State Field Coverage: A Metric for Oracle Quality

Facundo Molina*, Nazareno Aguirre†‡ and Alessandra Gorla*
*IMDEA Software Institute, Madrid, Spain
facundo.molina@imdea.org, alessandra.gorla@imdea.org
†University of Rio Cuarto and CONICET, Rio Cuarto, Argentina
naguirre@dc.exa.unrc.edu.ar
‡Guangdong Technion-Israel Institute of Technology, Shantou, China

***Abstract*—The effectiveness of testing in uncovering software defects depends not only on the characteristics of the test inputs and how thoroughly they exercise the software, but also on the quality of the oracles used to determine whether the software behaves as expected. Therefore, assessing the quality of oracles is crucial to improve the overall effectiveness of the testing process. Existing metrics have been used for this purpose, but they either fail to provide a comprehensive basis for guiding oracle improvement, or they are tailored to specific types of oracles, thus limiting their generality.**

In this paper, we introduce *state field coverage*, a novel metric for assessing oracle quality. This metric measures the proportion of an object's state, as statically defined by its class fields, that an oracle may access during test execution. The main intuition of our metric is that oracles with a higher state field coverage are more likely to detect faults in the software under analysis, as they inspect a larger portion of the object states to determine whether tests pass or not.

We implement a mechanism to statically compute the state field coverage metric. Being statically computed, the metric is efficient and provides direct guidance for improving test oracles by identifying state fields that remain unexamined. We evaluate state field coverage through experiments involving 273 representation invariants and 249,027 test assertions. The results show that state field coverage is a well-suited metric for assessing oracle quality, as it strongly correlates with the oracles' fault-detection ability, measured by mutation score.

I. Introduction

Improving the reliability of software systems is among the most challenging problems in software engineering. This problem is strongly related to finding software defects, i.e., identifying software behaviors that diverge from the expected behavior. Software testing is one of the most widely used systematic techniques to identify such software defects, and it demands various complex tasks [7]. Firstly, software testing requires crafting (manually or in an assisted manner) test inputs that are able to exercise the software under test (SUT) in realistic and sufficiently varied scenarios. Secondly, to increase the automation in test suite execution and checking, it is crucial to capture the intended behavior of software for the designed test cases through *test oracles*, assertions that attempt to capture the expectations on the execution of test cases as accurately as possible. The problem of producing accurate test oracles, known as the oracle problem [9], has proved to be both difficult and time-consuming.

Constructing accurate test oracles necessarily depends on the intended software behavior and is largely a manual task. It involves the software developers directly, who can greatly benefit from support in oracle construction, particularly via mechanisms to assess oracle quality. Indeed, oracles can be inaccurate either by misrepresenting the developer's intent or by being too weak, i.e., approximating the intended behavior in a way that fails to reveal many faults. An effective technique to assess oracle quality can direct developers to inaccuracies and other limitations, helping them produce stricter and more accurate oracles with an improved ability to detect defects.

Various approaches have been proposed to evaluate oracle quality. Among these, mutation testing [7], [41], a well-established technique for assessing test suite effectiveness, has been widely adopted due to its ability to approximate the fault-detection capability of tests and oracles. Additionally, more direct metrics have been introduced, such as *checked coverage* [43], which measures how thoroughly test assertions cover SUT statements that influence their outcomes, and the search-based oracle deficiency detection put forward in [24], which quantifies assert statement quality by identifying false positives (correct executions incorrectly flagged as erroneous) and false negatives (undetected faults) that the oracles lead to.

Despite these advances in oracle assessment, existing approaches suffer from various limitations. These techniques typically rely on dynamic analyses that are computationally expensive. In particular, mutation analysis requires repeated test executions to identify killed and surviving mutants; checked coverage depends on dynamic slicing to trace SUT statements affecting oracles; and search-based oracle deficiency detection combines evolutionary computation [34] with mutation analysis, both computationally expensive tasks. Furthermore, while these techniques provide accurate oracle quality metrics, their feedback for improving oracles remains indirect (e.g., surviving mutants or unassessed SUT statements, whose translation into oracle improvements is non-trivial).

In this paper, we introduce *state field coverage*, a novel metric for oracle assessment. Our approach evaluates oracle quality 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. As opposed to existing techniques to assess oracle quality, we present an approach that computes our metric statically, thus avoiding costly dynamic analyses. Moreover, our approach provides actionable feedback by explicitly identifying

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00222

uncovered state fields, guiding oracle enhancements. Also, while other metrics (e.g., mutation testing) do yield explicit feedback, they do not easily indicate oracle improvements. For example, surviving mutants indicate undetected faults, but require providing additional test data or crafting new tests to kill these mutants [15], which are typically non-trivial to provide. In contrast, state field coverage identifies specific state variables omitted from oracles, directing the improvement process.

Our experiments, comprising oracles originating from 273 representation invariants and 249,027 test assertions, 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.

II. Related work

This section discusses some of the established approaches to assess the quality of oracles, and other related works that are relevant to our proposal.

A. Oracle Quality Assessment

1) Mutation-based Oracle Assessment: Mutation testing [7], [41] is a widely used technique to assess the quality of test suites, in terms of the ability of test suites to detect (artificial) faults in the SUT. More precisely, mutation testing generates mutants of the SUT by injecting artificial faults in the code, and then measures the proportion of mutants that are detected (or killed) by the test suite, i.e., that make at least one test case fail. The proportion of mutants that are killed, known as the *mutation score*, is known to correlate with real fault detection better than other traditional testing metrics [27], [42]. This score has also been effectively used as a metric to assess the quality of oracles in a variety of techniques for test oracle automation [6], [11], [18]–[21], [36]–[38], [44] and oracle assessment studies [22], [24], [43], [47]. These works follow an approach similar to the use of mutation for test suite assessment: given the SUT and an oracle (e.g., an assertion) for it, the quality of the oracle is measured by its ability to *kill* mutants, i.e., to identify mutants through violations to the oracle, e.g., on generated or provided test suites. Through this analysis, one can measure the *strength* of the oracles in terms of their ability to detect faults.

Mutation-based oracle analysis can be effective in revealing potential oracle weaknesses, by identifying faults that the oracles are not able to detect. However, it also has some limitations. Firstly, the result of the mutation analysis points to the mutants that are not detected by the oracle, but exploiting this feedback to improve the oracle itself (i.e., to recognize the relationship between mutants and improvements to the oracle) is non-trivial. Secondly, oracle assessment based on mutation usually combines test generation with mutation analysis, both computationally expensive tasks, notably the latter.

2) Coverage-based Approaches: Checked coverage [43] is a metric that focuses on the evaluation of the quality of test assertions. The checked coverage metric essentially works by analyzing the code features that affect the expressions involved in test oracles. To calculate checked coverage, a dynamic backward slice of the test oracles is computed, which determines the statements that contribute to the checked expressions. The percentage of statements that contribute to the expressions checked in the test oracles, in relation to the total number of statements of the SUT, constitutes the checked coverage. In other words, it attempts to measure the proportion of sentences of the SUT whose effect is being directly or indirectly checked by the test assertions.

Checked coverage has proved to be a good indicator of test oracle quality, correlated with the ability of the oracles to detect faults and even more sensitive than mutation score [43]. However, checked coverage concentrates on the evaluation of test assertions, and is not easily adaptable to support other more general types of oracles, such as contract assertions (e.g., pre and postconditions) [32], [33]. Consider, for instance, a postcondition. Since the assertion is local to a method, it seems more reasonable to measure checked coverage with respect to sentences only of the method itself; the generality of such assertions together with their locality to specific methods would typically lead to high checked coverage values. Additionally, computing checked coverage is expensive, demanding from several hours to days for large projects [29], since it requires the computation of dynamic slices [30].

State coverage [31] is another approach based on statement coverage. As with checked coverage, state coverage also relies on program slicing. It measures the quality of checks (e.g., test assertions) by considering all output defining statements (ODS), i.e., statements that define an output variable. State coverage is computed by counting the number of ODS present in the dynamic slices of a given assertion, divided by the total number of ODS. Being based on ODS, this technique depends on the given test inputs (as for different inputs, different statements may be output defining). A positive aspect of the approach is that, in principle, it can provide ODS that are not being checked by the assertions, which may be useful for users to further improve the oracles. However, given the very limited evaluation that exists for this technique, just a small experiment with a proof of concept implementation in a short paper [31], there is no significant evidence to support its usefulness, or reveal deeper insights or limitations.

A second implementation of state coverage has also been proposed [45], implementing an extension in which the state coverage is computed as the ratio of state updates that are read by assertions with respect to the total number of state updates. Every code location in the source code in which an update is performed is considered a state update. To compute state coverage, a test suite is executed and monitored using a process that collects the set of state updates (writes), and the subset of updates read in the test assertions (reads); state coverage is the ratio of reads over the number or writes.

3) Oracle Deficiencies: The deficiencies of an oracle can be determined both qualitatively and quantitatively. For instance, OraclePolish [23] is a dynamic technique that qualitatively assesses the quality of test assertions by analyzing how they interact with the inputs in a specific test. The technique focuses

on detecting brittle assertions, i.e., assertions that check values of uncontrolled inputs (e.g, inputs declared outside the tests), and unused test inputs, i.e., inputs in the tests that are not checked in the assertions. Similarly to the checked coverage metric, OraclePolish is also tailored for test assertions, and it is not easily adaptable to other types of oracles.

The quality of an oracle can be quantitatively assessed by identifying more concrete oracle deficiencies: false positives and false negatives [24]. A false positive is a correct and expected program state for which the oracle fails, i.e., a false alarm; a false negative, on the other hand, is an incorrect and unexpected program state for which the oracle is true, i.e., a missed fault. OASIs [24] is a tool for automatically assessing the quality of oracles, by computing the above oracle deficiencies. It has been used as a metric for oracle quality [38] and to guide the manual as well as the automated improvement of oracles [24], [25], [44]. OASIs searches for false positives and false negatives using evolutionary computation. False positives are reported as test cases that falsify the oracle when they should not. False negatives, on the other hand, are calculated as mutations that are not detected by the oracle, and thus are based on mutation analysis. Oracle deficiencies provide a more direct input to the improvement of oracles. Their computation is however expensive, and is designed for specific oracles, expressed as assert statements in the code.

In general, the dynamic nature of the existing metrics for oracle quality makes them computationally expensive, and the use of their corresponding results as inputs for oracle improvement may demand subsequent time-consuming analyses. Some of the approaches, e.g., checked coverage and oracle deficiency identification, concentrate on specific kinds of oracles. *State field coverage*, the novel metric for oracle quality that we introduce in Section III, aims to overcome these limitations. Our metric can be efficiently computed, provides an output that more directly leads to oracle improvement, and is applicable to different kinds of oracles, including test assertions and contract specifications such as operational class invariants.

B. Automated Test Oracle Generation

Most of the metrics for oracle quality assessment introduced in the previous section have been used either to evaluate the quality of automatically inferred oracles, or to guide an oracle inference process. Notably, mutation testing has been the most widely used approach to assess the quality of automatically derived oracles [8], [10], [11], [36], [39], [44], [46]. Typically, these oracle generation approaches observe some artifact of the SUT, such as code comments or software executions, and infer oracles from these artifacts. Then, mutation analysis is used to assess the quality of the inferred oracles by measuring their ability to detect artificial faults (mutants). Moreover, mutation analysis has also been used to guide the oracle inference process by trying to maximize the mutation score [18], [36], [38] or by prioritizing the mutants used in the analysis [19], leading to more efficient processes and more precise oracles.

More recently, oracle deficiencies have also been utilized as part of oracle inference processes. In particular, the OASIs tool has been used as a core component in a technique that implements an evolutionary approach that tries to infer assertions minimizing the number of false positives and false negatives [44]. Though effective, computing oracle deficiencies using dynamic analysis, as in the case of OASIs, is computationally expensive [44].

Our state field coverage metric may also be used to evaluate the quality of automatically generated oracles, as well as to guide oracle inference processes by optimizing the state field coverage. In fact, as it is possible to compute our metric statically, it is expected to be more efficient than the existing metrics, and thus more suitable for guiding the oracle inference process. We plan to explore this application in future work.

III. The State Field Coverage Metric

In this section, we formally introduce *state field coverage*, our proposed metric to assess oracle quality. This metric is based on the idea of analyzing to what extent a given oracle predicates over the state of the software under analysis. The intuition here is rather straightforward: the more an oracle examines the software state, the better the oracle is. Instead of assessing state field coverage in a dynamic fashion, our approach concentrates on how the state is statically defined. Indeed, from the state definition of the software under analysis, e.g., in an object oriented setting, the definitions of fields in the classes that compose the software, we build a graph-like summary that we call the *type graph*, and statically analyze the proportion of the type graph that is involved in the oracle, i.e., the direct and indirect fields present in the oracle expressions. Below we formally define these concepts, and show how our metric is computed.

Let us first assume that the software under analysis is organized as a collection $C_1, \ldots, C_n$ of classes, where C_1 is a distinguished root class. Each class C_i defines a set of fields, whose types are among $C_1, \ldots, C_n$ and primitive datatypes. From a given class C, we can compute the set F_c of reachable fields, which includes all the fields in C plus all the reachable fields of the classes C_i for which there is a field in C of type C_i. A field $f \in F_c$ is *iterable* if its type is a collection (e.g., arrays, sets, etc), and *non-iterable* otherwise. Moreover, recursive fields (fields from a class to itself), and other fields present in classes that contain at least one recursive field (e.g, value fields in nodes), are also considered iterable, as they enable iteration over objects of linked structures. The set of iterable fields is denoted by I_c.

A. Type Graph

A *type graph*, introduced in [35], is an abstract representation of a class that captures the relationships between the types of all the reachable fields of a software under analysis.

Definition 1: Given a class C, its *type graph* G_c is defined as the structure (V_c, E_c), where V_C is a set of nodes representing types (C and all the types of the fields reachable from C), and E_c is the set of edges representing the reachable fields, i.e., for each field f of type T in a class C_i, there is an arc in

```
public class LinkedList<E> extends
    AbstractSequentialList<E> implements ... {
  int size = 0;
  Node<E> first, last;
  ...
  private static class Node<E> {
    E item;
    Node<E> next, prev;
    ...
```

Fig. 1: LinkedList class from the java.util package.

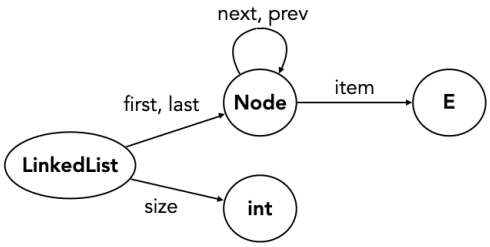

Fig. 2: Type graph of the LinkedList class.

the graph going from the node representing C_i to the node representing T.

As an example, consider the `LinkedList` class from the java.util package shown in Figure 1. The class declares three fields: `first` and `last` of type `Node`, and `size` of type `int`. Additionally, the inner class `Node` also declares three fields: `next` and `prev` of type `Node`, and `item` of the generic type `E`. Figure 2 shows the type graph for the `LinkedList` class. This graph contains four nodes, one per each reachable type (`LinkedList`, `Node`, `int` and `E`). Then, for each of the mentioned fields, there is an arc connecting the nodes representing the corresponding types. For this example, fields `first`, `last` and `size` are non-iterable, while `next`, `prev` and `item` are iterable, since the first two are recursive, and the last belongs to a class with a recursive field. Notice that these fields are considered iterable as they allow one to iterate over the nodes and values of a linked list. In fact, at run time, they can be considered to lead to the definition of sets of elements of type `Node` and `E`, respectively, e.g., all the nodes reachable through `next` (resp. `prev`) from a given node, or all items obtained from the nodes through field `item`.

B. State Field Coverage

To provide a formal definition of the state field metric, we define the concepts of *coverable and covered labels*, i.e., the set of target labels that an oracle may cover, and the subset of these that the oracle actually covers, respectively.

Definition 2: Let C be the target class and G_c its type graph. We define the set L_c of *coverable labels* as $E_c \cup S$, where E_c is the set of edges (fields) in the type graph G_c and S is the set of special labels, defined as $S = \{f+ \mid f \in I_c\}$, computed from the set I_c of iterable fields.

Basically, the set of coverable labels L_c includes a label f for each field $f \in F_c$, and a special label $f+$ for each iterable field $f \in I_c$. For the `LinkedList` example, the set of coverable labels will contain the field labels $\{first, last, size, next, prev, item\}$, and the special labels $\{next+, prev+, item+\}$ corresponding to the iterable fields `next`, `prev` and `item`.

```
public boolean isEmpty() {
  return size == 0;
}
```

(a) Method checking if the list is empty.

```
public boolean checkSize() {
  if (first == null) return size == 0;
  Set<Node> visited = new java.util.HashSet<>();
  visited.add(first);
  Node<E> current = first;
  while (current != null && visited.add(current)) {
    current = current.next;
  }
  return visited.size() == size;
}
```

(b) Method checking consistency between the list size and the number of nodes in the list.

Fig. 3: Two methods over the `LinkedList` class with different state field coverage.

Definition 3: Let p be a program taking as input an object of class C. We say that a label $l \in L_c$, corresponding to field f, is covered by p, if p accesses field f. A label $l+ \in L_c$, corresponding to an iterable field f, is covered by p if p iterates over the elements obtained through f. The *covered labels* associated with p is the set of all labels covered by p.

In the above definition, the covered labels are defined for an arbitrary program p. For the context of this paper, the program p will always represent an *oracle*, e.g., the statements that are called, directly or indirectly, within oracle assertions. Given an oracle assertion, its state field coverage is the proportion of coverable labels that are actually covered by the oracle.

Definition 4: Let ϕ be an oracle defined for class C, i.e., ϕ: $C \rightarrow Bool$. The *state field coverage* SFC_ϕ of ϕ is defined as the proportion of coverable labels L_c that are covered by ϕ, i.e.,

$$SFC_\phi = \frac{|L_\phi|}{|L_c|} \quad (1)$$

For example, consider the two methods in Figure 3 defined over `LinkedList`, and assume that these are called within respective test assertions. The first method checks if the list is empty, accessing only the `size` field. Therefore, it only covers the label $size$ out of 9 coverable labels, which leads to a state field coverage of 11.1%. The second method, on the other hand, checks that the size of the list is consistent with the number of nodes in the list, by accessing the `size`, `first` and `next` fields, and also *iterating* over the `next` field. Thus, it covers 4 labels ($size$, $first$, $next$ and $next+$), achieving an object state coverage of 44.4%.

Although state field coverage is defined for oracles predicating over a single class, it can be easily extended to oracles

predicating over multiple classes. In such cases, the coverable labels L_c will be the union of the coverable labels for each class, and the labels L_ϕ covered by an oracle ϕ will be the union of the covered labels in each class.

C. Implementation

To measure the state field coverage of an oracle, we implement a static analysis approach. Our implementation is for Java. The process takes as input a target class C and the source code of an oracle ϕ_c, and computes the oracle's state field coverage according to Definition 4.

Type Graph Generation: Given a (root) Java class C, we generate the type graph $G_c = (V_c, E_c)$ by first initializing the set of nodes V_c with a node for C, the target class, and then recursively adding edges and nodes for the reachable fields and classes, respectively. In our implementation, this process is performed in a depth-first fashion, and the type graph is built using the jgrapht [3] library.

Coverable labels: The set L_c of coverable labels is straightforwardly computed from the type graph G_c. Besides each edge in the graph being a label in L_c, we consider labels for iterable fields, based on the following two cases:

- For every edge $e = T_1, T_2$ such that T_2 is a class that implements the `Iterable` interface, or is an array type, e is deemed iterable, and we add label l_e+ to L_c,
- For every edge $e = T_1, T_2$ such that T_1 participates in a loop path (a non-empty graph path starting and ending in the same node) within G_c, e is deemed iterable, and we add label l_e+ to L_c.

Covered labels: The labels L_ϕ that are covered by an oracle ϕ_c are obtained by parsing the source code of ϕ_c, and identifying the fields accessed by the oracle, i.e., from expressions or methods called within (test) assertions. More precisely, the following two cases are considered:

- a label $l \in L_c$ is considered covered if there exists a statement reachable from ϕ_c's source code that accesses the field corresponding to l,
- a special label $l+ \in L_c$ is considered covered if there exists a statement in a loop body (e.g., the body of a `for` or `while` statement) reachable from ϕ_c's source, that accesses the field corresponding to $l+$.

Oracle source code parsing and analysis is implemented using the JavaParser [2] library. Our current implementation is specific to Java, as it relies on the Java syntax and type system. Since our coverage approach assesses how thoroughly the oracles evaluate the SUT's state definition, its implementation needs to take into account the mechanisms that the programming language provides for data representation. Most programming languages provide means to define custom datatypes, and these lead straightforwardly to notions of type graphs, similar to what we have described above for Java. Although our implementation computes our coverage metric for Java, adapting the process to other programming languages and datatype definition mechanisms is relatively direct.

Class	#Properties	#Rep. Invariants
SinglyLinkedList	3	7
SortedList	4	15
DoublyLinkedList	3	7
BinaryTree	3	7
SearchTree	4	15
RedBlackTree	5	31
HeapArray	4	15
BinomialHeap	5	31
DisjSet	4	15
FibonacciHeap	7	127
DAG	2	3
Total	44	273

(a) Representation Invariants.

Project	#Classes	#Tests	#Assertions
Chart	29	6,557	27,301
Cli	6	1,008	1,347
Closure	47	25,150	27,317
Codec	3	2,219	5,396
Collections	6	8,160	9,764
Compress	11	2,548	5,834
Csv	13	840	2,748
Gson	3	3,097	5,842
JacksonCore	33	1,738	10,181
JacksonDatabind	23	6,439	21,756
JacksonXml	6	494	1,796
Jsoup	7	2,056	6,287
JxPath	9	1,152	2,052
Lang	3	6,737	38,774
Math	6	13,010	25,544
Mockito	4	4,064	5,364
Time	6	11,998	51,724
Total	215	97,267	249,027

(b) Test Assertions

Fig. 4: Distribution of the target representation invariants from Korat (a) and the target test assertions from the 51 project versions of the Defects4J benchmark (b) used in the evaluation.

IV. Evaluation

Our evaluation of state field coverage is organized around the following research questions:

RQ1 *Is state field coverage correlated with fault detection?*
RQ2 *Can state field coverage be used for oracle improvement?*
RQ3 *Can oracle improvement based on state field coverage help real bug detection?*
RQ4 *How efficiently can state field coverage be computed?*

RQ1 analyzes the correlation between state field coverage and the ability of the oracles to detect faults, measured as the detection of mutants. RQ2 focuses on evaluating how the state field coverage metric can be used to guide the improvement of oracles; we analyze how the ability of test suites to detect artificial faults varies as tests with increasingly larger state field coverage are incorporated, comparing it with fault detection when such tests are randomly added. RQ3 evaluates the impact of state field coverage in detecting real faults, through an experiment similar to the one used for RQ2, but on real regression faults. Finally, RQ4 evaluates the efficiency with which the state field coverage metric can be computed.

A. Evaluation Subjects

In our evaluation, we use two types of oracles: *representation invariants* and *test assertions*. The representation invariants are taken from the Korat distribution [12], which provides 11 Java classes implementing data structures (e.g., linked lists, trees, and graphs) along with their invariants. These invariants check properties ranging from basic (e.g., no non-null values) to complex (e.g., cyclicity/acyclicity). Invariants are typically the conjunction of various properties that can be checked independently. We thus decompose each invariant into its corresponding individual properties, and consider subsets of the invariant as alternative (weaker) invariants of the same class. This yields a total of 273 distinct oracles for the Korat classes. Figure 4a summarizes the invariants used, including property counts and target invariants per class.

The test assertions in our evaluation are taken from the Defects4J benchmark [26] (version 2.0.1), providing us with developer-written test assertions for real-world projects. Due to the computational cost of mutation analysis, we restrict our evaluation to test assertions from the fixed versions of the three most recent bugs in each of the 17 Defects4J projects, leading to a total of 51 versions. For each version, we consider the modified classes (and their dependencies) as target classes, and all corresponding test assertions as target oracles. Figure 4b summarizes the distribution of test assertions, showing, per project, the sum (across the three versions) of target classes, tests, and individual assertions.

To evaluate the correlation between state field coverage and fault detection (RQ1), we use 273 representation invariants from Korat and 17 project versions (one per project) from Defects4J, comprising 83,032 test assertions. The latter subset consists of the latest version of each project, ensuring a representative sample while reducing mutation analysis computational costs compared to analyzing all 51 versions.

For RQ2, RQ3, and RQ4, we focus on test assertions, as they better reflect real-world oracles. Since RQ2 involves mutation analysis, we reuse the same 17-project subset from RQ1. For RQ3 and RQ4, we analyze all 51 project versions to study the relationship between state field coverage and real faults, as well as the efficiency of our implementation to compute our metric.

B. Experimental Setup

In this section we describe how we compute the metrics involved in our experiments.

State Field Coverage: We compute state field coverage using the static analysis implementation from Section III-C. For representation invariants, the type graph is derived from the target class (the class the invariant corresponds to). State field coverage is computed per invariant, based on its corresponding source code. For test assertions, the type graph is computed from classes modified in the bug-fixed version (as provided by Defects4J [1]). This choice has some advantages: Defects4J mutation analysis generates mutants for these classes, facilitating our evaluation and reducing the computational cost, compared to considering *all* classes of the corresponding projects. Additionally, it provides us with an unbiased criterion to select the target classes. State field coverage is computed per test, aggregating all assertions in the test. For each assertion, we inspect its code and compute covered labels, both those directly accessed and those accessed via invoked methods.

Our current implementation does not track field accesses that indirectly influence assertion parameters (e.g., via earlier statements or method calls). Detecting such cases would require more complex information flow analysis, which we leave for future work.

Mutation Analysis: To assess fault detection in RQ1 and RQ2, we employ mutation analysis as follows. For the representation invariant oracles (Korat subjects), mutants are generated using PIT [14] over each of the classes, excluding the invariants themselves (invariants are the oracles, not the SUT, and thus are not mutated). We obtained 26 mutants per class, on average. Each mutant is evaluated using a Randoop [40] generated test suite (up to 100 tests per class), with each test invoking the target invariant as test oracle. We favored the use of Randoop over EvoSuite [17] because of two reasons: EvoSuite's test generation is guided by mutation (among other metrics), and thus introduces a bias in the generated tests in relation to mutation analysis; also, EvoSuite aims to minimize the number of generated tests, resulting in too small test suite samples for our experiments.

For the test assertions from Defects4J projects, mutants are generated using Major [28] (on average, $\sim$1,593 mutants per project), taking advantage of the framework's support for mutation analysis. Mutation scores are computed per-test using the test suites available with the projects (on average, $\sim$1,572 tests per project). Mutation score is computed as the percentage of mutants killed by the oracles, excluding trivial mutants triggering runtime exceptions before oracle execution.

Checked Coverage: Since the checked coverage metric targets test assertions, we compute it for the Defects4J test assertions analyzed in RQ2. We were unable to use the original implementation [43] due to Java version incompatibilities between the slicer used and Defects4J. Instead, we rely on a re-implementation from [29], which is based on Slicer4J [4]. As discussed in the results, computation failed for some projects due to implementation errors.

Workstation: All the experiments described in this paper were run on a workstation with a Xeon Gold 6154 CPU (3GHz), 128 GB of RAM, running Debian GNU/Linux 12.

C. Correlation with Fault Detection (RQ1)

We first analyze state field coverage and mutation scores for both representation invariants and test assertions. Figure 5 shows the results of these metrics for each oracle type, including checked coverage for test assertions. Notice that this figure shows the overall distribution of the values obtained for each metric considering all projects and whole test suites. As mentioned earlier, for this RQ, we evaluate a subset of 17 project versions containing 83,032 test assertions; reported values on test assertions correspond to this subset.

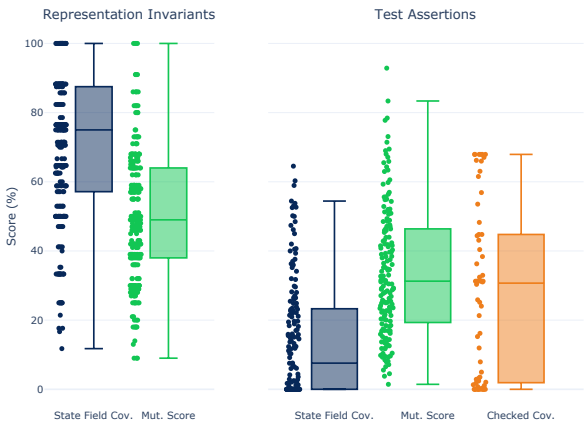

Fig. 5: Distribution of the obtained State Field Coverage and Mutation Score for each type of oracle, including Checked Coverage for test assertions.

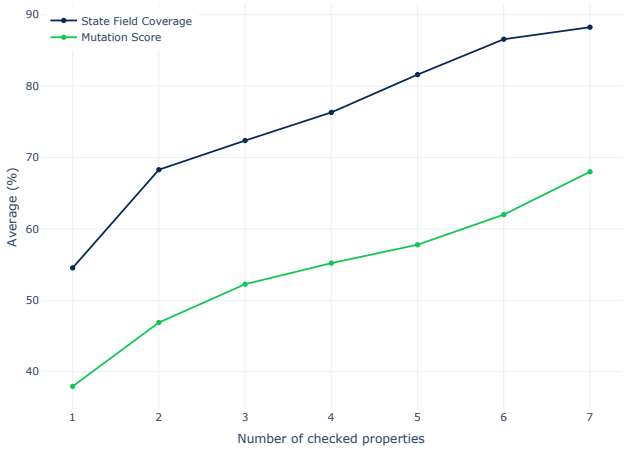

Fig. 6: State Field Coverage and Mutation Score as the number of properties in representation invariants increases.

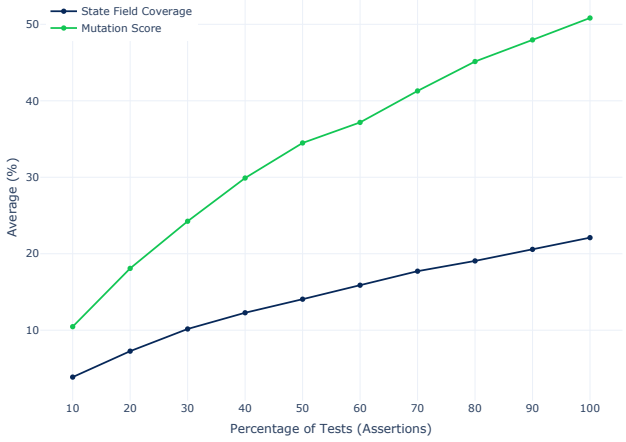

Fig. 7: State Field Coverage and Mutation Score for increased percentages of selected tests and assertions.

For the 273 representation invariants analyzed, state field coverage ranges from 11.7% to 100% (avg. 70.3%), while mutation scores range from 9% to 100% (avg. 50.2%). The high state field coverage is expected for invariants, since these assertions explicitly check object properties, requiring access to most state fields. Indeed, 50% of the invariants achieve over 75% state field coverage.

The 83,032 test assertions show state field coverage ranging from 0% to 64.5% (avg. 14.3%), significantly lower than invariants, as assertions typically verify method outputs corresponding to specific test inputs, rather than thoroughly inspecting object state. Over 75% of assertions fall below 22% state field coverage. Mutation scores range from 2.5% to 92.8% (avg. 34.1%), while checked coverage spans 0%–68% (avg. 27.8%).

To assess how state field coverage correlates with fault detection, we examine how increasing oracle quantity affects both state field coverage and mutation score. For representation invariants, we track these metrics as invariants grow more complex (checking additional properties). For test assertions, we evaluate them as test suites expand (adding more tests/assertions).

1) Correlation for Representation Invariants: Figure 6 shows how average state field coverage and mutation score increase with the number of properties checked by representation invariants. That is, we group invariants by the number of properties they check, and compute the average state field coverage and mutation score for each group. Both metrics exhibit similar growth trends as invariants become more complex. We quantify this relationship using Pearson correlation, which measures linear dependence between variables (ranges from -1 to 1, with values greater than zero indicating positive correlation). For our dataset, we find a coefficient of 0.54, indicating a high positive correlation between state field coverage and mutant detection.

2) Correlation for Test Assertions: Figure 7 presents the relationship between test suite size (percentage of selected tests/assertions) and both state field coverage and mutation score. For each project version, we randomly select increasing percentages of tests, and compute the state field coverage and mutation score for the selected tests. Results are averaged across all 17 project versions. To account for selection randomness, each data point represents the average of 100 runs. While we analyzed state field coverage across all 17 project versions, `Collections-28` (version id 28 of the Collections project) was excluded from mutation analysis due to a Defects4J framework error that prevented score computation.

The state field coverage exhibits slower but consistent growth compared to the mutation score, as test suites grow. While their growth rates differ more markedly than with representation invariants, both metrics increase with additional tests. Calculating the Pearson correlation coefficient (excluding projects with 0% state field coverage) yields $\sim$0.45, confirming a moderate positive correlation.

For a more detailed analysis, Figure 8 shows per-project correlations between average state field coverage (x-axis)

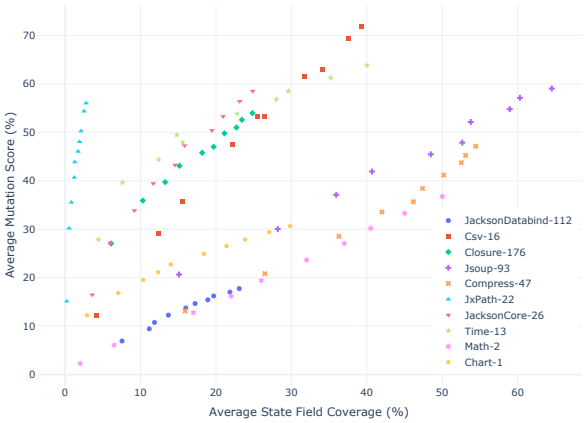

Fig. 8: State Field Coverage and Mutation Score correlation for each percentage of selected tests of each project version.

and average mutation score (y-axis) across test suite sizes. We excluded six projects (`Cli-40`, `Codec-18`, `Gson-16`, `JacksonXml-6`, `Lang-4`, and `Mockito-22`) where target classes were stateless (resulting in 0% state field coverage). As discussed in Section V, stateless classes lead to type graphs with no edges to cover, and thus low state field coverage does not necessarily reflect assertion limitations.

The data shows a strong positive correlation between state field coverage and mutation score, as evidenced by the consistent trend where increased state field coverage improves fault detection. This relationship is particularly robust, with Pearson coefficients exceeding 0.96 for most projects. The sole exception is `JxPath-22`, where maximum coverage plateaus at 2.78% due to tests overall covering a small number of labels. In this case, additional tests cannot significantly improve state field coverage, allowing mutation scores to increase independently. These results demonstrate that state field coverage effectively predicts oracle quality for both representation invariants and test assertions, showing consistent correlation with the established mutation score metric.

D. Oracle Improvement (RQ2)

To assess how state field coverage can guide oracle improvement, we conduct an experiment using test assertions. We simulate extending an initial single-test suite by incrementally adding tests whose oracles maximize uncovered state field labels, comparing the resulting mutation score against that of random test selection (averaged over 10 iterations to account for randomness). Since this experiment also involves mutation analysis, we use the same 17 project versions from RQ1. When possible, we simulate oracle guided test suite extension based on checked coverage, reporting the results for cases where this metric was successfully computed (projects `Csv-16`, `Jsoup-93`, and `Time-13` out of all the projects in this experiment). Note that `Collections-28` had to be excluded due to a Defects4J error, and other six projects are disregarded due to the corresponding modified classes being stateless. Additionally, we incorporate as a reference a test suite extension strategy based on optimizing statement coverage. Notice that this strategy chooses as new tests to extend the suite those that maximize covering previously uncovered statements, without considering whether these newly covered statements are called from oracles or not. Thus, it is not an oracle guided strategy, but serves as a reference with traditional test prioritization strategies.

Figure 9 shows the experimental results across all projects, plotting mutation score (y-axis) against number of selected tests (x-axis). The graph compares four test selection strategies: state field coverage maximization (blue line), random selection (green line), checked coverage maximization (orange line, where available), and statement coverage maximization (red line). We analyze each strategy's effectiveness below.

1) Statement Coverage-based Selection: Increasingly selecting tests that maximize statement coverage leads to higher mutation scores compared to the other strategies in most of the project versions. This is expected, as improving statement reachability improves mutant detection, and this approach does not focus on the oracles, but on the statements covered by the unit tests as a whole. The purpose of including this strategy is to provide a reference point for comparison, as is not directly related to oracle quality. As we discuss below, state field coverage-based selection is in general superior to the random and checked coverage-based selection approaches.

2) State Field Coverage-based Selection: Several project versions demonstrate significantly higher mutation scores when extending test suites via state field coverage maximization versus random selection. In `Chart-1`, `Math-2`, `Compress-47`, and `JacksonCore-26`, this strategy detects up to 20% more faults initially. The mutation score improvement remains substantial even at scale, e.g., for `Chart-1`, `Compress-47`, and `JacksonCore-26` with 300 tests, and for `Math-2` with 2,000 tests.

Figure 10 illustrates a representative test from `Math-2` that was prioritized by our state field coverage heuristic. This single test achieves a 9.7% mutation score (30% of the suite's total 31.08%) while covering 50% of target class labels with just four assertions. Similar patterns emerge in other projects: in `Chart-1`, the first test yields ∼3% mutation score, increasing to ∼10% with 10 tests and ∼20% with 100 tests, nearing the maximum ∼30%. For `Compress-47`, the first ten tests achieve ∼16.3% mutation score, a notable efficiency given the suite's 800+ tests and maximum ∼34.8% score.

In several project versions (`JxPath-22`, `JacksonDatabind-112`, `Time-13`, `Csv-16`, and `Jsoup-93`), our state field coverage strategy yields mutation scores comparable to random selection. These projects generally exhibit low state field coverage (e.g., below 25% for `JxPath-22` and `JacksonDatabind-112`), suggesting that our metric provides limited benefit when state field coverage is low. However, for projects like `JacksonDatabind-112` and `Csv-16`, our strategy achieves better initial mutation scores. The `Closure-176` project is unique in showing significantly worse performance with our strategy compared to random selection. This aligns

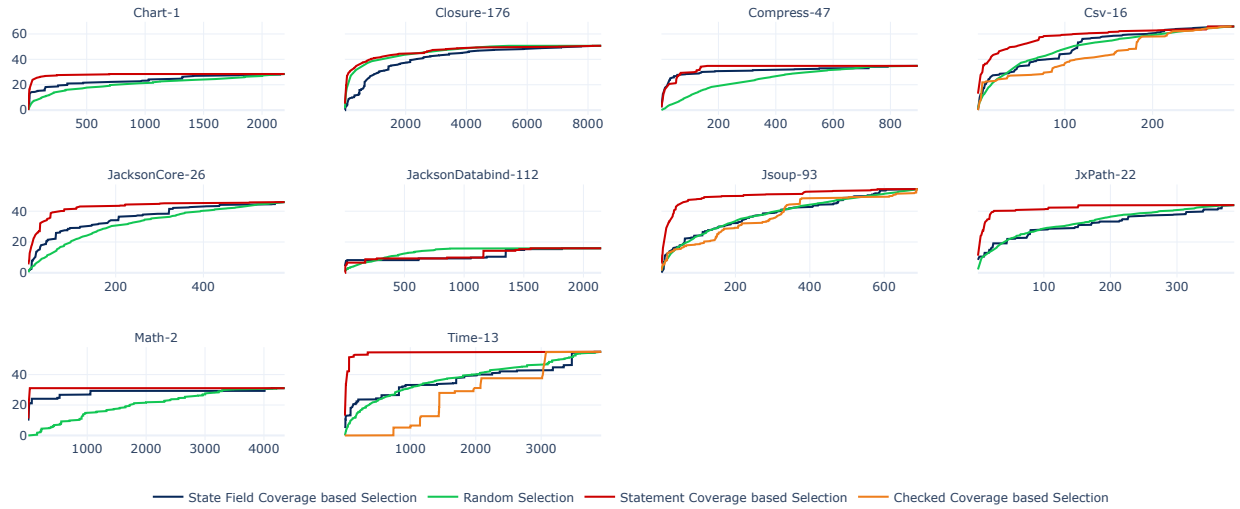

Fig. 9: Mutation score achieved by selecting tests that increasingly improve State Field Coverage, compared to selecting tests randomly. A selection based on improving Checked Coverage is also included when possible. The x axis represents the number of selected tests, and the y axis represents the mutation score achieved by the tests. Random selection is iterated 10 times.

```
@Test
public void testMoments() {
  final double tol = 1e-9;
  HypergeometricDistribution dist;

  dist = new HypergeometricDistribution(1500, 40,
    100);
  Assert.assertEquals(dist.getNumericalMean(), 40d *
     100d / 1500d, tol);
  Assert.assertEquals(dist.getNumericalVariance(), (
     100d * 40d * (1500d - 100d) * (1500d - 40d) ) /
     ( (1500d * 1500d * 1499d) ), tol);

  dist = new HypergeometricDistribution(3000, 55,
    200);
  Assert.assertEquals(dist.getNumericalMean(), 55d *
     200d / 3000d, tol);
  Assert.assertEquals(dist.getNumericalVariance(), (
     200d * 55d * (3000d - 200d) * (3000d - 55d) ) /
     ( (3000d * 3000d * 2999d) ), tol);
}
```

Fig. 10: Test from `Math-2` with a 50% state field coverage and 9.7% mutation score.

with its very low state field coverage (∼25%), among the poorest of all projects.

These results demonstrate that state field coverage can effectively guide oracle improvement, particularly when high state field coverage is achievable. Furthermore, our metric's ability to identify uncovered state portions directly supports targeted assertion generation for improved fault detection.

3) Checked Coverage-based Selection: Figure 9 includes checked-coverage based selection results for `Time-13`, `Csv-16`, and `Jsoup-93`, the only projects where both checked coverage and state field coverage could be computed. The analysis shows that for `Csv-16` and `Jsoup-93`, checked coverage yields comparable mutation scores to state field coverage in early test selection phases, while `Time-13` shows inferior performance for checked coverage compared to the other strategies. While state field coverage generally outperforms checked coverage in our experiments, more extensive studies are required to validate this observed trend, and better characterize the relationship between these metrics.

Finally, we analyze whether the effect of state field coverage based selection is due to indirectly improving code coverage, or not. We build test suites where all tests have very similar statement coverage, selecting the maximum subset of tests where the difference in statement coverage among any two tests is at most 10%. Then, we perform state field coverage-based and random selections from these subsets. To compare these strategies, we compute the progression of the APFD metric [16], which measures a weighted average of the percentage of faults detected, for increasingly larger subsets (10%, 20%, and so on), of the suite with similar statement coverage. The results of this experiment are shown in Table I.

Notably, for smaller percentages of selected tests (10% to 40%), state field coverage selection considerably outperforms random selection in most of the project versions, obtaining a higher APFD in at least 8 out of 10 project versions. This indicates that, for the same level of code coverage, selecting tests that improve oracles according to state field coverage leads to better fault detection than selecting tests randomly. As the percentage of selected tests increases, the advantage of state field coverage-based selection diminishes, but still outperforms random selection in most projects. These results show that our metric provides benefits beyond code coverage, and can guide test selection for improved fault detection.

It is worth remarking that, in some cases, the APFD progression for greater suite subsets can decrease. For instance, in `Chart-1`, the APFD progression for state field coverage

TABLE I: Weighted Average of the Percentage of Faults Detected (APFD) by the State Field Coverage-based Selection (SFC) and Random Selection (Random). For each target project, we report the APFD values achieved by each strategy considering a percentage of selected tests. Green SFC cells indicate that SFC outperforms random selection, red cells otherwise.

Subject	Technique	APFD by Test suite percentages									
		10	20	30	40	50	60	70	80	90	100
Chart-1	SFC	67.75	68.58	76.95	79.79	83.3	37.46	46.25	52.96	57.8	61.26
	Random	81.98	67.37	72.49	78.88	82.55	83.73	83.16	44.72	50.25	55.09
Closure-176	SFC	65.36	75.55	79.31	81.81	83.47	78.45	78.91	79.96	81.14	80.92
	Random	63.82	76.2	81.06	81.74	84.02	84.43	85.49	85.04	85.25	86.08
Compress-47	SFC	62.16	60.71	63.41	67.53	67.69	70.43	74.91	75.93	77.48	78.27
	Random	31.25	40.86	57.73	60.34	51.54	54.18	54.51	54.22	58.35	57.75
Csv-16	SFC	74.84	77.61	73.82	73.57	66.76	68.59	71.83	72.5	73.76	75.11
	Random	52.79	65.9	71.51	72.39	75.07	75.1	76.11	76.33	76.21	77.48
JacksonCore-26	SFC	64.7	76.88	77.13	74.3	74.61	77.09	77.04	77.97	79.89	79.74
	Random	52.75	61.03	62.47	65.74	67.07	65.35	66.46	67.44	70.24	71.84
JacksonDatabind-112	SFC	98.74	94.7	96.46	66.67	70.45	75.37	73.66	76.94	74.61	77.16
	Random	67.7	60.6	61.41	56.82	58.07	60.6	57.28	62.66	64.74	68.29
Jsoup-93	SFC	69.96	72.65	64.08	63.24	66.97	70.06	69.55	71.84	72.47	73.97
	Random	51.54	59.41	70.73	70.96	70.2	70.96	71.88	72.15	74.33	74.68
JxPath-22	SFC	66.45	80.81	83.44	87.07	73.54	70.6	74.86	73.83	70.13	69.01
	Random	56.53	75.39	76.21	70.77	73.34	76.69	78.97	72.31	74.3	76.9
Math-2	SFC	94.51	97.25	98.17	88.21	90.02	91.69	83.66	85.71	85.72	87.14
	Random	7.01	46.06	58.63	48.69	44.11	46.69	54.31	60.02	63.27	65.45
Time-13	SFC	38.42	69	79.2	76.79	80.63	83.8	81.09	83.41	85.28	86.78
	Random	57.03	68.55	77.49	83.22	86.63	83.7	86.07	75.46	57.42	58.54
Times SFC > Random:		**8**	**9**	**8**	**8**	**6**	**5**	**4**	**7**	**6**	**6**

drops from 83.3 to 37.46, when growing from 50% to 60%. While this may be counter intuitive, it is indeed possible. This effect is due to new tests being added, that kill new mutants, but do so with the latest tests in the selection order, thus causing the average percentage of faults detected to drop. This issue is also observed for random selection, where the APFD drops from 83.14 to 44.72 when moving from 70% to 80%.

E. Real Fault Detection (RQ3)

To evaluate our metric's impact on real fault detection, we conduct the following experiment using Defects4J buggy versions with failing tests. We measure how many test executions are needed to trigger a bug under two strategies: *state field coverage-based execution*, where tests are ordered to maximize state field coverage growth (as in RQ2), and *random execution*, where tests are selected randomly (averaged over 10 runs). From the initial 51 project versions, 38 (75%) have non-empty type graphs (enabling state field coverage computation), 27 of which (71% of 38) yield a positive state field coverage. We focus our analysis on these 27 versions, since as discussed in Section V, our metric cannot be computed for stateless classes.

Table II presents the results comparing test selection strategies. For each project and bug id, we show the total tests available, and the number of tests needed to first trigger the bug using state field coverage ordering, and random ordering. The state field coverage ordering outperformed random selection in 17/27 cases (63%), requiring 34.7× fewer tests on average. In the remaining 10 cases, random selection performed modestly better (1.8× fewer tests). This demonstrates our metric's potential to significantly improve fault detection efficiency by prioritizing tests more likely to reveal bugs.

TABLE II: Test cases needed to first trigger Defects4J bugs, when selecting tests based on State Field Coverage, and on Random selection.

Project	Bug ID	#Tests	Tests needed to trigger the bug	
			State Field Cov.	Random
Chart	1	2,193	4	781.5
Chart	3	2,187	16	1,423.9
Cli	38	317	56	97.8
Closure	174	8,308	1,337	2,081.3
Closure	175	8,410	231	1,098.4
Closure	176	8,432	5,895	4,579.4
Collections	26	2,720	171	1,364.3
Compress	46	829	2	506.9
Compress	47	895	32	461.6
Csv	14	257	97	33.5
Csv	15	290	212	190
Csv	16	293	88	114.6
JacksonCore	26	585	275	333.7
JacksonCore	25	573	57	285.8
JacksonCore	24	580	109	58.6
JacksonDatabind	111	2,146	621	946.4
JacksonDatabind	112	2,148	2,145	973.5
Jsoup	92	689	317	135.8
Jsoup	93	690	309	377.5
JxPath	21	384	76	144.6
JxPath	22	386	347	168.7
Math	1	4,378	185	1,252.2
Math	2	4,350	3,000	2,463.4
Mockito	1	1,370	49	60.2
Time	1	4,041	2,258	1,696.8
Time	2	4,041	1,031	1,923.2
Time	13	3,916	3,705	2,035.2
Summary		Times better:	**17**	10
		Average improvement:	**34.7x**	1.8x

F. Efficiency (RQ4)

Finally, we evaluate the execution time required to (statically) compute state field coverage, and compare it with mutation analysis and checked coverage. Our evaluation in-

volves test assertions from the Defects4J projects. For the 17 projects analyzed, state field coverage computation required an average of 5,164.0 seconds (∼1.4 hours) per project, totaling 82,624.2 seconds (∼22 hours) for all 83,032 test assertions (∼6 seconds per test). This contrasts with mutation analysis, which averaged 72,902.9 seconds (∼20 hours) per project and exceeded 300 hours (∼12.5 days) total due to its inherently dynamic nature. Similarly, checked coverage (measured for the 6 projects for which the checked coverage tool could be run) averaged 20,327.3 seconds (∼5.6 hours) per project, totaling 121,964.1 seconds, with its dynamic slicing process contributing to the higher computation time. These results demonstrate that static field coverage provides rapid, practical feedback on oracle quality, enabling efficient preliminary assessment before committing to more computationally expensive analyses like mutation analysis.

V. Limitations

Our metric requires the target class to be stateful, containing fields accessed by the class methods. Stateless classes (e.g., `Cli-40`, `Codec-18`, `Gson-16`, and `Mockito-22`) with no fields and only static methods yield empty type graphs and consequently 0% state field coverage. However, this limitation affects only a minority of cases: 38 of the 51 analyzed project versions (75%) contained stateful classes with at least one field, demonstrating our technique's broad applicability. Also, our current metric definition considers only the target class's fields and their direct and indirect dependencies, but excluding types returned by methods. Consequently, assertions that only verify method return values yield 0% state field coverage, limiting the metric's ability to assess such oracles. We plan to extend the metric to include return type fields, which would both address this limitation and resolve the stateless class issue, by analyzing returned objects' state. This enhancement, part of our future work, would further broaden the metric's applicability to more oracle types and target classes.

Another limitation of our approach is the potential infeasibility of state fields as test requirements. Unlike mutation testing, where infeasible requirements (e.g., equivalent mutants) are inherent, state field coverage infeasibility would correspond to fields that are unreachable from test oracles. Such infeasibility highlights limitations in the testability of the system under test (e.g., poorly exposed state), rather than a flaw in the metric itself. This behavior is analogous to unreachable code in statement coverage metrics and reflects useful diagnostic information rather than a weakness.

VI. Threats To Validity

A threat to external validity stems from our use of a Defects4J subset rather than all available projects, limited by mutation analysis costs. However, we included a representative variety of 215 target classes. Implementation issues with checked coverage also constrained our comparison, despite best efforts to use available implementations.

For internal validity, potential threats include: *(1)* our static implementation may conservatively include field access paths not executed at runtime, potentially inflating metric values; *(2)* the observed correlation between state field coverage and mutation score might indirectly stem from improved code coverage; and *(3)* stochastic variations in our experiments could inadvertently bias results. To address these threats, we implemented the following mitigation strategies. For *(1)*, we developed a dynamic variant of state field coverage and compared it against our static metric. The two agreed exactly in 40% of cases and differed by $<$10% in the remaining 60%, demonstrating strong alignment. For *(2)*, we conducted controlled experiments by clustering tests with identical statement coverage, then rerunning the state field coverage and random selection strategies within these clusters. The results reaffirmed our original correlation findings, isolating the effect of state field coverage. For *(3)*, we repeated randomized trials and manually verified outcomes to minimize chance effects. Summaries of the additional experiments can be found as part of our replication package.

VII. Conclusion and Future Work

Testing effectiveness ultimately resorts on oracle quality. Existing techniques, such as mutation analysis, checked coverage, and oracle deficiency, have advanced oracle assessment but are computationally costly and often provide indirect, hard-to-act-on feedback. We introduced *state field coverage*, a novel metric for assessing oracle quality based on the premise that oracles referencing more of the SUT's state definition are more effective at detecting faults. Our static approach to compute this metric addresses a key limitation of dynamic techniques, by avoiding their computational costs while maintaining strong correlation with fault detection (as validated through mutation analysis). Unlike existing methods, our metric directly identifies uncovered state elements, providing developers with actionable insights for oracle improvement. Future work includes extending state field coverage to additional oracle forms (e.g., properties in property-based testing [13]), broadening the empirical evaluation, and integrating state field coverage as a fitness signal in evolutionary test generation (e.g., EvoSuite [17]) to favor tests whose assertions more thoroughly predicate on program state.

VIII. Data Availability

Our current state field coverage implementation as well as the scripts and data required to reproduce our experiments are publicly available in our replication package [5].

Acknowledgments

This work is supported by the Ramón y Cajal fellowship RYC2020-030800-I, by the Spanish Government through grants TED2021-132464B-I00 (PRODIGY), PID2022-142290OB-I00 (ESPADA), and CEX2024-001471-M/funded by MICIU/AEI/10.13039/501100011033 and by the Comunidad de Madrid as part of the ASCEND project co-funded by FEDER Funds of the European Union.

REFERENCES

[1] Defects4j framework. https://github.com/rjust/defects4j, 2025.

[2] Javaparser. https://javaparser.org/, 2025.

[3] Jgrapht. https://jgrapht.org/, 2025.

[4] Slicer4j. https://github.com/resess/Slicer4J, 2025.

[5] State field coverage implementation and replication package. https://zenodo.org/records/17255287, 2025.

[6] Juan C. Alonso, Sergio Segura, and Antonio Ruiz-Cortés. AGORA: automated generation of test oracles for REST apis. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 1018–1030. ACM, 2023.

[7] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[8] Jon Ayerdi, Valerio Terragni, Gunel Jahangirova, Aitor Arrieta, and Paolo Tonella. Genmorph: Automatically generating metamorphic relations via genetic programming. *IEEE Trans. Software Eng.*, 50(7):1888–1900, 2024.

[9] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Trans. Software Eng.*, 41(5):507–525, 2015.

[10] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 242–253. ACM, 2018.

[11] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Antonio Carzaniga. Memo: Automatically identifying metamorphic relations in javadoc comments for test automation. *J. Syst. Softw.*, 181:111041, 2021.

[12] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In Phyllis G. Frankl, editor, *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, pages 123–133. ACM, 2002.

[13] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000.

[14] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: a practical mutation testing tool for java (demo). In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 449–452. ACM, 2016.

[15] Hang Du, Vijay Krishna Palepu, and James A. Jones. To kill a mutant: An empirical study of mutation testing kills. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 715–726. ACM, 2023.

[16] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, February 2002.

[17] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 416–419. ACM, 2011.

[18] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 147–158. ACM, 2010.

[19] Aayush Garg, Renzo Degiovanni, Facundo Molina, Maxime Cordy, Nazareno Aguirre, Mike Papadakis, and Yves Le Traon. Enabling efficient assertion inference. In *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023, Florence, Italy, October 9-12, 2023*, pages 623–634. IEEE, 2023.

[20] Ishrak Hayet, Adam Scott, and Marcelo d'Amorim. Chatassert: Llm-based test oracle generation with external tools assistance. *IEEE Trans. Software Eng.*, 51(1):305–319, 2025.

[21] Soneya Binta Hossain and Matthew B. Dwyer. TOGLL: correct and strong test oracle generation with LLMS. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, pages 1475–1487. IEEE, 2025.

[22] Soneya Binta Hossain, Antonio Filieri, Matthew B. Dwyer, Sebastian G. Elbaum, and Willem Visser. Neural-based test oracle generation: A large-scale evaluation and lessons learned. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 120–132. ACM, 2023.

[23] Chen Huo and James Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 621–631. ACM, 2014.

[24] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. Test oracle assessment and improvement. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 247–258. ACM, 2016.

[25] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. An empirical validation of oracle improvement. *IEEE Trans. Software Eng.*, 47(8):1708–1728, 2021.

[26] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In Corina S. Pasareanu and Darko Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440. ACM, 2014.

[27] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 654–665. ACM, 2014.

[28] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: an efficient and extensible tool for mutation analysis in a java compiler. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 612–615. IEEE Computer Society, 2011.

[29] Roxane Koitz-Hristov, Lukas Stracke, and Franz Wotawa. Checked coverage for test suite reduction - is it worth the effort? In *IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2022, Pittsburgh, PA, USA, May 21-22, 2022*, pages 6–16. ACM/IEEE, 2022.

[30] Bogdan Korel and Janusz W. Laski. Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195, 1990.

[31] Kenneth Koster and David C. Kao. State coverage: a structural test adequacy criterion for behavior checking. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 541–544. ACM, 2007.

[32] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[33] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[34] Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, 1998.

[35] Facundo Molina, César Cornejo, Renzo Degiovanni, Germán Regis, Pablo F. Castro, Nazareno Aguirre, and Marcelo F. Frias. An evolutionary approach to translating operational specifications into declarative specifications. *Sci. Comput. Program.*, 181:47–63, 2019.

[36] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. Fuzzing class specifications. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1008–1020. ACM, 2022.

[37] Facundo Molina, Alessandra Gorla, and Marcelo d'Amorim. Test oracle automation in the era of llms. *ACM Trans. Softw. Eng. Methodol.*, 34(5):150:1–150:24, 2025.

[38] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. Evospex: An evolutionary algorithm for learning postconditions. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1223–1235. IEEE, 2021.

[39] Agustín Nolasco, Facundo Molina, Renzo Degiovanni, Alessandra Gorla, Diego Garbervetsky, Mike Papadakis, Sebastián Uchitel, Nazareno Aguirre, and Marcelo F. Frias. Abstraction-aware inference of metamorphic relations. *Proc. ACM Softw. Eng.*, 1(FSE):450–472, 2024.

[40] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 75–84. IEEE Computer Society, 2007.

[41] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: An analysis and survey. *Adv. Comput.*, 112:275–378, 2019.

[42] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and René Just. Does mutation testing improve testing practices? In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 910–921. IEEE, 2021.

[43] David Schuler and Andreas Zeller. 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*, pages 90–99. IEEE Computer Society, 2011.

[44] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. Evolutionary improvement of assertion oracles. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1178–1189, New York, NY, USA, 2020. Association for Computing Machinery.

[45] Dries Vanoverberghe, Jonathan de Halleux, Nikolai Tillmann, and Frank Piessens. State coverage: Software validation metrics beyond code coverage. In Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán, editors, *SOFSEM 2012: Theory and Practice of Computer Science - 38th Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 21-27, 2012. Proceedings*, volume 7147 of *Lecture Notes in Computer Science*, pages 542–553. Springer, 2012.

[46] Congying Xu, Valerio Terragni, Hengcheng Zhu, Jiarong Wu, and Shing-Chi Cheung. Mr-scout: Automated synthesis of metamorphic relations from existing test cases. *ACM Trans. Softw. Eng. Methodol.*, 33(6):150, 2024.

[47] Yucheng Zhang and Ali Mesbah. Assertions are strongly correlated with test suite effectiveness. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 214–224. ACM, 2015.