with `mark` being the only uncovered field.

Tori can be executed from the command line using 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 `-t` and `-m` arguments specify, respectively, the source file of the test class and the particular test method to analyze. These parameters allow Tori to locate the relevant test code and identify the assert statements contained within the target method. For our running example, the parameters would be set as follows:

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

The `-metric` parameter specifies which quality metric to compute. In this case, it is set to `org.tori.metrics.StateFieldCoverage`, the fully qualified name of the class that implements the state field coverage metric within Tori. This design allows users to easily switch to other available metrics or even plug in custom ones by providing the appropriate class name.

The `-metric-config` parameter points to a properties file containing configuration settings for the selected metric. This file enables users to supply metric-specific parameters without modifying the tool's source code. 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
```

The `-target-class` parameter specifies the source file of the SUT class. This parameter is required, as Tori needs to analyze the class to identify its state fields and subsequently determine which of them are covered by the assert statements found in the test method.

The `-exec-level` parameter defines the granularity at which the metric is computed. In this example, it is set to `test_method`, meaning that all assert statements within the specified test method are analyzed together to produce a single coverage result.

Running the command above executes Tori and computes the state field coverage metric for the assert statements within the specified test method. The output reports the computed metric value and, more importantly, lists any uncovered state fields, allowing testers to quickly identify gaps in their oracles.

For our running example, Tori produces the following output:

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

In the output, the state field coverage is reported as 0.75, indicating that three out of the four state fields of the SUT are covered by the test's assertions. The `mark` field appears in the list of uncovered fields, confirming that no assertion in the test method references any code that accesses this field.

Overall, the results provided by Tori help testers identify potential weaknesses in their test oracles. An uncovered state field suggests that certain aspects of the software behavior—those involving that field—are not being verified by any assertion, even if the relevant code is executed. This insight enables testers to take targeted action, such as adding new assertions or extending existing ones to cover the identified gaps, ultimately leading to more effective test suites and greater confidence in the correctness of the system.

4 Tori

Let us now discuss the implementation details of Tori. Figure 3 provides a high-level overview of the tool's architecture, which is implemented in Java. Tori operates in two main phases: an oracle

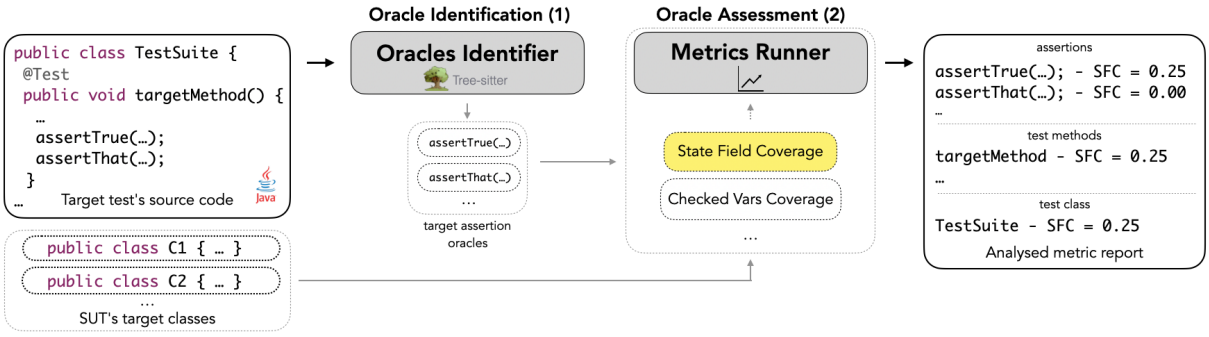


Figure 3: An overview of Tori.

identification step, followed by an assessment step that computes the user-specified metric.

Given a test suite $\mathcal{T}$ containing a set of test cases, Tori first scans the source code to locate and extract all assertion statements, which serve as the test oracles. Once the oracles have been identified, the tool proceeds to the assessment phase, where it computes the selected metric (in our case, state field coverage) over the collected assertions. Below we describe each of these steps in greater detail.

4.1 Target Oracles Identification

To identify the target oracles, Tori performs a static analysis that loads the given test suite and collects assert statements from the test methods. For this purpose, Tori relies on a static analysis component built on top of the `tree-sitter` library[2], a widely used incremental parsing framework that supports multiple programming languages. The component parses each test class source file and constructs its corresponding abstract syntax tree (AST). Tori then traverses the AST to locate and extract assertion statements, which represent the test oracles to be assessed.

The assert statements collected for analysis depend on two user-specified parameters: the target test method and the execution level. If a target method is specified via the `-m` parameter, Tori focuses only on the assert statements present in that method; if no target method is specified, the tool analyzes all assert statements in the test class. The execution level then determines how the identified assert statements are grouped for the analysis. Tori supports three execution levels: `assert`, `test_method`, and `test_class`. In the `assert` mode, all assert statements are analyzed individually, treating each assertion as a separate oracle. In the `test_method` mode, all assert statements within the test method being analyzed are considered together as a single oracle, while in the `test_class` mode, all assert statements across the test class are aggregated. These different levels allow users to assess oracle quality at varying granularities, providing complementary insights depending on whether one wishes to focus on individual assertions, whole test methods, or the test suite as a whole.

Finally, to identify assert statements, Tori inspects all method invocations within the test cases and checks whether the name of the invoked method starts with the prefix `assert`. This approach allows Tori to recognize not only standard Java `assert` statements, but also assertion methods provided by popular testing frameworks such as JUnit [3] and AssertJ [4].

4.2 Oracle Assessment

Once the target oracles have been identified, Tori proceeds to compute the user-specified metric to assess their quality. Although the tool currently provides two built-in metrics, it is designed to be extensible and customizable, enabling users to define and implement additional metrics for oracle quality assessment as needed. 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., the class fields, in the context of object-oriented code—that are referenced by the oracle. A field is considered "covered" if it is directly or indirectly referred to in oracle expressions, either through explicit field access or via method invocations that ultimately read the field's value.

Being computed statically, the state field coverage metric provides an efficient means of assessing oracle quality, as it does not require executing the tests or performing any dynamic analysis. This makes it particularly suitable for scenarios where rapid feedback is essential, such as continuous integration pipelines or large-scale codebases where dynamic approaches may be prohibitively expensive.

Beyond its efficiency, the metric also delivers actionable feedback by explicitly identifying which state fields are omitted from the oracles. Each uncovered field serves as a signal for testers to investigate whether the omission is an oversight in the test design or an intentional design choice. This direct feedback helps testers pinpoint potential weaknesses in their test oracles and provides clear guidance for improvement, such as adding new assertions that cover the missing fields, ultimately leading to more thorough and reliable test suites.

To compute the state field coverage, Tori requires the source file of the SUT class, which is used both to identify the state fields of the SUT and to determine whether they are covered by the assert statements. For these tasks, Tori employs `tree-sitter`-based static analyses to parse the source code and traverse its AST.

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

[3] https://junit.org/

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

To identify the target fields, Tori inspects the AST of the SUT class and collects all non-static fields that are reachable from the SUT class. This includes not only fields defined directly in the SUT class itself, but also fields inherited from its superclasses, as well as fields of the types referenced by the SUT's own fields, recursively. This comprehensive collection ensures that the metric accounts for the full state space that the SUT manages.

To determine whether a field is covered by an assert statement, Tori recursively inspects the code reachable from that assertion. The reachable code includes the body of the test method itself, as well as any methods invoked from it, whether directly or indirectly. If any of this reachable code accesses the field—either by directly reading it or by invoking a method that reads it—the field is considered covered by the assertion. The state field coverage is then computed as the proportion of state fields that are covered by the assertions, aggregated according to the execution level specified by the user.

4.2.2 Checked Variable Coverage. Tori also includes a second, more straightforward metric called checked variable coverage, which measures the proportion of local variables defined in the test method that are checked by the assert statements. Like state field coverage, this metric is computed statically and does not require test execution. It provides a more coarse-grained assessment of oracle quality, as it focuses solely on local variables—typically used to store intermediate results or values obtained from the SUT—and determines whether they are referenced by any assertion in the test method.

Tori's current implementation supports the analysis of test oracles in Java test cases. However, the tool has been designed with language extensibility in mind. This is the primary reason we adopted `tree-sitter` as our parsing framework: it supports a wide range of programming languages, which will facilitate future extensions to other languages beyond Java.

5 Evaluation

In our previous work [6], we evaluated the *state field coverage* implementation available in 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

In this paper, we presented Tori, a static analysis tool that enables software testers and researchers to efficiently assess the quality of test oracles in Java test cases. Tori identifies assertion-based oracles within test suites and provides a flexible framework for computing different quality metrics over them. The tool currently implements two metrics: state field coverage, which measures the proportion of SUT state fields referenced by assertions, and checked variable coverage, which measures the proportion of local variables checked by assertions. Both metrics are computed statically, without requiring test execution, making Tori suitable for scenarios where rapid feedback is essential.

Given a user-specified metric, Tori performs a static analysis of the test cases, computes the selected metric, and delivers actionable feedback to the tester. The tool reports the assessment results at different execution levels, allowing users to obtain insights at different granularities, from individual assertions to entire test classes.

Thanks to its modular architecture, Tori is designed to be easily extended to support additional metrics for oracle quality assessment. We believe Tori represents a valuable contribution to the software testing community, offering a lightweight, actionable, and extensible approach to oracle quality assessment that complements existing dynamic techniques.

Acknowledgments

This work is partially supported by the Ramón y Cajal fellowship RYC2020-030800-I, by the Spanish Government through grants TED2021-132464BI00 (PRODIGY), PID2022-142290OB-I00 (ESPADA), and CEX2024-001471-M funded by MICIU/AEI/10.13039/5011000 11033 and by the Comunidad de Madrid as part of the ASCEND project co-funded by FEDER Funds of the European Union.

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
[9] Matt Staats, Michael W. Whalen, and Mats Per Erik Heimdahl. 2011. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 391–400. https://doi.org/10.1145/1985793.1985847