ZoomIn: Discovering Failures by Detecting Wrong Assertions

Fabrizio Pastore* and Leonardo Mariani†
*University of Luxembourg, Centre For Security, Reliability and Trust, Luxembourg, LUXEMBOURG
†University of Milano - Bicocca, Department of Informatics, Systems and Communications, Milano, ITALY
Email: {fabrizio.pastore@uni.lu, mariani@disco.unimib.it}

***Abstract*—Automatic testing, although useful, is still quite ineffective against faults that do not cause crashes or uncaught exceptions. In the majority of the cases automatic tests do not include oracles, and only in some cases they incorporate assertions that encode the observed behavior instead of the intended behavior, that is if the application under test produces a wrong result, the synthesized assertions will encode wrong expectations that match the actual behavior of the application.**

In this paper we present *ZoomIn*, a technique that extends the fault-revealing capability of test case generation techniques from crash-only faults to faults that require non-trivial oracles to be detected. ZoomIn exploits the *knowledge encoded in the manual tests* written by developers and the *similarity between executions* to automatically determine an extremely small set of suspicious assertions that are likely wrong and thus worth manual inspection.

Early empirical results show that ZoomIn has been able to detect 50% of the analyzed non-crashing faults in the Apache Commons Math library requiring the inspection of less than 1.5% of the assertions automatically generated by EvoSuite.

I. Introduction

To reduce manual verification effort, test case generation techniques can be used to automatically generate unit [1], [2], [3], integration [4] and system test cases [5], [6], [7] that extensively sample the behavior of the application under test. Developers can thus focus their manual effort on the cases that have not been well addressed by the automatic tests. Unfortunately, automatically generated test cases can easily discover problems like *crashes* and *uncaught exceptions* but cannot reveal problems that require *explicit oracles* to be detected [8]. For instance, an automatic test might reveal that a sum operation implemented by a calculator crashes when two negative numbers are added, but cannot reveal that a sum operation erroneously behaves as a multiply operation.

Modern test case generators have the capability to synthesize tests that include assertions encoding the *behavior observed during test case execution* [2], [3]. For instance, EvoSuite can generate a test case that invokes an `add` method with values 2 and 4 and incorporates the assertion `assertTrue(6==add(4,2))` in the test. These assert statements are useful for regression testing, to detect if a change unintentionally modifies the behavior of the system, but are almost useless to reveal bugs on the first place. For instance, if the `add` method erroneously behaves as a multiply operation, the assert statement automatically included in the test would look like `assertTrue(8==add(4,2))`. This happens because test case generators produce assert statements that capture the *actual behavior of the tests* - "the execution of `add(4,2)` returned 8" - rather than capturing the *expected result* - "the execution of `add(4,2)` should return 6".

The difficulty to automatically generate correct assert statements is an instance of a well-known problem: the *oracle problem* [8]. Solving the oracle problem, even partially, would dramatically increase the effectiveness of test case generators, resulting in a major improvement in software testing: *the automatic tests would scale from tests that can only detect faults that cause crashes and uncaught exceptions to tests that can potentially detect any functional fault*.

In principle it is possible to turn the assertions encoding the behavior observed during test execution into assertions encoding the expected behavior by manually inspecting and fixing every assert statement generated by test case generators. However, this approach does not scale to non-trivial programs. For instance, in our experiments EvoSuite generated 1887 tests and 4002 assertions for 77 classes in the Apache Commons Math library [9]. So many tests and assertions cannot be manually inspected in a reasonable amount of time.

As an alternative to manually fixing assertions, researchers investigated the use of the crowd [10]. However, to be successful, crowdsourcing requires a qualified crowd, which is not easy to find, and a monetary investment that might be significant when many tests and assertions need to be inspected.

The automatic tests that expose faulty behaviors without failing due to the lack of proper assert statements, such as the example of the test with the assertion `assertTrue(8==add(4,2))`, might be also detected using anomaly detection solutions [11], [12], [13]. Anomaly detection can be used to automatically identify the anomalous executions that occur in a set of executions. Under the assumption that failures are sparse in the execution space, anomaly detection techniques can be potentially used as an oracle to distinguish failures from successful executions.

So far anomaly detection approaches have not been precise enough to be used as test oracles, as demonstrated in a recent study by Nguyen, Marchetto, and Tonella [11], where anomaly detection techniques produced poor results when used to address the oracle problem. In particular, anomaly detection generates far too many false positives to be useful.

In this paper we show that anomaly detection *can* be effectively used to address the oracle problem if equipped with proper mechanisms that exploit both the *tester's knowledge* of the application under test and the *degree of similarity* between anomalous and regular executions. Our empirical results show that anomaly detection without these additional mechanisms is ineffective, confirming the results obtained in the study by Nguyen, Marchetto, and Tonella [11], while it can reveal several faults with little inspection effort when augmented as described in this paper.

The approach that we have defined, called *ZoomIn*, is a technique that can be integrated with any unit test case generator that synthesizes assertions as part of the automatic tests. ZoomIn can heuristically detect wrong assertions, and consequentially *expose faults that would otherwise go undetected*. In contrast with state of the art techniques, the objective of ZoomIn is not to detect all the wrong assertions (or to recognize all the failing tests), which is hardly doable without generating many false positives, but to pinpoint an extremely small number of likely wrong assertions that require the attention of the developers.

ZoomIn is based on two key ideas: (1) to exploit the *knowledge encoded in the test cases implemented by the developers* to detect the erroneous behaviors in the automatic tests, and (2) to *combine anomaly detection with code coverage* to make the detection of erroneous behaviors precise.

On a technical perspective, ZoomIn pinpoints the wrong assertions by comparing the executions produced by the manual test cases to the executions produced by the automatically generated test cases. ZoomIn originally performs this comparison by working at *two abstraction levels simultaneously*. The first level is code coverage, that is ZoomIn compares the statements covered by manual and automatic tests. The second level is program variables, that is ZoomIn uses Daikon [14] to generate constraints about the values that can be legally assigned to program variables when the manual tests are executed. ZoomIn detects anomalous variable values by checking the executions produced by the automatic tests using the constraints generated by Daikon from the manual tests. These two levels are combined according to the following intuition: *the execution of an automatic test case is likely to constitute a failure if it produces anomalous variable values while covering a case already tested by the developers*. In practice, we assume that an automatic test case that follows a path similar to one covered by a manual test case while generating anomalous variable values is an automatic test case that reveals a failure by *covering a special untested case of an already tested functionality*. On the contrary, an automatic test that produces unexpected variable values while executing a case not well tested by the manual test cases is not necessarily a suspicious test case. For instance, it could be a test case that covers untested features.

To properly assist developers, ZoomIn does not simply pinpoint the test cases that are likely to fail, but it directly points at the suspicious assertions that are likely to encode a wrong condition (i.e., the assertions that encode the actual behavior of the application rather than the expected behavior). Fixing these assertions would cause the tests to fail, thus revealing bugs in the program. The early empirical results show that ZoomIn applied to test cases generated by EvoSuite [2] can discover about 50% of the faults covered by the automatic tests by recommending the inspection of less than 1.5% of the generated assertions.

The major contributions of this paper are:

- the definition of ZoomIn, an anomaly detection technique that demonstrates how anomaly detection augmented with proper analysis mechanisms is suitable to address the oracle problem, complementing and extending the previous findings about the lack of effectiveness of anomaly detection applied to the oracle problem [11].
- the definition of a mechanism to extract information useful to guide anomaly detection from the manual tests designed by the developers.
- the use of coverage information to identify the executions that can be soundly compared.
- the definition of a strategy to pinpoint suspicious assertions.
- the early empirical results that show that the novel mechanisms introduced in this paper increase the fault detection ability of the EvoSuite test case generator [2] producing a tolerable set of false positives.

The paper is organized as follows. Section II overviews the ZoomIn approach. Section III presents how ZoomIn extracts information from manual test cases. Section IV presents how ZoomIn detects anomalous behaviors in automatic test cases, based on the information extracted from the manual tests. Section V describes how ZoomIn filters out the irrelevant anomalies from the set of discovered anomalies. Section VI presents the ranking of the assertions in the automatic tests. Section VII presents the empirical results. Section VIII discusses related work. Section IX provides final remarks.

II. ZoomIn

ZoomIn takes as input the automatic tests generated for a unit under test and returns a set of assertions that must be inspected manually. The analysis process is automatic and consists of four sequential steps as illustrated in Figure 1.

The first step, *Extracting Knowledge From Test Cases*, extracts information about the behavior of the class under test when executed with the passing test cases designed by developers. ZoomIn records both the return value and the parameter values for every executed method, and runs Daikon [14] to derive method pre- and post-conditions, generically referred as program constraints in this paper. For instance, step 1 in Figure 1 shows the case of no precondition and one post-condition derived by ZoomIn for method `conjugate()` implemented by the class under test `Complex`, which is one of the classes studied in our empirical evaluation. The post-condition `ret.imaginary!=0` indicates that the field `imaginary` of the object returned by method `conjugate()` is expected to be always different than zero (method `conjugate()` returns an object of type `Complex`).

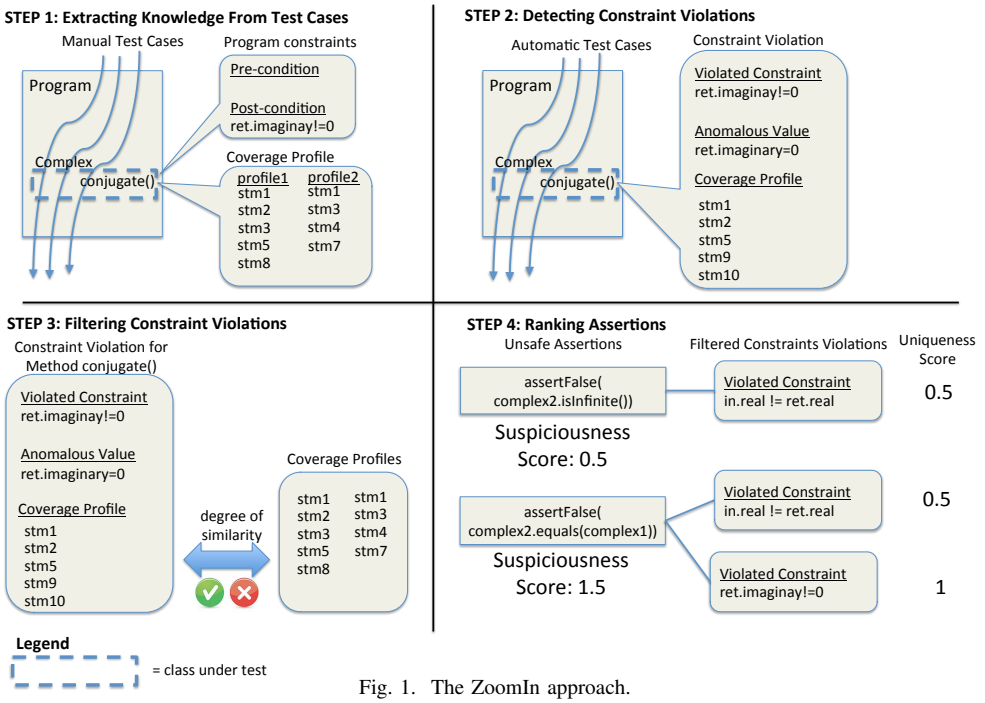

Fig. 1. The ZoomIn approach.

In addition to program constraints, ZoomIn collects the coverage profile associated with every method in the class under test. A coverage profile of a method `m` consists of the set of statements in the class under test executed by a test until reaching an exit point of method `m`. The total number of profiles associated with a method depends on the number of times the method is executed by the manual test cases. For instance, step 1 in Figure 1 shows the case of method `conjugate()` associated with two profiles, $profile1$ and $profile2$. In the example we use the generic label `stm` followed by a number to indicate a statement of the program. The profiles of a method represent the set of program behaviors that have been used to generate the program constraints (i.e., the pre- and post-conditions) associated with that method.

The second step, *Detecting Constraint Violations*, detects the automatic tests that produce anomalous variable values when executed. An anomalous value is a variable value that violates any of the constraints derived in step 1. For instance, step 2 in Figure 1 shows the case of the constraint `ret.imaginary!=0` violated by the method `conjugate()` when returning a number with the imaginary part equals to 0. ZoomIn uses the violated constraints to determine the most suspicious executions and assertions.

In general it is not enough that an automatic test violates a constraint to claim that the test failed, as done by classic anomaly detection. In fact, an automatic test might generate anomalous values simply because it executes behaviors that have been never tested by the manual test cases. ZoomIn addresses this problem exploiting the notion of *coverage profile*. Every time a constraint is violated, ZoomIn records its coverage profile. Since the program constraints can only be method pre- and post-conditions, the coverage profile of a violated constraint can be defined as the coverage profile of the method associated with the constraint. ZoomIn uses coverage profiles to assess if constraint violations have been observed in contexts that are comparable to the behaviors tested with the manual test suite. For instance, Step 2 in Figure 1 shows the case of a constraint violation associated with a profile composed of five statements.

The third step, *Filtering Constraint Violations*, has the objective of eliminating from the analysis all the constraint violations that have been detected with automatic executions that are too different from the set of manual executions used to generate the program constraints. The intuition is that a violated constraint is likely to detect a faulty behavior only if it is associated with a coverage profile that is similar to a profile already observed when executing the manual test cases. For instance, step 3 in Figure 1 shows the case of the constraint `ret.imaginary!=0` defined on the return value of the method `conjugate()` violated by the anomalous value `ret.imaginary=0`. The coverage profile of this constraint violation is compared with the profiles collected when executing the manual test cases. If the same or a close coverage profile is found, the violation is considered relevant, otherwise it is dropped.

The fourth step, *Ranking Assertions*, splits the assertions in the automatic test cases into two groups: the *safe* assertions and the *unsafe* assertions. The safe assertions are likely correct assertions that do not need to be inspected. The unsafe assertions are possibly wrong assertions that are worth inspection. The unsafe assertions are ranked, the higher the ranking of an assertion is the more likely the assertion is faulty.

ZoomIn determines the unsafe assertions and places them in a ranking based on the number of constraint violations that are related to them. The intuition is that *the assertions that use test variables that are related to program variables that produced*

one or more constraint violations are likely to be wrong. ZoomIn uses dynamic data-flow analysis to discover relations between program variables and variables in the assertions. The assertions that are not impacted by any anomalous value are considered safe. The rest of the assertions are considered unsafe and are included in the ranking.

To rank unsafe assertions, ZoomIn takes into consideration the number of constraint violations produced by the related program variables. However, not all the constraint violations are equally important. Since erroneous behaviors are not frequent in adequately tested software, ZoomIn weights each constraint according to the number of times the constraint has been violated. The frequently violated constraints are likely imprecise constraints that erroneously detect legal values as anomalous values, while constraints that are seldom violated are likely to carry useful information. ZoomIn captures this aspect with the *uniqueness score*. For instance, step 4 in Figure 1 shows the case of three constraint violations associated with different uniqueness scores.

ZoomIn associates the unsafe assertions with a score, the *suspiciousness score*, that represents the likelihood the assertion is wrong. The suspiciousness score of an assertion depends on both the number and the uniqueness scores of the related constraints that are violated. Intuitively the highly scored assertions, i.e., the most suspicious assertions, are the ones associated with several constraint violations with high uniqueness scores.

To keep the inspection effort low and the effectiveness high, developers are assumed to inspect only the top assertions in the ranking. Results show that inspecting the top *five* unsafe assertions is enough to discover several faults without wasting time inspecting too many correct assertions.

In the following sections we present each step of the approach in detail.

III. Extracting Knowledge From Test Cases

In the first step, ZoomIn extracts information on the behavior of the class under test by executing the manual test cases. We assume that testers have already written the manual test cases when the automatic tests are generated. We also assume that either all the manual test cases pass, that is the faults revealed by manual test cases have been fixed, or the manual test cases that fail have been excluded from the analysis.

Since analyzing all the variables in the program might be extremely expensive, ZoomIn limits the analysis to the interfaces, that is it observes the values assigned to any parameter and return value produced by any method and function in the class under test. The values observed while monitoring the manual test cases are recorded in trace files.

The recorded values are used to synthesize program constraints encoding method pre- and post-conditions. To obtain these constraints ZoomIn runs Daikon [14], which is an inference engine that can synthesize Boolean expressions over a set of variables from a set of observations. The generated expressions are statistically significant generalizations of the observations reported in the trace files, that is all the observations satisfy the expressions returned by Daikon and there is a small probability that the expressions hold incidentally. For instance, in our experiments Daikon generated the program constraint `ret.imaginary!=0` for the method `conjugate()` implemented in the class `Complex` from the data recorded during the execution of the test cases available for the Apache Commons Math library.

The program constraints derived in this phase represent the knowledge on the behavior of the class under test that has been automatically extracted from the execution of the manual test cases. Since the constraints are derived from passing executions, they essentially represent the values that can be legally assigned to program variables. Of course it is not possible to completely prevent the generation of constraints that overfit the cases covered with the manual tests. For instance, in our experiments, ZoomIn derived the over-fitting constraint `in.real != ret.real` for method `divide(Complex in)` implemented by class `Complex`: it is clearly false that the real part of the divisor (parameter `in`) and the real part of the result (return value `ret`) cannot be the same.

ZoomIn handles over-fitting constraints by keeping track of the executions used to derive them and employing the constraints only in comparable contexts. Intuitively an over-fitting constraint is an accurate constraint in the restricted context used to derive it. To enable this analysis, ZoomIn uses the notion of coverage profile: The coverage profile of a method `m` is defined as the set of statements that have been executed until reaching an exit point of `m`. ZoomIn records the coverage profile for all the executed methods.

As shown in step 1 of Figure 1, once the execution of the manual test suite is completed, each method of the class under test is associated with a set of program constraints (representing method pre- and post-conditions) and a set of coverage profiles: *these constraints are expected to hold for executions similar to the collected profiles*.

IV. Detecting Constraint Violations

In the second step, ZoomIn looks for evidence of failure in the automatic tests. To this end, it runs the automatic test cases and checks executions with the program constraints derived in step 1 from the manual test cases. Every variable value that violates a constraint is an anomalous value, which is interpreted as an evidence of failure.

For instance, if an automatic test case causes `conjugate()` to return an object of type `Complex` with the field `imaginary` set to 0, the value is considered anomalous because it violates the constraint `ret.imaginary!=0` derived in step 1. Similarly, if an automatic test case divides two imaginary numbers and the real part of the resulting number is the same as the divisor, the values are considered anomalous because they violate the constraint `in.real != ret.real` for method `Complex.divide(Complex in)` derived in step 1.

Anomalous values can be generated for two reasons. The first reason is that there is a fault in the program and when

the fault is executed the application assigns wrong values to program variables. The second reason is that the constraint is inaccurate, and new legal executions incidentally produce values that violate the constraint. Clearly, only the first class of violations are useful to ZoomIn.

ZoomIn filters out violations caused by inaccurate constraints by taking into account the characteristics of the executions that produced the violations. If the execution is too different from the executions that have been used to generate the constraints, the anomalous value is likely to be a new legal value produced by an untested scenario rather than a symptom of program failure. Thus, the anomalous value should be dropped. To enable this filtering strategy, ZoomIn collects the coverage profiles associated with constraint violations.

In particular, given an automatic test `t` that executes a method `m` with parameter and return values that violate a constraint `c` on `m` (i.e., `c` is either a `m`'s pre- or post-condition), the coverage profile of the constraint violation is defined as the set of statements executed by `t` until exiting method `m`. Note that the definition of coverage profile for constraint violation matches the definition of method profile given in Section III.

At the end of this step, for every constraint violation detected during the execution of the automatic tests, ZoomIn identified the violated constraint, the anomalous values that violate the constraint, and the coverage profile of the constraint, as illustrated in the step 2 of Figure 1.

V. Filtering Constraint Violations

In the third step ZoomIn filters out the constraint violations that should not be interpreted as evidence of failure. The idea is that only the constraint violations detected while running the automatic tests that cover cases similar to the cases already covered by the manual tests are likely to indicate a problem in the program.

In the practice, for each constraint violation detected during the execution of the automated tests (see step 2) ZoomIn compares the coverage profile of the violation with the coverage profiles recorded while running the manual tests (see step 1). More formally, given a constraint violation cv detected by a program constraint c associated with method `m` (i.e., c is either a `m`'s pre-condition or `m`'s post-condition), ZoomIn compares the coverage profile associated with cv to the collected coverage profiles. Note that the coverage profile of cv includes statements in the body of `m` even if c is a pre-condition. This is useful to discard the constraint violations generated by pre-conditions violated with legal invocations that activate behaviors of `m` that have been never executed before. The comparison between two profiles returns a value in the range $[0, 1]$ that defines the similarity between the profiles. If none of the profiles is similar enough to the profile associated with the anomalous value, the anomalous value is dropped.

ZoomIn uses the Jaccard similarity index [15] to compute the degree of similarity (DoS) between two profiles. Formally, given two coverage profiles $prof1$ and $prof2$, the DoS is:

$$DoS(prof1, prof2) = \frac{stms\ \ in\ \ prof1 \cap stms\ \ in\ \ prof2}{stms\ \ in\ \ prof1 \cup stms\ \ in\ \ prof2}$$

where *stms* indicates of the set of statements in a profile.

The Jaccard index is a popular formula used to measure similarity between two finite sets. In this case it computes the number of statements that are present in both profiles divided by the total number of statements in the profiles.

ZoomIn compares profiles to take decisions about constraint violations. More formally, given a profile $profile_{cv}$ associated with a constraint violation cv and given a set of coverage profiles $profiles$, the constraint violation cv is discarded if the following condition holds:

$$\nexists profile \in profiles \text{ s.t. } DoS(profile_{cv}, profile) > 0.75$$

that is the constraint violations detected with a coverage profile that shares less than 75% of the statements with the most similar coverage profile collected by executing the manual tests are dropped. The constraint violations that are not dropped are significant and used in step 4 of the technique.

We set the threshold for filtering to 0.75 based on our early empirical results. We did not notice big changes on the results for small changes to the threshold. A more extensive study aimed at empirically defining an optimal value for this parameter is part of our future work.

VI. Ranking Assertions

In the fourth step ZoomIn splits the assertions into two disjoint sets: the *safe* assertions and the *unsafe* assertions. The unsafe assertions are ranked according to their *suspiciousness score*, which is a positive number that represents the likelihood the assertion is wrong.

Since an automatically generated assertion encodes the behavior observed when running an automatic test, the likelihood the assertion is wrong depends on the correctness of the execution produced by the test. The rationale implemented in ZoomIn is that the more constraint violations are generated by an automatic test the more likely the assertions in the test capture a wrong behavior.

Since ZoomIn runs the analysis at the granularity of the individual assertions, not all the constraint violations generated by a test are relevant to every assertion. ZoomIn discovers what constraint violations impact on what assertions using dynamic data-flow analysis. A constraint violation is relevant to a given assertion only if the anomalous values that violated the constraint contribute to the definition of the assertion. ZoomIn discovers the anomalous values that contribute to the definition of the assertions in the automatic tests looking at dynamic data-flow chains [16].

More formally, we say that a variable v is defined by a statement e if e sets the value of v. A variable v is used by a statement e if e accesses the value of v. ZoomIn detects a dynamic def-use pair when the execution of a test case produces the definition of a variable v and successively the use of the same variable v, without redefining the value of v before the use. ZoomIn detects a dynamic def-use chain between variable v and variable w when the execution of a test produces a sequence of def-use pairs $\langle d_i, u_i \rangle\ i = 1 \ldots n$ with d_1 definition of variable v, $(u_i, d_{i+1})\ \forall i = 1, \ldots n-1$ pairs of

uses and definitions produced by the same statement, and u_n use of variable w. Intuitively variables v and w are connected by a dynamic def-use chain if v contributed to determine the value of w in a concrete execution.

An anomalous value related to variable v is said to impact an assertion a defined in the automatic test t if the execution of t produces a def-use chain that connects v to any of the variables used in the assertion a. A constraint violation cv *affects an assertion* a only if the anomalous values that generated cv impact the assertion a. According to this definition each assertion in an automatic test can be associated with a possibly empty set of constraint violations that affect it. For instance, step 4 in Figure 1 shows the case of two assertions, one affected by one constraint violation and another affected by two constraint violations.

The assertions affected by no constraint violation are considered *safe* and dropped from the analysis. The rest of the assertions are considered *unsafe*. ZoomIn ranks the unsafe assertions according to the number and severity of the constraint violations that affect them.

Since erroneous behaviors are not frequent in well tested software, ZoomIn determines the severity of each constraint violation according to its *observational frequency*. In particular, the more frequently a constraint is violated the less the violations produced by that constraint are informative. On the contrary, tricky behaviors and erroneous corner cases are not expected to occur frequently. To capture this intuition, ZoomIn weights each constraint violation with a value in the range $[0, 1]$ that we called *uniqueness score*.

Given an automatic test suite *ATC*, and a constraint violation *cv* produced by a constraint c, the uniqueness score (*US*) of *cv* is defined as follows:

$US(cv) = \frac{1}{viol(c,ATC)}$

where $viol(c, ATC)$ is the number of times the constraint c is violated by the automatic tests in *ATC*.

The uniqueness score is used to properly weight the constraint violations when computing the suspiciousness of an assertion. In particular, given an assertion a and a set constraint violations cv_i with $i = 1 \ldots n$ that affect a, we define the suspiciousness (*SOSP*) of a as

$SOSP(a) = \sum_{i=1}^{n} US(cv_i)$

Note that the value of the suspiciousness is not bounded and can arbitrarily increase depending on the number of constraint violations that affect the assertion and their uniqueness scores.

The suspiciousness value of the assertions in the automatic tests is used to generate a ranking. ZoomIn does not aim to discover all the wrong assertions in the automatic tests, but it is designed to focus the attention of the developers on the highly suspicious assertions that are likely to reveal new faults in the program when inspected. Although developers could inspect an arbitrary number of assertions following the ranking (e.g., all the unsafe assertions), according to our experiments we recommend using ZoomIn to inspect *no more than the top five assertions*. In this way the inspection effort is low, and the effectiveness of the approach high.

For example, in the Apache Commons Math library ZoomIn ranked the following assertion at the third position:

```
assertFalse(complex2.equals(complex1))
```

with `complex1` equals to -1 and `complex2` assigned with the conjugate of `complex1` which is by definition still -1. This wrong assertion exposes a fault in the program. The fault has been covered by the automatic tests but would go unnoticed if this assertion is not inspected and fixed. Note that in our experiments ZoomIn automatically selected this assertion among 331 assertions generated by EvoSuite.

VII. Empirical Evaluation

In this section we briefly illustrate the tool that we used for the empirical evaluation, we present the subjects used for the evaluation, and then we present a study that investigates the effectiveness of ZoomIn using more than 1,400 assertions and six real life non-crashing faults. The study considers multiple inspection strategies and includes an empirical comparison between ZoomIn and classic anomaly detection. Early results demonstrate that ZoomIn can restrict the set of assertions that must be inspected to less than 1.5% of the overall set of generated assertions and reveal 50% of the faults, while regular anomaly detection ineffectively selected more than 30% of the assertions for inspection. We also discuss ZoomIn qualitatively presenting one of the studied faults in detail. We finally discuss threats to validity.

Prototype Implementation: Our ZoomIn prototype is implemented in Java and integrates different third-party libraries and tools. Program variables are monitored using the TPTP Probekit [17] while execution profiles are collected using the JaCoCo library [18]. Daikon is the inference engine used to derive program constraints from traces [14]. Runtime checking of constraints generated with Daikon is again implemented with TPTP. Dynamic data-flow analysis is implemented using the Eclipse Java Development Tools (JDT) [19]. Finally we used the EvoSuite test case generator [2], which is a test case generation tool that can synthesize test cases with assertions.

TABLE I
Real Faults Investigated with ZoomIn

Class name	Bug ID	Bug Descriptor
`Complex`	MATH-221	https://issues.apache.org/jira/browse/MATH-221
`MathUtils`	MATH-241	https://issues.apache.org/jira/browse/MATH-241
`Max`	MATH-57	https://issues.apache.org/jira/browse/MATH-57
`Min`	MATH-57	https://issues.apache.org/jira/browse/MATH-57
`OpenMap-RealVector`	MATH-326	https://issues.apache.org/jira/browse/MATH-326
`Random-DataImpl`	MATH-294	https://issues.apache.org/jira/browse/MATH-294

Subjects: For the purpose of the evaluation we selected the Apache Commons Math library [9], which is a popular open source library available with a suite of test cases and with several publicly documented bugs. To identify the specific

TABLE II
RESULTS OBTAINED WITH REAL FAULTS

Class	Manual Tests		Constraints	Constraint Violations		Tests	Assertions		Position
	Total/Class Tests	Total/Class Cov		All	Filtered		All	Unsafe	
Complex	819 / 116	92% / 93%	4726	260	85 (33%)	83	331	34 (10%)	**3 (1%)**
MathUtils	819 / 95	92% / 87%	171	61	8 (13%)	60	67	8 (13%)	-
Max	654 /42	93% / 100%	28	9	8 (88%)	26	69	8 (6%)	**5 (7%)**
Min	654 / 46	93% / 100%	26	33	2 (12%)	22	62	2 (6%)	**-**
OpenMapRealVector	2050 / 7	90% / 88%	5277	17856	44 (1%)	74	170	14 (6%)	**3 (2%)**
RandomDataImpl	2050 / 37	90% / 92%	1205	250	0 (0%)	52	722	0 (0%)	**-**

faults and classes for our study, we selected all the bugs in the Apache Commons Math bug repository that fulfill the following three criteria: (1) belong to either version 1.x or 2.x, (2) are not revealed by the Commons Math test suite, and (3) fail without generating crashes or uncaught exceptions but simply generating wrong outputs. We then generated the unit tests for the classes with the faults using EvoSuite. Evosuite generated 317 tests and 1421 assertions and has been able to cover 6 non-crashing faults. We say that a test covers a fault if it executes the lines that have been modified in the official fix of the fault and at least one of its synthesized assertions fails when executed on the fixed version of the program. Table I shows for each fault the class containing the fault, the bug id and the url pointing at the corresponding bug report. Note that none of these faults can be automatically revealed by state of the art test case generators. In our evaluation, ZoomIn addresses these faults by working on a set of 1,421 assertions.

Evaluation with Real Faults: We applied ZoomIn to the subjects cases and manually inspected the ranked assertions to evaluate the technique.

Table II shows the results. Column *Class* indicates the class with the fault. Column *Manual Tests* provides information about the manual test suite. Column *Total/Class Tests* reports the size of the entire test suite and the number of test cases that cover the class with the fault. Column *Total/Class Cov* reports the statement coverage achieved by the manual test suite on the whole library and on the class with the fault only. Column *Constraints* indicates the number of program constraints (i.e., method pre- and post-conditions) generated by ZoomIn when executing the test cases distributed with Apache Commons Math. Column *Constraint Violations All* indicates the number of constraint violations detected by ZoomIn when executing the tests generated with EvoSuite. Column *Constraint Violations Filtered* indicates the number and percentage of constraint violations used by ZoomIn to identify and rank the unsafe assertions, the rest of the violations have been automatically filtered out by ZoomIn using execution profiles. The column *Tests* indicates the number of tests generated by EvoSuite. The Column *Assertions All* indicates the number of assertions generated by EvoSuite. The Column *Assertions Unsafe* indicates the number and percentage of assertions that ZoomIn classified as unsafe. Finally, column *Position* indicates the position in the ranking of the first assertion that points at the fault in the program. In case the assertion has the same ranking as other assertions, the position is obtained by referring to the average case. For instance if both the wrong assertion and a correct assertion are at position 4 of the ranking, the reported value would be 4.5. The value between parentheses indicates the percentage of the generated assertions that need to be inspected before reaching the wrong assertion. The symbol – indicates that no assertion pointing at the fault is part of the ranking.

ZoomIn generated a variable number of constraints depending on the complexity of the case. For instance, it generated more than four thousands constraints for `Complex`, while it generated 26 constraints for `Min`. The reason for this largely different results is the complexity of the parameters and return values that occur in the cases. When methods exchange several complex objects that recursively include attributes that are objects, ZoomIn recursively records the values of all these attributes and exploits them to generate many program constraints. On the contrary, when the parameters and return values mainly consist of variables with a primitive type, the number of constraints that are generated decreases drastically.

When running the automatic tests, ZoomIn checks the executions using the constraints generated from the manual tests. Again a variable number of constraint violations have been detected. Interestingly, when a non trivial number of violations have been detected, ZoomIn always filtered out many of the violations. For instance, ZoomIn filtered out 67% of the constraint violations for `Complex`, and more than 80% of the constraint violations for `MathUtils`, `Min`, `OpenMapRealVector`, and `RandomDataImpl`. ZoomIn kept the majority of the constraint violations only for `Max`, where a small number of constraint violations were already detected on the first place. *These results provide evidence of the capability of the filtering step to rule out the violations that result from incomparable executions and focus the analysis on the most relevant constraint violations*.

For all the cases, EvoSuite generated a significant number of tests and assertions. In particular it always generated more than 60 assertions, generating up to 722 assertions in one case. To discover the number of wrong assertions in the tests we executed the EvoSuite test cases on the correct version of the classes (the investigated bugs have an official fix) and we discovered that for all the six cases there is *only one test with a single assertion that reveals the fault*. Finding the wrong assertions within such a large set of assertions is extremely demanding for developers (there are only 6 assertions that can reveal the six faults among the 1,421 generated assertions).

In this challenging setting, ZoomIn demonstrated its effectiveness. Overall, ZoomIn isolated 66 unsafe assertions

from a total of 1,421 assertions. In the individual cases, the percentage of unsafe assertions ranged from 0% to 13% of the total number of assertions. This strong reduction has been obtained losing only half of the wrong assertions, that is *ZoomIn effectively selected less than 5% of the assertions preserving 50% of the wrong assertions*.

In some cases the developers might decide to inspect all the unsafe assertions, such as for `Max` and `Min` where the set of unsafe assertions is small (8 unsafe assertions for `Max` and 2 for `Min`). In some other cases, such as for `Complex` and `OpenMapRealVector`, the number of unsafe assertions might be too large. It is thus important to direct the developers' effort toward the most suspicious assertions. ZoomIn efficiently implements this capability by ranking unsafe assertions.

In fact, in the three cases with a faulty assertion, ZoomIn ranked the faulty assertion fifth in the worst case. This means that developers could discover new faults that would otherwise go undetected by just inspecting at most five assertions per case. If we consider the total number of assertions we started from, *the reduction of the search space has been dramatic, ranging from 1% to 7% of the overall set of assertions generated by EvoSuite*.

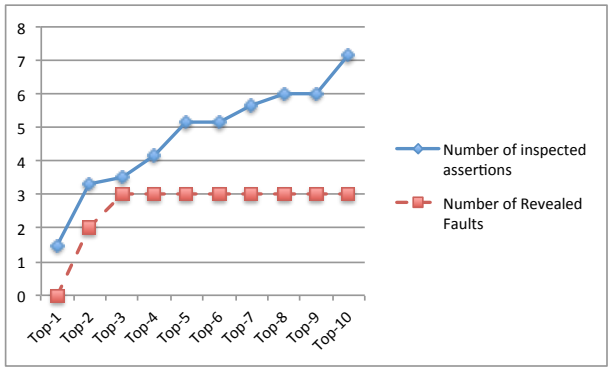

Fig. 2. Effectiveness of Inspection Strategies.

Stopping Criterion: In the practice developers do not know if there is any fault that must be discovered by inspecting assertions, thus they need a criterion to decide when stopping the inspection of the assertions. The main purpose of ZoomIn is to keep the inspection activity practical, at the cost of missing some faulty assertions. Thus the recommendation is to inspect an extremely small number of assertions located at the top of the ranking and then stop inspecting.

According to the early evidence collected with the study presented in this paper, inspecting the top five assertions is enough to detect most of the faults. We investigated more in details the inspection strategy by analyzing the tradeoff between the inspection effort and the number of revealed faults. Figure 2 plots the number of detected faults and the average number of inspected assertions for different inspection strategies. In particular, we consider the *Top-<n> inspection strategy* that requires inspecting the n assertions at the top of the ranking. If after having inspected n assertions there are other assertions ranked at the same position than the last assertion that has been inspected, the criterion requires developers to also inspect the equally ranked assertions. Note that the average values represented in the plot consider all the six cases, including the cases where the set of unsafe assertions includes no wrong assertion.

In the plot we only consider strategies from Top-1 to Top-10, assuming that inspecting more than 10 assertions is impractical. Results show that the Top-3 strategy is the best compromise between the capability to discover faults and the number of assertions that need to be inspected (3.5 on average). Note that in the case of the Apache Commons Math this strategy corresponds to inspecting only 21 assertions from a total of 1,421 assertions. However, these results cannot be yet generalized, thus inspecting a higher number of assertions, for instance by applying the Top-5 or Top-6 strategy, could be a safer option. Note that in our evaluation the Top-10 strategy requires inspecting less than 10 assertions because the total set of unsafe assertions is often smaller than 10.

TABLE III
COMPARISON BETWEEN ZOOMIN AND ANOMALY DETECTION

Approach	Bugs	Inspected assertions
ZoomIn (Top-<3>)	3/6	21/1,421 (1.5%)
ZoomIn (Top-<5>)	3/6	30/1,421 (2.1%)
Anomaly Detection	6/6	454/1,421 (31.9%)

Comparison to Anomaly Detection: We compared the effectiveness of ZoomIn to classic anomaly detection. As already done in other studies [11], we use anomaly detection to learn the legal behavior of the application (i.e., we learn the method pre- and post-conditions using Daikon) when the manual tests are executed, and then we check the executions of the automatic tests against the inferred constraints. The tests, and the assertions included in these tests, that violate the inferred constraints are reported to the user. In practice, anomaly detection implements the learning and the checking ability of ZoomIn, but does not include any of the other capabilities that characterize ZoomIn, such as the ability to collect coverage profiles, the ability of identifying soundly comparable executions based on the profiles, and the ability to correlate constraint violations to assertions. Results are reported in Table III. For ZoomIn we report the results obtained applying both the Top-<3> strategy, which is the most effective strategy for Apache Commons Math, and the Top-<5> strategy, which represents a more conservative choice for the inspection strategy.

We can notice that although anomaly detection can detect all the faults, it requires the inspection of a so large number of assertions that the result would not be practically useful in any context (454 assertions must be inspected). These results confirm the empirical observations reported in other papers, such as [11]. On the contrary, ZoomIn detected half of the faults with a manageable inspection effort: 21 or 30 assertions must be inspected, corresponding to 1.5% or 2.1% of the generated assertions, depending on the strategy that is applied.

Qualitative Analysis: To qualitatively demonstrate the ability of ZoomIn to identify the wrong assertions based on

anomalous values, we discuss more in details one of the cases considered in our empirical evaluation. Listing 1 shows the code of one of the tests generated by EvoSuite for the `Complex` class. The test covers a bug in the `Complex` class and includes a wrong assertion. We indicated the wrong assertion with the comment "`//WRONG`".

In our evaluation ZoomIn has been able to rank this assertion as a highly suspicious assertion (it occurs at position 3 of the ranking) among more than 300 generated assertions.

Listing 1. Sample test generated by EvoSuite

```
@Test
public void test35()  throws Throwable  {
 Complex complex0 = Complex.I;
 Complex complex1 = complex0.multiply(complex0);
 Complex complex2 = complex1.conjugate();
 assertEquals(1.0, complex2.abs(), 0.01D);
 assertEquals((-1.0), complex2.getReal(), 0.01D);
 assertFalse(complex2.equals(complex1)); //WRONG
 assertEquals((-1.0), complex1.getReal(), 0.01D);
}
```

The fault that causes the generation of a wrong assertion is in the `equals` method of the `Complex` class, which returns a wrong result when an imaginary number with +0 as imaginary part is compared to an imaginary number with the same real part but with -0 as imaginary part. The test generated by Evosuite interestingly covered this situation through a non-trivial sequence of method calls. Variable `complex0` is initialized to the imaginary number i. The variable `complex1` is then assigned to -1 (with -1 as real part and +0 as imaginary part) as a consequence of the call to `multiply`. The next invocation to method `conjugate()` assigns to `complex2` the conjugate of -1 which is still -1 (with -1 as real part and -0 as imaginary part). Although the sign of the imaginary part should be ignored when comparing the two numbers, the implementation of `equals` compares the bit level representations of these two numbers, thus incorrectly taking into account the sign even if the compared value is 0.

ZoomIn ranks this assertion as a highly suspicious assertion because it is impacted by violated constraints with high uniqueness scores. The two impacting constraints with the highest scores are `returnValue.imaginary != 0` and `returnValue.I.real != returnValue.imaginary`, both of them are associated with method `conjugate()` of class `Complex`. Since `conjugate()` is the method that produces the number with -0 as imaginary part, the constraint violations exactly characterize the execution that produces the failure among all the executions of the class `Complex`.

To determine the presence of a fault in the program the developer is only asked to inspect the test. There is no need to inspect the execution trace or to look at the constraint violations that determined the ranking of the assertion. For instance, in this case it is enough to follow the test to see that `complex1` and `complex2` are both -1 and thus the method `equals` must return true and not false as asserted in the test.

Threats to Validity: The most relevant threats to validity concern internal and external validity.

The main threat to the internal validity is related to the existence of a causal relation between the effectiveness of ZoomIn and its capabilities, that is the capability to filter out the irrelevant constraint violations, to filter out likely safe assertions, and to create the ranking. Since it was not feasible to manually inspect all the constraint violations and all the executions considered in the study, we addressed this threat by manually inspecting 10% of the violations that have been filtered out and inspecting all the constraint violations associated with the assertions that revealed faults in the program. The inspection of the discarded constraint violations confirmed that these violations are typically not related to any fault in the code but are related to over-fitting constraints. The inspection of the violations associated with the assertions that revealed the faults confirmed that the assertions have been properly ranked thanks to multiple well scored constraint violations that are clearly caused by the fault. The case discussed in the previous section is a practical example of the way ZoomIn works. We can thus claim that according to the early empirical evidence that we obtained, the effectiveness of ZoomIn is directly related to the effectiveness of its heuristics.

The main threats to external validity relate to the generalizability of the results to other applications and test suites. The good results obtained with different real faults mitigate this threat. However more studies with more subjects and test suites need to be completed to claim the generalizability of the results to other classes of applications.

In the empirical evaluation reported in this paper we used EvoSuite as test case generator. Although we do not see any strong reason why the effectiveness of the technique should not generalize to other test case generators that synthesize assertions, this aspect should be confirmed empirically.

VIII. Related Work

In this section we discuss approaches for automatically detecting failures, with special emphasis on techniques that do not require user-defined oracles.

Anomaly Detection: Anomaly detection techniques analyze sets of executions to identify the ones that are anomalous (i.e., that differ significantly) compared to the other executions in the set. The assumption of anomaly detection techniques is that the executions that exhibit anomalous behaviors are also the ones that are faulty.

Different techniques look at different aspects to determine what the anomalous behaviors are. For instance, Zheng et al. mine predicate rules that represent the conditions that hold at relevant program points, such as branches and exit points [20]. The executions that violate these rules are classified as potential failures. In a similar way Raz et al. mine constraints on the values returned by online data sources [21]. Anomalous values are discovered by comparing newly collected data to the mined constraints.

Other approaches identify anomalous executions by looking at the sequences of operations executed by an application rather than looking at the values assigned to variables. For instance, several techniques mine finite state models from

execution traces [22], [23] and use these models to assess the correctness of newly collected executions [24], [13], [25].

Anomaly detection techniques can easily generate many false alarms, especially when applied to the oracle problem, as demonstrated in a recent study [11]. In contrast with these approaches, ZoomIn can isolate an extremely small number of assertions that require inspection, thus always maintaining manual effort under control. Moreover assertions can be cheaply evaluated by inspecting the tests only, while inspecting an entire execution isolated by a classic anomaly detection technique might be far more expensive.

Spectra-Based Techniques: Spectra-based techniques compare executions according to a specific aspect of the program behavior. The ability to compare executions can be exploited to cluster test cases and identify failing runs. For instance, Yilmax et al. show that hybrid program spectra can be effectively used to train a *j48* classifier that distinguishes passing and failing executions [26]. Lo et al. investigate the use of support vector machines, trained with passing and failing executions, to classify unknown executions [27].

The main limitation of these approaches is that they require a set of passing and failing executions to train the classifiers. ZoomIn does not need to be trained with sample failing executions, that are difficult to be obtained for multiple applications and multiple types of failures, but it can be applied relying uniquely on the manual tests designed by the developers.

Dickinson et al. investigate the classification of unknown executions using unsupervised hierarchical clustering [28]. Since failures are assumed to end up in clusters that only contain failures, developers can inspect the output produced by this technique by inspecting one execution per cluster.

The technique by Dickinson et al. requires estimating the number of clusters that must be returned by the clustering algorithm. An optimal estimate is difficult to produce and suboptimal choices might significantly impact the quality of the results. If many clusters are generated, a significant inspection effort is required, while producing few clusters might lead to clusters that mix up failing and passing executions. ZoomIn does not require these kinds of estimates and it always requires the inspection of a few assertions only.

Specification-Based Oracles: Software specifications have been extensively exploited to generate test oracles. For instance, Carzaniga et al. recently investigated how to generate oracles from a specification of the equivalent sequences, which is a specification that indicates the sequences of method calls that have the same effect on an object state [29]. In some cases the specification can be part of the program. For instance ARTOO uses contracts to generate tests and detect faults [30].

Specification-based approaches are extremely effective, but require suitable specifications to be applied [8]. Unfortunately, specifications are often unavailable, and when they are available are often incomplete, outdated, and inconsistent. ZoomIn supports testers in the common case no specification suitable for the generation of program oracles is available.

Test Driven Oracle Generation: Several test case generation techniques can generate test cases that include assertions that encode the observed behaviors, such as Randoop [3] and Evosuite [2]. Techniques that produce a small but effective set of assertions have been also investigated [31], [32]. ZoomIn augments these techniques with the capability to identify the assertions that are likely wrong due to faults in the program.

Some approaches investigated the integration of test case generation and failure detection. For instance, Eclat [33] and DSDCrasher [34] can augment an existing test suite with test cases that are likely to expose failures by causing executions that violate methods' post-conditions without violating methods' pre-conditions. Pre- and post-conditions are generated with Daikon. Since this strategy can generate several false positives, DSDCrasher only reports the program crashes.

Compared to these approaches, ZoomIn is more precise, because it limits the number of false positives that are generated while maintaining the capability to reveal non-crashing faults, and more accurate, because it works at the granularity of assertions instead of entire executions.

Human-Centric Approaches: While specifying program oracles (e.g., writing assertions) is a relatively simple activity for testers, it is an extremely challenging task for computer programs. For instance, a recent study by Staats et al. demonstrated that developers may easily misclassify automatically generated invariants [35].

Pastore et al. early investigated how to involve humans in the synthesis of program oracles by using Crowdsourcing to fix the assertions automatically generated by EvoSuite [10]. The results show that this solution is feasible, although the performance of the crowd depends on the complexity of the domain, and the economical advantage of this solution depends on the number of tests and assertions that must be inspected.

ZoomIn does not require the inspection of many Daikon invariants and does not involve non-experts in the process, as potentially done by crowdsourcing. ZoomIn only requires the developers of the application to inspect a few specific assertions generated by test case generation tools.

IX. Conclusions

Test case generation techniques can automatically generate test cases that include assertions encoding the behavior observed when running the tests. Thus if a test reveals a fault that makes the program fail by producing a wrong output, the synthesized assertion, instead of detecting the failure, will encode a wrong condition. This fact significantly reduces the fault detection capability of test case generation techniques.

In this paper we presented ZoomIn, a technique that can augment test case generators with the capability to pinpoint the assertions that are likely incorrect due to the presence of faults in the program. The empirical results obtained by integrating ZoomIn with EvoSuite demonstrated the effectiveness of the approach. In particular the most effective inspection strategy identified 50% of the faults in the program by inspecting less than 1.5% of the assertions synthesized by EvoSuite.

Future work includes gaining experience with ZoomIn with other faults, applications, and test case generators to increase confidence on the generality of the results obtained so far.

ACKNOWLEDGMENT

This work has been partially supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03).

REFERENCES

[1] N. Tillmann and J. D. Halleux, "Pex: white box test generation for .NET," in *proceedings of the International Conference on Tests and Proofs*, 2008.

[2] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proceedings of the European Conference on Foundations of Software Engineering, tool demo*, 2011.

[3] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *Companion to the Conference on Object-oriented Programming Systems and Applications*, 2007.

[4] M. Pezzè, K. Rubinov, and J. Wuttke, "Generating effective integration test cases from unit ones," in *proceedings of the International Conference on Software Testing, Verification and Validation*, 2013.

[5] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 884–896, 2005.

[6] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "AutoBlackTest: Automatic black-box testing of interactive applications," in *proceedings of the International Conference on Software Testing, Verification and Validation*, 2012.

[7] F. Gross, G. Fraser, and A. Zeller, "Search-based system testing: High coverage, no false alarms," in *proceedings of the International Symposium on Software Testing and Analysis*, 2012.

[8] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "A comprehensive survey of trends in oracles for software testing," University of Sheffield, Tech. Rep. CS-13-01, 2013.

[9] Apache, "Commons math," http://commons.apache.org/proper/commons-math/.

[10] F. Pastore, L. Mariani, and G. Fraser, "Crowdoracles: Can the crowd solve the oracle problem?" in *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2013.

[11] C. D. Nguyen, A. Marchetto, and P. Tonella, "Automated oracles: An empirical study on cost and effectiveness," in *proceedings of the Joint Meeting on Foundations of Software Engineering*, 2013.

[12] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *proceedings of the European Software Engineering Conference and the Symposium on The Foundations of Software Engineering*, 2007.

[13] L. Mariani, F. Pastore, and M. Pezzé, "Dynamic analysis for diagnosing integration faults," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 486–508, 2011.

[14] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, 2001.

[15] P. Jaccard, "The distribution of the flora of the alpine zone," *New Phytologist*, vol. 11, pp. 37–50, 1912.

[16] J. Huang, "Detection of data flow anomaly through program instrumentation," *IEEE Transactions on Software Engineering*, vol. 5, no. 3, pp. 226–236, 1979.

[17] Eclipse Community, "TPTP probe kit," http://www.eclipse.org/tptp/platform/.

[18] ——, "JaCoCo library," http://www.eclemma.org/jacoco/.

[19] ——, "Java development tools (JDT)," http://www.eclipse.org/jdt/.

[20] W. Zheng, M. Lyu, and T. Xie, "Test selection for result inspection via mining predicate rules," in *proceeding of the International Conference on Software Engineering*, 2009.

[21] O. Raz, P. Koopman, and M. Shaw, "Semantic anomaly detection in online data sources," in *Proceedings of the International Conference on Software Engineering*, 2002.

[22] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *proceedings of the International Conference on Software Engineering*, 2008.

[23] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller, "Mining object behavior with ADABU," in *proceedings of the International Workshop on Dynamic Systems Analysis*, 2006.

[24] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy, "Unifying FSM-inference algorithms through declarative specification," in *Proceedings of the International Conference on Software Engineering*, 2013.

[25] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *proceedings of the International Symposium on Software Reliability Engineering*, 2008.

[26] C. Yilmaz and A. Porter, "Combining hardware and software instrumentation to classify program executions," in *proceedings of the International Symposium on Foundations of Software Engineering*, 2010.

[27] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun, "Classification of software behaviors for failure detection: A discriminative pattern mining approach," in *proceedings of the International Conference on Knowledge Discovery and Data Mining*, 2009.

[28] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles," in *proceedings of the International Conference on Software Engineering*, 2001.

[29] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzé, "Cross-checking oracles from intrinsic software redundancy," in *proceedings of the International Conference on Softwrae Engineering*, 2014.

[30] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "ARTOO: Adaptive random testing for object-oriented software," in *Proceedings of the International Conference on Software Engineering*, 2008.

[31] M. Staats, G. Gay, and M. Heimdahl, "Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing," in *Proceedings of the International Conference on Software Engineering*, 2012.

[32] P. Loyola, M. Staats, I.-Y. Ko, and G. Rothermel, "Dodona: automated oracle data set selection," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2014.

[33] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *proceedings of the European Conference on Object-Oriented Programming*, 2005.

[34] C. Csallner, Y. Smaragdakis, and T. Xie, "Dsd-crasher: A hybrid analysis tool for bug finding," *ACM Transactions on Software Engineering Methodology*, vol. 17, no. 2, pp. 1–37, 2008.

[35] M. Staats, S. Hong, M. Kim, and G. Rothermel, "Understanding user understanding: determining correctness of generated program invariants," in *proceedings of the International Symposium on Software Testing and Analysis*, 2012.