Enabledness-based Testing of Object Protocols

JAVIER GODOY, Universidad Nacional de General Sarmiento and Departamento de Computación, FCEyN-UBA, Argentina
JUAN PABLO GALEOTTI and DIEGO GARBERVETSKY, Departamento de Computación, FCEyN-UBA and ICC CONICET, Argentina
SEBASTIÁN UCHITEL, Departamento de Computación, FCEyN-UBA and ICC CONICET, Argentina and Department of Computing, Imperial College London, UK

A significant proportion of classes in modern software introduce or use object protocols, prescriptions on the temporal orderings of method calls on objects. This article studies search-based test generation techniques that aim to exploit a particular abstraction of object protocols (enabledness preserving abstractions (EPAs)) to find failures. We define coverage criteria over an extension of EPAs that includes abnormal method termination and define a search-based test case generation technique aimed at achieving high coverage. Results suggest that the proposed case generation technique with a fitness function that aims at combined structural and extended EPA coverage can provide better failure-detection capabilities not only for protocol failures but also for general failures when compared to random testing and search-based test generation for standard structural coverage.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Empirical software validation**; **Search-based software engineering**;

Additional Key Words and Phrases: Automatic test generation, enabledness-preserving abstractions, genetic algorithms

ACM Reference format:
Javier Godoy, Juan Pablo Galeotti, Diego Garbervetsky, and Sebastián Uchitel. 2020. Enabledness-based Testing of Object Protocols. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 12 (December 2020), 36 pages.
https://doi.org/10.1145/3415153

1 INTRODUCTION

An object protocol is a restriction on the order in which the methods of a particular object can be called [24, 87]. Consider, for instance, an object that implements the call protocol of the

This work was partially supported by ANPCYT PICT 2015-3638, 2015-1718, 2015-2741, 2018-3835, CONICET PIP 2014/16, 2015/17, UBACYT 2018-20020170200297BA, 20020170100419BA, 20020170200249BA.
Authors' addresses: J. Godoy, Universidad Nacional de General Sarmiento, Departamento de Computación, FCEyN-UBA, Buenos Aires, Argentina; email: jgodoy@dc.uba.ar; J. P. Galeotti, Departamento de Computación, FCEyN-UBA, ICC CONICET, Buenos Aires, Argentina; email: jgaleotti@dc.uba.ar; D. Garbervetsky, Departamento de Computación, FCEyN-UBA, ICC CONICET, Buenos Aires, Argentina; email: diegog@dc.uba.ar; S. Uchitel, Departamento de Computación, FCEyN-UBA, ICC CONICET, Buenos Aires, Argentina, Department of Computing, Imperial College London, London, UK; email: suchitel@dc.uba.ar.

1049-331X/2020/12-ART12 $15.00
https://doi.org/10.1145/3415153

`ListIterator` interface in Java [1]. Calls to methods `remove()` or `set()` will terminate normally (i.e., no exceptions) only if preceded by normal termination of calls to `next()` and `previous()`.

A recent study [24] of open-source projects suggests that classes that introduce object protocols are commonplace. Indeed, they found that up to "*7.2% of all types defined protocols, while 13% of classes were clients of types defining protocols. (For comparison, 2.5% of the types in the Java library define type parameters using Java Generics.)*"

Testing objects with protocol behaviour is particularly challenging independently of whether it is general failure detection or protocol violations that are being targeted [46, 49, 88, 96]. Complexity is related to generating valid objects representing different protocol states and exercising them appropriately. The interplay between the accumulated state resulting from a specific sequence of calls and the possible behaviours of the next method execution should be considered to avoid missing meaningful results or important functionality [24].

Automated test generation is an active research area [69] within which significant effort has been invested in automatic generation of tests in the form of call sequences (e.g., References [16, 59, 70, 88, 89]). These efforts are of particular interest to test objects with rich protocols, where specific sequences of methods, that are protocol compliant, must be generated to explore meaningful states.

A noteworthy tool for generating call sequences is Randoop [70], a purely random (and black box) approach that generates method sequences by reusing previously returned values and discarding sequences that terminate abnormally (i.e., exceptions). Although this increases the likelihood of reaching "deep" states within the object protocol, genetic algorithms aiming at structural coverage can outperform Randoop (e.g., References [40, 84]) at failure detection.

A hypothesis explored by many (e.g., References[29, 45, 48, 75, 81, 91]) is that aiming for protocol coverage can yield good results in terms of failure detection. Given that the actual protocol of a class is an infinite state machine, the challenge is first to define an appropriate finite abstraction of the protocol over which to compute coverage. Some approaches (e.g., References [45, 81, 91]) assume the existence of some user-provided abstract specification of the intended class protocol to automatically produce tests. Although there is a lack of empirical studies on the effectiveness of these approaches, a key factor is not only the correctness of the user provided specification but also the syntactic form that is given to the specification [47, 67, 76, 79, 80, 95].

In this article, we explore the *hypothesis that it is possible to automatically build an abstraction of the call protocol of an object under test and exploit it to automatically generate more effective test suites.* More specifically, we investigate the use of enabledness-preserving abstractions (EPAs) [30, 33] that quotient a potentially infinite object state space into a finite set of abstract states by grouping concrete states that enable the same set of method invocations. Transitions are labelled with method names and represent normal termination of method calls. The EPA over-approximates the sequences of normally terminating method calls. As EPAs are an abstraction of the object's behaviour rather than an abstraction of the code, they provide a basis for semantic notions of coverage that have the potential to avoid pitfalls of syntactic coverage of human-constructed specifications.

Czemerinski et al. proposed a conformance testing adequacy criteria based on covering EPAs [28]. They showed a correlation between transition coverage in the EPA of the intended object protocol and the ability of test suites to detect protocol conformance failures.

In this article, we investigate how to build on these results to produce test case generation techniques that can provide improved failure-detection capabilities. Indeed, we extend EPAs (xEPAs) to include transitions that model abnormal method termination (i.e., exceptions) and report on a test case generation technique that dynamically builds the xEPA of a class while simultaneously exploiting the xEPA to build, using search-based techniques, test suites that cover the xEPA. The technique requires the provision of methods to check if the precondition of each public method of the class under test holds. The contributions of this article are as follows:

(i) A test case generation technique that dynamically builds the xEPA of a class while simultaneously exploiting it to build, using search-based techniques, test suites that cover the xEPA.
(ii) Results showing that adjacent transition pair xEPA coverage can be combined with classical coverage criteria to provide better failure revealing capabilities when compared to standard coverage, state-of-the-art search based techniques and random techniques. Improvements are observed both for general failures and also for protocol failures on synthetic defects produced by mutations and real defects from existing benchmarks and found in the wild.

2 BACKGROUND

2.1 Object Protocols

de Caso et al. [33] proposed a full formal treatment of object protocols and enabledness preserving abstractions. Here we present an intuition that is sufficient for the purpose of this article. We start with Labeled Transition Systems, which we use to represent object protocols and their abstractions. A Labeled Transition System (LTS) is a tuple $\langle \Sigma, S, s_0, \delta \rangle$ where Σ is a set of labels, S a set of states, $s_0 \in S$ the initial state, and $\delta \subseteq S \times \Sigma \times S$ the transition relation.

An *object protocol* can be represented as a labeled transition system, where states represent the different values that the object's attributes can have, and transitions, labeled with method names and actual parameter values, represent how calls change the state of the object. The existence of an outgoing transition labeled with a specific method name m and parameter values $p_1, \ldots, p_n$ from a state c represents the fact that calling $m(p_1, \ldots, p_n)$ on the object with state with configuration c will not result in a runtime exception or undefined behavior.

As an example, consider the code for a bounded integer stack in Listing 1. In Figure 2, we depict a portion of the protocol transition system its object protocol defines. State c_1 is the initial state of a bounded stack object. If integer `2` is pushed, then the object will evolve to c_2, while if `1` is pushed the next state is c_5. Other integers pushed when the object is in state c_1 will lead to other states (not depicted in the diagram). Note that in state c_1 there is no outgoing transition labelled `pop()` as this would produce an exception. Similarly, c_4 has no outgoing transitions labeled `push(i)` for any `i`.

Given the transition system for an object protocol, a valid call sequence is a sequence of calls $m^0(p_1^0, \ldots, p_i^0), \ldots, m^n(p_1^n, \ldots, p_j^n)$ that is accepted by the transition system; in other words, that it is possible to reproduce the sequence by following transitions starting from the initial state. An example of a valid object call sequence for the bounded stack is `push(2), push(1), pop(), push(3)`. An example of an invalid call sequence is `push(0), pop(), pop()`.

2.2 General and Protocol Failures

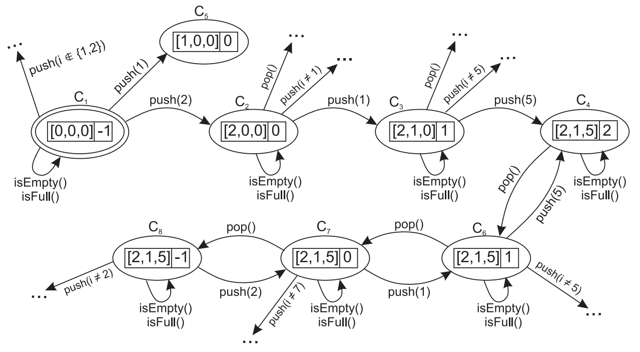
A failure corresponds to an observable deviation in the execution of a program from its intended behaviour. Such deviation may, for instance, lead to the unexpected termination of a program, or a method call that returns a different value from the expected return value. In this article, we are interested in a particular class of failures: those that exhibit that an object implements an object protocol that is different from its intended protocol. We refer to these as *protocol failures* as opposed to *general failures*.

More precisely, we define a *protocol failure* as follows: Given the transition system for the intended protocol of an object and an implementation of the object, we say a sequence of calls is a protocol failure if either (i) it is valid according to the intended call protocol yet raises an exception

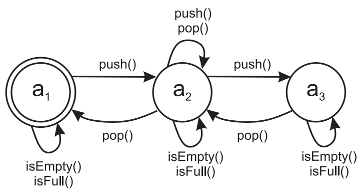
```java
public class BoundedStack {
  private final int[] elements;
  private int index;

  public BoundedStack() {
    elements = new int[3];
    index =-1;
  }
  public void push(int i) {
    if (isFull())
      throw new IllegalStateException();
    elements[++index] = i;
  }
  public int pop() {
    if (isEmpty())
      throw new IllegalStateException();
    return elements[index--];
  }
  public boolean isFull() {
    return index==elements.length-1;
  }
  public boolean isEmpty() {
    return index==-1;
  }
}
```

Fig. 1. An implementation of a bounded stack.



(a) The object protocol for the BoundedStack consisting of all correctly terminating (i.e., no exceptions) sequences of calls.



(b) The EPA for the BoundedStack

Fig. 2. Object protocol and EPA for the BoundedStack.

when executed on the implementation or (ii) it is invalid according to the intended protocol yet does not raise an exception when executed on the implementation.

Note that any defect may lead to a protocol failure, or, in other words, there is no clean cut distinction to be made between defects in terms of whether they introduce protocol or non-protocol failures. Consider replacing the returned expression `index--` of `pop()` in Figure 1 with `--index`. Such a defect would make a test sequence `push(2)`, `push(1)`, `assert(pop() == 1)` produce a general failure. However, the execution of the sequence of method calls `push(2)`, `pop()` would produce an index out of bounds exception that constitutes a protocol failure if the intended protocol of the object is that of Figure 2(a).

2.3 Enabledness-preserving Abstractions

Given a set of methods $M = m_1, \ldots, m_n$, an EPA of an object protocol is a finite LTS $\langle M, 2^M, s_0, \delta\rangle$ where each state is characterised by a set of methods in M (i.e., $s \subseteq M$). We refer to the states of an

EPA as *abstract states*, because semantically they can be thought of as describing a set of concrete states of the object protocol. Indeed, EPAs quotient the concrete states of the potentially infinite state and non-deterministic object protocol into sets such that two concrete states are grouped together if they enable the same set of method calls. Figure 2(b) depicts the EPA for the bounded stack protocol.

The set of method calls enabled by a concrete state c is defined as all the methods m for which there exists parameters $p_0, \ldots, p_n$ such that $m(p_0, \ldots, p_n)$ is an outgoing transition of c in the object protocol transition system. In the bounded stack example, concrete states c_2, c_3, and c_5 to c_7 (Figure 2) have the same set of method calls enabled (namely, `pop`, `push`, `isEmpty`, `isFull`). Thus, they are all grouped in one abstract state of the protocol's EPA (state a_2, Figure 2(b)). Similarly, concrete states c_1 and c_8 represent empty stacks (despite having different internal representations) and are abstracted by state a_1.

In conclusion, the number of states in an EPA is bounded by $2^{|M|}$. Each state s represents a subset of M and abstracts a group of concrete object protocol states (those for which the enabled methods of the object state correspond exactly to s). All concrete states of an object is abstracted by exactly one abstract state.

Note that the notion of method enabledness refers to the existence of a parameter such that the method terminates normally. Thus, for instance, in two instances of a bounded integer set implementation, the abstraction will consider equivalent two sets that are full even when they contain different elements despite the fact that `add(int)` will terminate correctly for different integers (adding an element already in a set will not increase its size).

A transition labeled with method name m between abstract states a and a' appears in the enabledness preserving abstraction of an object protocol if and only if there is a concrete state of the object abstracted by a and parameters $p_0, \ldots, p_n$ such that when $m(p_0, \ldots, p_n)$ is called on the object, its state is changed in such a way that the object is now abstracted by a'. In the bounded stack example, there is a transition from state a_2 back to a_2 labelled push, because concrete state c_2, abstracted by a_2, when applied `push(1)` results in a concrete state c_3 that is also abstracted by a_2.

Note that EPAs may accept method sequences for which there are no possible parameters to make then finish normally. An example is sequence *push*, *pop*, *pop* for which no value for method `push(i)` can be selected to make the second `pop()` not fail. However, any sequence of calls accepted by the concrete object protocol is guaranteed to be feasible in the EPA. Over-approximation of the actual protocol is the price to pay for a compact finite representation.

2.4 Search-Based Test Generation

The field of search-based software testing (SBST) [64] is concerned with the application of efficient search algorithms to generate test cases. Genetic Algorithms (GA) [86] are one of the most commonly applied classes of search algorithms. The intuition behind GAs is to imitate the natural process of evolution. Populations of candidate solutions are evolved by probabilistically applying domain specific operators that mimic mutation and recombination via crossover. The parents for individuals of the next generation are chosen based on their *fitness* to bias the survival of the fittest. The GA continually improves the fitness of the population until either an optimal solution is reached or a stopping criterion is met (e.g., maximum number of fitness evaluations or time limit). In the domain of evolutionary testing, a population would represent a set of test cases and the fitness measures how close a candidate solution is to satisfy a coverage goal.

Fitness functions are used to increase the likelihood of choosing promising individuals for reproduction and thereby gradually improving the fitness of each generation. In evolutionary testing, fitness functions for branch coverage [64] usually integrate the *approach level* (number of unsatisfied control dependencies) and the *branch distance* (heuristic for how close the deviating condition is to evaluating as desired). Such search techniques are not restricted to the context of primitive datatypes; they have also been applied to test object-oriented software using method sequences [43, 89].

The conventional approach to SBST is to search for test cases that satisfy a single coverage goal in isolation. Since limited computational resources need to be allocated to the entire set of coverage goals, dead code, and unsatisfiable branch conditions could waste a lot of runtime that could help to cover feasible branches. EvoSuite [39] tries to fix this issue by optimizing an entire test suite at once toward satisfying a coverage criterion and therefore prevents results from being adversely affected by the order, difficulty or infeasibility of individual coverage goals.

The Genetic Algorithm implemented in EvoSuite finishes when all the testing goals have been reached or the specified time budget has been exhausted. After completion, EvoSuite produces a JUnit test suite with one of the fittest (i.e., best) individuals found.

EvoSuite coverage criteria include structural criteria such as regular line and branch and also exception coverage. The goal of this criterion is to exercise all possible exceptions thrown at every program line of the targeted class. In contrast to line and branch criteria this criterion cannot be specified as a percentage, since the number of potential exception for a given line is unknown.

Multiple coverage criteria can be combined natively in EvoSuite [77]. Given $f_1, \dots f_n$ fitness functions for each selected criterion, the overall fitness function is computed as:

$$f_{comp}(TestSuite) = \sum_{i=1}^{n} (f_i(TestSuite)).$$

In turn, the user can select which criteria he or she would like to use during test generation.

By default, EvoSuite computes f_{comp} for a test suite $TestSuite$ as the sum of the fitness function for maximizing line coverage (namely, f_{LC}), the fitness function for maximizing branch coverage (namely, f_{BC}) and the fitness function for exception coverage (namely, f_{EC}).

The fitness function for maximizing line coverage is defined as follows:

$$f_{LC}(TestSuite) = \nu(|NCLs| - |CoveredLines|) + \sum_{b \in B_{CD}} \nu(d_{min}(b, TestSuite)),$$

where $NCLs$ is the set of all non-commented lines of codes of the target class, $CoveredLines$ is the total set of lines covered by the execution traces of every test in $TestSuite$, and $\nu(x)$ is the normalization function $\nu(x) = x/(x+1)$. B_{CD} is the set of all conditional statements that are control dependent to some other statement in the code. $d_{min}(b, TestSuite)$ is the minimum branch distance among all observed executions to every branch b in B_{CD}. Given the definition of f_{LC}, it follows that for any test suite it holds that $0 \leq f_{LC} < 1 + \|B_{CD}\|$.

The fitness function for the branch coverage criterion estimates how close a test suite is to cover all branches in the target class:

$$f_{BC}(TestSuite) = \sum_{b \in B} \nu(d(b, TestSuite)),$$

where B is the set of all branches in the target class and $d(b, TestSuite)$ is computed as follows:

$$d(b, TestSuite) = \begin{cases} 0 & \text{if the branch has been covered} \\ \nu(d_{min}(b, TestSuite)) & \text{if the predicated has been executed at least twice}. \\ 1 & \text{otherwise} \end{cases}$$

Therefore, for any test suite it holds that $0 \leq f_{BC} < \|B\|$.

Finally, the fitness function for the exception coverage criterion:

$$f_{EC}(TestSuite) = \frac{1}{1 + N_E},$$

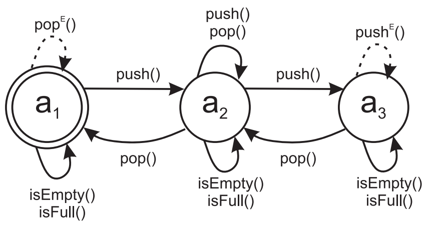


Fig. 3. The Exception EPA for the BoundedStack.

where N_E is the number of different exceptions thrown by a method of the target class during the execution of all tests in $TestSuite$. Notice that $0 < f_{EC} \leq 1$. The closer the value to 0, the more exceptions are thrown by methods in the test suite. In contrast, a value of f_{EC} equal or closer to 1 indicates that the test suite signals a small number of such exceptions.

All the aforementioned fitness functions (i.e., f_{LC}, f_{BC}, and f_{EC}) are all minimisation functions. In other words, the lower the value of the function, the closer the test suite to satisfying the target test goals (i.e., covering lines, branches, and exceptions, respectively).

3 ENABLEDNESS AND EXCEPTION PRESERVING ABSTRACTIONS

In this section, we introduce an extension to Enabledness Preserving Abstractions (xEPAs) that accounts for abnormal termination of method calls.

Our goal is to exploit an abstraction of the object protocol of a class to automatically generate test suites that are effective in detecting general and in particular protocol failures. The idea of covering the transitions of an EPA to find protocol violations has one potential flaw. Covering EPA transitions may help finding regressions in which protocol failures that correspond to sequences that should not raise exceptions but do so in subsequent code versions. However, covering EPA transitions is not geared to finding the other kind of protocol failures: regressions in which call sequences should raise exceptions but do not.

In other words, if the only tests that are run are tests that conform to the intended protocol, then the only protocol failure regressions that can be found are those in which changes make the call protocol raise exceptions where it should not. This could incorrectly lead to the conclusion that a more permissive implementation (e.g., one accepting *any* method call sequence without raising exceptions) is free from defects.

Ideally, what we aim to cover are all the transitions that are not present in the intended object protocol transition system. For instance, the non-existence of an outgoing transition pop from a concrete state c_1 (recall Figure 2) should be covered by a test that aims to confirm that popping an element out of the empty bounded stack raises an exception. The problem is that there are a potentially infinite number of transitions missing from an object protocol.

In the same way EPAs finitise the valid object protocol calls, an extension to EPAs could be defined to have a transition labeled m^E between two abstract states a and a' if it is possible to go from a to a' via calling method m with some parameter and raising an exception.

More precisely, given an EPA for an intended object protocol with methods $M = m_1, \ldots, m_n$ of the form $\langle M, 2^M, s_0, \delta\rangle$, an *EPA extended with exceptions* (or xEPA) is a transition system of the form $\langle M \cup M^E, 2^M, s_0, \delta \cup \delta^E\rangle$ where $M^E = \{m_1^E, \ldots, m_n^E\}$, $\delta^E \subseteq (2^M \times M^E \times 2^M)$, and $(a, m, a') \in \delta^E$ if and only if there are parameters $p_1, \ldots, p_n$ such that an object in a state abstracted by a when executing $m(p_0, \ldots, p_n)$ *raises an exception* and changes its state to one that is abstracted by a'.

In Figure 3 we depict the xEPA for the BoundedStack. Note the additional transitions from states a_1 and a_3 labelled $pop^E()$ and $push^E()$ that represent the fact that pop() and push() can raise exceptions if called on an empty and full stack. Note also that in both cases neither `pop()` nor `push()` modify the state of the stack despite raising an exception, hence the looping $pop^E()$ and $push^E()$ transitions.

An xEPA may help detect protocol violations when changes in the code make the object more permissive (i.e., less exceptions) than intended. For instance, in the Bounded Stack example, the test sequence `new BoundedStack(); pop();` would cover the transition (a_1, pop^E, a_1) of the xEPA, since the execution of `pop()` will signal an `IllegalStateException` for any concrete Bounded Stack that is abstracted by EPA state a_1 (i.e., any empty stack). Note that covering the transition (a_1, pop^E, a_2) (i.e, the situation in which popping an element from an empty stack raises an exception but lands the object in a non-empty state) is impossible for the code listed in Figure 1.

4 OBJECT PROTOCOL COVERAGE CRITERIA

In this section, we define various coverage criteria for EPAs and xEPAs. In the following sections we discuss an approach based on a genetic algorithm that attempts to maximise these criteria and then evaluate the extent to which this impacts failure detection.

4.1 EPA Transition Coverage

Test adequacy criteria based on model coverage [93] has been studied extensively to understand the degree to which it serves as a predictor of the failure detection capabilities of a given test suite. Classical protocol testing approaches are defined over some form of transition system [57]. A popular criteria is *all transitions* coverage [91]. This criterion is satisfied when the test suite exercises all transitions of the transition system.

As previously mentioned, Czemerinski et al. [28] proposed a conformance testing adequacy criteria based on covering EPA transitions. Given an EPA for a target class, the EPA transition coverage for a test suite TS is defined as follows:

$$\frac{|covered\_txs|}{|\delta|},$$

where δ is the set of EPA transitions, and $covered\_txs$ is the set of EPA transitions exercised during the test suite execution. A transition $(a, m, a') \in \delta$ is *exercised* if there is at least one test case t in the test suite such that the execution of t forces at some point calls method m on an object in concrete state that is abstracted by a and as a result the object evolves to a concrete state that is abstracted by a'. Furthermore, the execution of m does not signal an exception (i.e., the execution of method m finishes normally). Note that the parameters used to call m are not relevant in this notion of coverage, nor is the return value of the method. They key concept is normal versus exceptional termination.

4.2 xEPA Transition Coverage

In Section 3, we argued that an extension of EPAs to include methods that terminate abnormally (xEPAs) may help for detecting protocol violations of implementations that are more permissive than intended. Given an xEPA for a target class, the xEPA Transition coverage is defined as follows:

$$\frac{|covered\_txs| + |covered\_exception\_txs|}{|\delta| + |\delta^E|},$$

where $covered\_exception\_txs$ is the set of exception transitions (a, m^E, a') such that there is at least one test case t in the test suite such that by executing t the object under test reaches a concrete state that is abstracted by a and as a result of calling m an exception (of any kind) is raised and the object evolves to a concrete state that is abstracted by a'.

Note that this criterion does not distinguish the type of exception with which a method call terminates. It could be argued that abstracting termination only into successful and non-successful categories is too coarse grained. This reasoning could be taken further to also consider that method outputs (to further distinguish successful terminations) should also be considered. Indeed, many decisions can be made to calibrate the level of abstraction at which the object call protocol should be covered. Such calibration involves a tradeoff between the difficulty of covering a larger space that may or may not pay off in terms of actual effectiveness of test suites. Given the variety of exception types that classes can use (consider all unchecked exception types), this number would grow significantly, and we conjecture that overall it may become detrimental.

4.3 xEPA Transition Pairs Coverage

The criteria defined above do not take into account which method sequences were considered to reach a given transition. For instance, two test sequences tc_1 = `new BoundedStack(); push(2)` and tc_2 = `new BoundedStack(); push(2); pop(); push(7);` cover the transition $(a_1, push, a_2)$. Nevertheless, tc_1 exercises that transition from a newly created bounded stack while tc_2 exercises the transition on an emptied stack. Both scenarios are worth testing.

Similarly, this also applies to exception transitions. Given the exception transition (a_1, pop^E, a_1), consider test sequences tc'_1 = `new BoundedStack(); pop()` and tc'_2 = `new BoundedStack(); push(2); pop(); pop();`. These sequences are both potentially relevant as tc'_1 exercises the exception transition from a fresh stack, while tc'_2 operates on an emptied stack.

A pair is defined as a tuple $\langle t_i, t_j \rangle$, where $t_i \in \delta$ is a (normal) transition, and $t_j \in \delta \cup \delta^E$ is a normal or an exception transition. Additionally, the destination state of t_i ($dest(t_i)$) is required to match the source state of t_j ($source(t_j)$).

For example, in the bounded stack $\langle (a_1, push, a_2), (a_2, pop, a_1) \rangle$ is a valid transition pair, while $\langle (a_1, pop^E, a_1), (a_1, push, a_2) \rangle$ is not, as the definition requires the first element of the tuple to not be an exception transition.

Given these definitions, the xEPA transition pairs test adequacy criterion will be satisfied when for each valid transition pair, the test suite contains at least one test case that exercises that transition pair. Note that this criterion will not aim to cover pairs $\langle t_i, t_j \rangle$ where t_i is exception transition. This decision is due to the fact that covering these transitions would require tests that perform exception handling, which can be very complex to automate in some cases.

Following the previous definition, we define the xEPA Transition Pairs Coverage as follows:

$$\frac{|covered\_txs\_pairs|}{|all\_txs\_pairs|},$$

where *all_txs_pairs* is the set of connected pairs in the xEPA defined as: $\{\langle t_i, t_j \rangle |\ t_i \in \delta \wedge t_j \in \delta \cup \delta^E \wedge\ dest(t_i) = source(t_j)\}$ and *covered_txs_pairs* is the set of actual pairs exercised by the test suite.

5 EPA AND XEPA COVERAGE TEST GENERATION

In this section, we describe how we implemented our proposed approach as an extension to the EvoSuite version 1.0.6 test generator. We refer to it as Evo+EPA.

To achieve EPA and xEPA coverage, we do not rely on a third party (manual or automated) EPA/xEPA construction to guide test generation. Instead, generated tests are used to infer an under-approximation (i.e., less states and transitions) of an EPA/xEPA, which in turn is used to generate more tests.

Instead of requiring a full EPA specification, the user of our approach is only required to provide a instrumentation in the form of Boolean queries, one for each public method, that return when the

```
@EpaActionPrecondition(name = "push()")
private boolean isPushEnabled() {
  return !isFull();
}
@EpaActionPrecondition(name = "pop()")
private boolean isPopEnabled() {
  return !isEmpty();
}
@EpaActionPrecondition(name = "isEmpty()")
private boolean isEmptyEnabled() {
  return true;
}
@EpaActionPrecondition(name = "isFull()")
private boolean isFullEnabled() {
  return true;
}
```

(a)

```
@Test
public void test() {
 BoundedStack s0 = new BoundedStack();
 BoundedStack s1 = new BoundedStack();
 s0.push(0);
 s0.pop();
 assertEquals(s1.isEmpty(),s0.isEmpty());
 assertEquals(s1.isFull(),s0.isFull());
}
```

(b)

Fig. 4. (a) Enabledness Boolean queries for each method and (b) a test case that exercises two bounded stacks with different EPA traces.

method is enabled (Section 5.1). Using these queries, a test can be run while tracking what part of an EPA/xEPA is being covered, thus allowing not only the construction of an under-approximation of the implementation's EPA/xEPA (Section 5.2). but also exploiting appropriate fitness function (Section 5.3) to increase EPA/xEPA coverage.

In conclusion, the outcome of Evo+EPA is twofold, a test suite built to achieve EPA/xEPA coverage, and an inferred under-approximation of the EPA implemented by the class under test.

5.1 Enabledness Boolean Queries

Boolean queries [59] are argument-less instance (i.e., non-static) pure (i.e., no side effects) methods that return a `boolean` value. Given an action *m*, an *enabledness* Boolean query specifies if the current object has action *m* enabled. That is, if the Boolean query of *m* returns true then there exist a combination of parameters that can be used to invoke *m* such that its execution terminates normally. In contrast, if *m* returns false (or throws any exception), then all executions of *m* will throw an exception.

Figure 4(a) shows four of such enabledness Boolean queries, one for each method in `BoundedStack`. Each enabledness Boolean query is annotated with an `@EpaActionPrecondition` annotation to specify which action refers to. Notice that as constructors cannot be invoked once the instance has been initialised, they are only enabled in the initial EPA state. Boolean queries can be manually written (as the ones described in Figure 4(a)) or automatically extracted from the predicates obtained by tools that build EPAs such as Contractor [33]. In this article, we rely on manual construction of queries and report on the manual effort involved in Section 6.1.5.

Enabledness Boolean queries must be provided for all SUT public methods. By checking the Boolean values these queries return at a particular point in a test, it is possible to know exactly in what abstract state of the EPA/xEPA an object is currently at (recall that an abstract state characterises all concrete objects that enable the same set of methods).

5.2 Dynamically Inferring an EPA/xEP

Our extension of EvoSuite infers dynamically an under-approximation of the implementation EPA/xEPA and outputs it an XML document. Figure 5 shows an excerpt of the XML document *automatically* generated during the generation of test cases for the `BoundedStack` class.

When the test generation begins, an initial model is synthesised with a unique initial state (named `_INITIAL_STATE`) where only class constructors are enabled.

```
<?xml version="1.0" encoding="utf-8"?>
<abstraction initial_state="_INITIAL_STATE" name="null">
  <label name="BoundedStack()" />
  <label name="isEmpty()" />
  <label name="isFull()" />
  <label name="pop()" />
  <label name="push()" />
  <state
  name="[BoundedStack()=false,isEmpty()=true,isFull()=true,pop()=false,push()=true]">
    <enabled_label name="isEmpty()" />
    <enabled_label name="isFull()" />
    <enabled_label name="push()" />
    <transition
    destination="[BoundedStack()=false,isEmpty()=true,isFull()=true,pop()=false,push()=true]"
     label="isEmpty()" />
    <transition
    destination="[BoundedStack()=false,isEmpty()=true,isFull()=true,pop()=false,push()=true]"
     label="isFull()" />
    <transition
    destination="[BoundedStack()=false,isEmpty()=true,isFull()=true,pop()=true,push()=true]"
    label="push()" />
  </state>
 ...
</abstraction>
```

Fig. 5. An excerpt of the automatically generated XML document specifying the dynamically inferred EPA for class BoundedStack.

Every time a new test is created, the test is run and enabledness Boolean queries are used to infer the state of the EPA/xEPA that are traversed. Inference is performed after each call to a public method (e.g., m) of the SUT. Thus, the EPA/xEPA source and target states for a transition labelled with the public method can be inferred (namely, the $\langle a_i, m, a_j\rangle$ or $\langle a_i, m^E, a_j\rangle$ depending if m terminated normally or not). Thus, the execution of the test determines a path in the EPA/xEPA being inferred. If the current representation of the EPA/xEPA cannot accept the path, then it is extended by adding missing states and transitions accordingly.

Note that tests may instantiate several objects (see Figure 4(b)), thus EPA/xEPA path inference must be done on a per object basis. Besides, each test can contribute more than one path to the EPA/xEPA under construction.

Thus, during the search-based test generation, new tests cases are generated and executed to continuously add states and transitions to the under-approximated EPA/xEPA. When the test generation process finishes, our prototype writes an XML file with the model. Figure 5 shows an extract of the XML obtained for BoundedStack. Observe that each state is represented unambiguously as a ordered list of the protocol actions and its enabledness value for that particular state. For example, the state named as ‘‘[BoundedStack()=false, isEmpty()=true, isFull()=true, pop()=false, push()=true]’’ stands for the EPA state where actions BoundedStack() and pop() are disabled, while actions isEmpty(), isFull(), and push() are enabled.

5.3 Fitness Functions for EPA/xEPA Coverage

To guide test generation toward covering EPA/xEPA models, fitness functions must be defined. These functions will be used by the genetic algorithm implemented in EvoSuite.[1] Next, we define fitness functions for maximizing each of the test adequacy criterion defined in Section 4.

[1]By default, EvoSuite uses a Monotonic GA [26].

As EPAs/xEPAs are not be provided to Evo+EPA, it is necessary to define fitness functions in a way such that (i) it is not necessary to know the EPA/xEPA to compute its fitness value and (ii) by optimising the fitness value the corresponding coverage criterion (defined in Section 4) increases. Of course, the functions can rely on the runtime information regarding the states and transitions that generated test have covered in the under-approximated EPA/xEPA that is being inferred (as explained in the Section 5.2).

The fitness function for maximizing EPA transition coverage (Section 4.1) is defined as follows:

$$f_{EPAC}(TestSuite) = \frac{1}{1 + |covered\_txs|}.$$

The lower the value of this function, more EPA transitions will be covered. The set $covered\_txs$ contains all EPA transitions traversed during the execution of all the tests in $TestSuite$. Observe that knowing the complete set of EPA transitions (i.e., δ) it is not necessary to compute f_{EPAC}.

We define the function for maximizing xEPA coverage (described in Section 4.2) as:

$$f_{EPAXC}(TestSuite) = \frac{1}{1 + |covered\_txs| + |covered\_exception\_txs|},$$

where $covered\_exception\_txs$ is the set of all exceptional transitions (regardless of the type of the thrown exception) observed during the execution of a tests suite. Again, the lower the value of f_{EPAXC} the higher the number of exceptional and normal transitions in δ and δ^E that are covered. In addition, the total number of transitions of the true xEPA is not required to compute f_{EPAXC}.

Similarly, we define the fitness function for maximizing the xEPA transition pairs (defined in Section 4.3) as:

$$f_{EPAXPC}(TestSuite) = \frac{1}{1 + |covered\_txs\_pairs|}.$$

As previously described, EvoSuite allows to natively combine multiple coverage criteria into a single aggregated fitness function that guides the test generation. The only restriction when combining multiple coverage criteria is that all of them should be minimization (respectively, maximization) functions. Notice that f_{LC}, f_{BC} and f_{EC} are all minimization functions (i.e., the smaller the value of the function, the higher chances are of more covered goals). Given the implemented fitness functions (f_{EPAC}, f_{EPAXC}, and f_{EPAXPC}) we can combine them with other fitness functions in EvoSuite (such as f_{LC}, f_{BC}, or f_{EC}).

Therefore, we define the following configurations of our EvoSuite prototype using the following composed fitness functions for a test suite:

- Evo+EPA: $f_{LC} + f_{BC} + f_{EC} + f_{EPAC}$
- Evo+EPAx: $f_{LC} + f_{BC} + f_{EC} + f_{EPAXC}$
- Evo+EPAxp: $f_{LC} + f_{BC} + f_{EC} + f_{EPAXPC}$

Additionally, we will refer to Only-EPA as the EvoSuite test generator using exclusively the fitness function f_{EPAC} to guide the generation, Only-EPAx and Only-EPAxp will refer respectively to the EvoSuite prototype using exclusively f_{EPAXC} and f_{EPAXPC}.

6 EVALUATION

In this section, we report on an evaluation that aims to assess the degree to which the test case generation approach proposed in the Section 5 is effective for failure detection. To compensate for the lack of benchmarks that both contain real defects and are aimed at classes that implement object protocols we split the evaluation into two. We first study effectiveness of our proposed coverage criteria on classes identified by third parties as implementing object protocols. These classes do not have real defects, hence we use mutations to perform a statistical analysis of failure detection effectiveness over synthetic defects. We then study the best performing criterion against a selected subset of well-known defects (taken from the Defects4J benchmark), and we also add a small number of real defects for classes that implement non-trivial object protocols to provide an alternative, albeit not statistically relevant, viewpoint.

6.1 Mutation Study on Subjects with Third-party EPAs

We report on a study to assess effectiveness in failure detection on mutations of classes that implement object protocols compared to other tools. Although there are many tools that use code abstractions to guide test case generation (i.e., Model Based Testing [58, 63, 82, 93]) only some of them target the Java language (e.g., References [2, 22, 53, 92, 97]). However, in all cases, they require a full model of the unit under test to be provided manually. None, use enabledness Boolean queries as the basis for the abstraction. To use them, specific manually produced models would be required, introducing further bias. Therefore, we decided to dismiss these MBT tools for Java for our experiments.

We first study detection of general failures and then focuses on protocol failures. In both cases we first compare against a vanilla search based approach (baseline), then against state of the art search-based approaches and finally against a random approach.

6.1.1 Research Questions. The research questions that study general failure detection are described below:

- **RQ#1.1:** Can enableness-based test generation improve *general* failure detection compared to a vanilla search-based approach?
- **RQ#1.2:** How does the best enabledness-based criterion from **RQ#1.1** compare to other state-of-the-art search-based approaches for *general* failures?
- **RQ#1.3:** How does the best enabledness-based criterion from **RQ#1.1** compare to a purely random approach in terms of *general* failure detection?

Once the effectiveness on general fault detection is studied, we focus on failures that are particularly relevant for object protocols: protocol failures. The following questions are identical to those above, but for protocols failures.

- **RQ#2.1:** Can enableness-based test generation improve *protocol* failure detection compared to a vanilla search-based approach?
- **RQ#2.2:** How does the best enabledness-based criterion from **RQ#2.1** compare to other state-of-the-art search-based approaches for *protocol* failures?
- **RQ#2.3:** How does the best enabledness-based criterion from **RQ#2.1** compare to a purely random approach in terms of *protocol* failure detection?

Considering that structural coverage criteria are widespread (and arguably standard) in both academia and industry, a potentially pernicious side-effect of striving for EPA/xEPA coverage is the reduction of structural coverage. Thus, we assess if the test case generation approach proposed in Section 5 impacts structural coverage negatively.

- **RQ#3:** Is there a negative impact in terms of structural coverage when including the best enabledness-based criterion from **RQ#1.1** and **RQ#2.1** to a search-based approach?

6.1.2 Subjects. We use 12 subjects that were previously used in the literature. Each of these subjects corresponds to a Java class that implements an object protocol. We selected all case studies from all papers studying EPAs: Seven subjects were taken from the work of Krka et al. [54] on

Table 1. Subject Summary

Subject	LOC	$\|B\|$	$\|M\|$
JDBCResultSet [28]	518	176	75
ListItr [28]	124	22	23
NFST [54]	32	18	05
SftpConnection [54]	281	64	33
Signature [28] [54]	92	28	20
SMTPProcessor [28]	571	340	57
SMTPProtocol [54]	141	41	18
Socket [28] [54]	237	90	35
StackAr [54]	27	10	07
StringTokenizer [54]	109	68	11
ToHTMLStream [54]	274	118	11
ZipOutputStream [54]	205	82	14

LOC is lines of code, $|B|$ is number of branches, $|M|$ is number of methods.

automatic mining of EPA models, while five subjects were taken from the article by Czemerinski et al. [28]. These subjects also appear in other papers (e.g., References [24, 33, 56, 74]).

Table 1 provides an overview of each experimental subject. `Signature`, `ListItr`, and `Socket` were taken from the Java Development Kit implementation, the `JDBCResultSet` is an implementation of the `ResultSet` interface of the JDBC specification of the HyperSQL 2.0.0 database. The `SMTPProcessor` class is from the JES mail server 2.0, a Java SMTP and POP3 e-mail server. `SMTPProtocol` is an implementation of the client side SMTP protocol. `StringTokenizer` and `NumberFormatStringTokenizer` (`NFST` for short) are string data processing classes that can be found in JDK and the Apache XALAN [3] project respectively. `ZipOutputStream` allows the user to compress a stream of data while `ToHTMLStream` (also from Apache XALAN) serializes a series of SAX or SAX-like events and to the given stream. Finally, `SftpConnection` is a wrapper used to establish a secure file transfer among two endpoints.

Although EvoSuite offers support for handling enviromental dependencies using mocked versions [19], generated network data [20], and automatic synthesis of simple stubs [21], it was not sufficient to run our selected subjects properly. As an example, EvoSuite effectively handles the mocking of class `Socket`, but it does not allow to mock *inner* classes that are used in the `Socket` implementation. This is reasonable as this mechanism is not meant to generate tests targeting classes of the JDK library. Similarly, EvoSuite's current version does not support mocking database accesses. These mocks are necessary for the subject `JDBCResultSet`, or native calls (e.g., `Signature`, `ZipOutputStream`). Consequently, we manually wrote mocks for dependency classes in 8 of 12 subjects.

6.1.3 *Experimental Design.*

Intra-tool comparisons. For both **RQ#1.1**, **RQ#2.1**, and **RQ#3** we compare our extension of EvoSuite to the standard configuration of EvoSuite that aims to produce test suites with high statement and branch coverage. We also added exception coverage to this configuration [77]. The rationale for including exception coverage is that f_{epax} and f_{epaxp} target signalling exceptions. Therefore, exclusively generating a baseline without aiming for exception coverage would lead to an unfair comparison. We will refer to the execution of test generation using this specific combination (i.e., statement, branch and exception coverage) directly as "EvoSuite."

For both **RQ#1.2** and **RQ#1.3** we chose as representatives of alternative state-of-the-art search-based approaches the following EvoSuite configurations:

- EvoSuite *using a strong mutation criterion*: Strong mutation coverage [41] is a criterion based on mutation testing where a mutant (i.e., a syntactic change of the original SUT) is considered *strongly killed* if the execution of a test on the mutant leads to an externally visible difference w.r.t. the original program. This criterion is designed specifically to generate test suites good at discovering mutants. However its potential high overhead has discouraged its use in past studies [26, 77, 78]. We refer to this particular use of EvoSuite with the combined criterion that aggregates statement, branch, exception and strong mutation coverage as "Evo+SMC" (i.e., EvoSuite with strong mutation coverage). We included Evo+SMC, since we would like to contrast the failure detection capabilities of our approach (i.e., Evo+EPA, Evo+EPAx and Evo+EPAxp) against a criterion specially targeted at killing mutants.
- EvoSuite *using a many-objective optimisation algorithm*: In contrast to single-objective algorithms, many-objective search strategies treat each coverage goal as an independent optimisation objective. MOSA [71] is a variant of the multi objective genetic algorithm NSGA-II [34]. NSGA-II uses a preference sorting criterion to reward the best tests for each non-covered target, regardless of their dominance relation with other tests in the population. As MOSA [71] (and more recently DynaMOSA [72]) have been shown to result in higher coverage of some selected criteria than traditional genetic algorithms for whole test suite optimisation [26], we would also like to compare the performance of our approach against those test suites generated using the MOSA variant of EvoSuite aiming at statement, branch, exception and strong mutation coverage simultaneously. We will refer to this evosuite configuration in our experiments as "EvoMOSA."

Cross-tool comparisons. For **RQ#1.3** and **RQ#2.3**, we included Randoop[2] [70] as a representative of the state-of-the-art random test generators. As some studies suggest [84, 85], although evolutionary algorithms are more effective at covering complex branches, a random search may suffice to achieve high coverage of most object-oriented classes. We would like to study if for the case of object protocols, random search is more effective than the proposed approach for detecting failures. Note that Randoop is not an extension or a particular configuration of EvoSuite. Although this might introduce some difficulties for comparison (as tools could handle differently several implementation decisions such as generated assertions, string generation, etc.) this acts as an external validation to our approach.

Measuring test suite effectiveness with mutation analysis. We use mutation analysis [90] as a predictor to measure how many non-artificial defects each generated test suite can detect. In particular, we chose PITEST [27] for performing the mutation analysis on each generated test suite.

For protocol failures, we also perform mutant analysis to cope with the absence of an existing corpus of known protocol failures for our experimental subjects. Once the test generation concludes, EvoSuite automatically appends *assertions* on the observed return values of the invoked calls. To report the protocol failure capabilities of a given generated test suite, we systematically remove these assertions on each generated test suite and re-execute the PITEST mutation analysis tool. In this way, each mutant that is killed by the modified test case is due to the fact that (i) the sequence of calls throws an exception when it is not supposed to or (ii) the sequence of calls *does*

[2]Version 4.1.0.

not signal an exception when it is expected to do so. This matches the definition of protocol failure of Section 2.2.

In the aforementioned setting a mutant, (e.g., syntactic change in the SUT) could correspond to: (i) a protocol failure as defined above, (ii) a non-protocol failure (i.e., a change such that an *assert* on a return or field value is needed to detect it), or (iii) an equivalent mutant [60] (i.e., is a change to the SUT such that no test case exists that can uncover the change).

Parameter settings and statistical tests. We kept EvoSuite's default values for all parameters such as population size, type of selection mechanism, type of crossover, archive, and so on. For every selected generation algorithm (i.e., the previously described EvoSuite configurations plus Randoop), we executed the configuration with a budget of 10 minutes on each subject. We followed the guidelines described in Arcuri et al. [18] to statistically compare non-deterministic test generators. We repeated each execution of every generation algorithm 60 times with different random seeds. In total, this lead to 12 subjects × 10 algorithms × 60 repetitions × 10 minutes = 1,200 processing hours (≈50 days). This number of processing hours refers exclusively to the execution of the generation algorithms.

To compare if given two criteria #1 and #2, one performs better than the other we measure the effect size with the non-parametric hypothesis test Vargha-Delaney A_{12}. A_{12} defines the probability that running the criterion #1 yields better results than running criterion #2. If the two criteria are equivalent, then $A_{12} = 0.5$. If $A_{12} < 0.5$, then there is evidence that criterion #2 achieves better, while $A_{12} > 0.5$ evidences the opposite. To show statistical confidence, we compute the Wilcox-Mann-Whitney U-test where a p-value below <0.05 (values highlighted in bold in the tables) serves as evidence against the null hypothesis.

6.1.4 *Experimental Results.*

We now discuss results for each research question.

RQ#1.1: Can enableness-based test generation improve *general* failure detection compared to a vanilla search-based approach?

We first compare average mutation scores of different EPA/xEPA fitness functions to determine which is more effective at detecting general failures. We then use the best one in a statistical comparison agains EvoSuite.

Table 2 shows the average mutation score for EvoSuite using EPA/xEPA fitness function in *isolation* (i.e., those using only f_{EPAC}, f_{EPAXC}, or f_{EPAXPC}) and also *combined* with structural criteria. The table also ranks for each subject the various EvoSuite configurations. Numbers in parenthesis indicate the rank of one configuration against the rest when competing for one particular subject. The bottom line shows the average rank of each configuration across all subjects.

The results in Table 2 show that for all subjects but one (`ToHTMLStream`), the best ranking EvoSuite configuration uses f_{EPAXPC} either with or without additional structural criteria. In particular, the combined criteria Evo+EPAxp ranks first in 7 of 12 subjects and for the remaining 5, it ranks in the top 3. Its average rank is 1.5 compared to second best Only-EPAxp with an average rank of 2.67.

RQ#1.1: *Criterion* Evo+EPAxp *is the best performing* EvoSuite *configuration*

Having identified Evo+EPAxp, as the best performing candidate we report a statistical analysis of average mutation scores (i.e., μ) and Varga-Delaney measure of effect sizes (i.e., A_{12}) of EvoSuite versus Evo+EPAxp (see Table 3).

Results show that Evo+EPAxp outperforms EvoSuite with statistical significance in 9 of 12 subjects. In the remaining three there is no statistical evidence of either outperforming the other.

Table 2. Average Mutation Score of Different EvoSuite Configurations (RQ#1.1) for General Failures

Subject	EvoSuite	Only-EPA	Only-EPAx	Only-EPAxp	Evo+EPA	Evo+EPAx	Evo+EPAxp
JDBCResultSet	62.4% (6)	53.4% (7)	63.9% (5)	72.2% **(1)**	65.8% (4)	65.9% (3)	68.1% (2)
ListItr	75.7% (7)	79.4% (6)	84.2% (3)	85.1% **(1)**	83.3% (4)	83.0% (5)	85.1% **(1)**
NFST	91.8% (2)	75.5% (7)	80.2% (6)	91.7% (3)	91.3% (4)	91.0% (5)	92.3% **(1)**
SftpConnection	45.1% (3)	22.0% (7)	22.4% (6)	44.3% (5)	45.3% (2)	45.1% (3)	50.6% **(1)**
Signature	80.4% (6)	75.7% (7)	87.8% (5)	96.5% **(1)**	90.3% (3)	89.7% (4)	94.7% (2)
SMTPProcessor	35.5% (7)	38.5% (6)	45.9% (3)	54.6% **(1)**	41.2% (5)	44.9% (4)	53.2% (2)
SMTPProtocol	50.4% (4)	39.3% (7)	39.9% (6)	41.3% (5)	52.2% (2)	52.0% (3)	53.1% **(1)**
Socket	75.2% (7)	78.2% (6)	82.3% (2)	83.2% **(1)**	81.8% (5)	81.9% (3)	81.9% (3)
StackAr	88.2% (4)	87.4% (6)	87.4% (6)	99.9% (2)	88.2% (4)	88.6% (3)	100.0% **(1)**
StringTokenizer	53.8% (5)	39.5% (7)	47.4% (6)	56.0% (2)	54.1% (4)	54.2% (3)	57.2% **(1)**
ToHTMLStream	22.4% **(1)**	10.1% (6)	10.0% (7)	15.3% (5)	21.2% (4)	21.9% (3)	22.1% (2)
ZipOutputStream	32.5% (2)	14.4% (7)	15.3% (6)	19.0% (5)	32.3% (3)	32.3% (3)	33.2% **(1)**
Average Rank	4.5 (5)	6.58 (7)	5.08 (6)	2.67 (2)	3.67 (4)	3.5 (3)	1.5 **(1)**

Table 3. Average Mutation Scores and Effect Sizes of EvoSuite versus Evo+EPAxp (RQ#1.1) for General Failures

Subject	μ EvoSuite	Evo+EPAxp			
		μ	A_{12}	p-value	Improvement
JDBCResultSet	62.4%	68.1%	**0.01**	<0.0001	+9.1%
ListItr	75.7%	85.1%	**0.02**	<0.0001	+12.4%
NFST	91.8%	92.3%	0.49	0.9	+0.5%
SftpConnection	45.1%	50.6%	**0**	<0.0001	+12.2%
Signature	80.4%	94.7%	**0**	<0.0001	+17.8%
SMTPProcessor	35.5%	53.2%	**0**	<0.0001	+49.9%
SMTPProtocol	50.4%	53.1%	**0.17**	<0.0001	+5.4%
Socket	75.2%	81.9%	**0**	<0.0001	+8.9%
StackAr	88.2%	100.0%	**0.12**	<0.0001	+13.4%
StringTokenizer	53.8%	57.2%	**0.06**	<0.0001	+6.3%
ToHTMLStream	22.4%	22.1%	0.54	0.48	−1.3%
ZipOutputStream	32.5%	33.2%	0.45	0.31	+2.2%

Statistical significance (<0.05) is highlighted in bold.

Improvements in mutation scores are of up to 49.9%. Observe that, even when Evo+EPAxp achieves less mutation score than EvoSuite, the difference is very slim (i.e., −1.3%).

To gain more insight on these results, we would like to study how much coverage vanilla EvoSuite achieves by its own (i.e., without any help of the enabledness criteria described previously) over the intended xEPA model. Unfortunately, we do not have the intended xEPA model to compute the total number of possible adjacent pairs for each subject. However, we can mitigate the lack of an independent xEPA model by automatically computing an inferred approximation to this model. More specifically, we automatically build an approximated xEPA model for a particular subject by aggregation all the transitions covered using all test suites (generated using any EvoSuite configuration or using Randoop).

Table 4. Average Mutation Scores and Effect Sizes of Other Search-based Approaches versus Best Enabledness-based Criterion from RQ#1.1 (Evo+EPAxp) for General Failures (RQ#1.2)

Subject	Evo+	Evo+SMC (2)				EvoMOSA (3)			
	EPAxp (1)	μ	A_{21}	p-value	Improvement	μ	A_{31}	p-value	Improvement
JDBCResultSet	68.1%	49.9%	**0**	<0.0001	+36.5%	46.1%	**0**	<0.0001	+47.7%
ListItr	85.1%	75.7%	**0.04**	<0.0001	+12.4%	75.1%	**0.01**	<0.0001	+13.3%
NFST	92.3%	93.7%	**0.64**	<0.0001	−1.5%	91.2%	0.43	0.13	+1.2%
SftpConnection	50.6%	40.8%	**0**	<0.0001	+24%	36.0%	**0**	<0.0001	+40.6%
Signature	94.7%	81.5%	**0**	<0.0001	+16.2%	82.3%	**0**	<0.0001	+15.1%
SMTPProcessor	53.2%	31.7%	**0**	<0.0001	+67.8%	30.2%	**0**	<0.0001	+76.2%
SMTPProtocol	53.1%	49.2%	**0.11**	<0.0001	+7.9%	40.5%	**0**	<0.0001	+31.1%
Socket	81.9%	71.4%	**0**	<0.0001	+14.7%	70.8%	**0**	<0.0001	+15.7%
StackAr	100.0%	84.0%	**0.01**	<0.0001	+19%	86.0%	**0.05**	<0.0001	+16.3%
StringTokenizer	57.2%	55.0%	**0.18**	<0.0001	+4%	56.9%	0.45	0.29	+0.5%
ToHTMLStream	22.1%	14.8%	**0.04**	<0.0001	+49.3%	11.7%	**0**	<0.0001	+88.9%
ZipOutputStream	33.2%	26.0%	**0.04**	<0.0001	+27.7%	25.7%	**0.14**	<0.0001	+29.2%

Statistical significance (<0.05) is highlighted in bold.

Using the approximated xEPA model, we computed the average coverage of adjacent pairs for each subject achieved by EvoSuite. The subjects for which EvoSuite achieved higher average coverage of adjacent pairs matched those for which Evo+EPAxp failed to outperform EvoSuite with statistical significance (i.e., `NFST`, `ToHTMLStream` and `ZipOutputStream`). We conjecture that this could mean that structural coverage in these subjects suffices to achieve good xEPA coverage and that the gains that Evo+EPAxp can provide are limited for those particular subjects.

RQ#1.1: *Criterion* Evo+EPAxp *is statistically better than* EvoSuite *in 9 of 12 subjects with an increase of up to* 49.9% *in average fault detection.*

RQ#1.2: How does the best enabledness-based criterion from **RQ#1.1** compare to other state-of-the-art search-based approaches for *general* failures?

Table 4 shows the average mutation score for general failures (i.e., μ) and Varga-Delaney measure of effect sizes when executing the state-of-the-art approaches Evo+SMC (i.e., A_{21}) and EvoMOSA (i.e., A_{31}), compared to Evo+EPAxp.

Results show that Evo+EPAxp outperforms Evo+SMC with statistical significance for all subjects except `NFST`, with improvements of up to +67.8%. When compared to EvoMOSA, Evo+EPAxp fairs better, with statistical significance, in 10 of 12 subjects with improvements of up to +88.9%.

The only subject where one of the two approaches outperformed Evo+EPAxp with statistical significance is `NFST`. Observe that, even when Evo+SMC achieved better average fault detection, the gains are very small (i.e., −1.5%) while on every remaining subject where statistical significance was achieved, Evo+EPAxp obtained higher failure detection with a larger difference, with values going up to +88.9%. Therefore, we can conclude that, although it does not hold for all subjects, the benefits of applying Evo+EPAxp outperform the potential drawbacks in terms of fault detection.

RQ#1.2: Evo+EPAxp *outperformed* Evo+SMC *in 11 of the 12 subjects, while outperforming* EvoMOSA *in 10 of the 12 subjects. In both cases, increased performance of fault detection was in some cases over* 60%.

Table 5. Average Mutation Scores and Effect Sizes of a Purely Random Approach versus Best Enabledness-based Criterion from RQ#1.1 (Evo+EPAxp) for General Failures (RQ#1.3)

Subject	Randoop	Evo+EPAxp			
		μ	A_{12}	p-value	Improvement
JDBCResultSet	67.7%	68.1%	**0.38**	0.02	+0.6%
ListItr	6.4%	85.1%	**0**	<0.0001	+1229.7%
NFST	100.0%	92.3%	**0.98**	<0.0001	−7.7%
SftpConnection	0.0%	50.6%	**0**	<0.0001	—%
Signature	41.0%	94.7%	**0**	<0.0001	+131%
SMTPProcessor	31.2%	53.2%	**0**	<0.0001	+70.5%
SMTPProtocol	58.8%	53.1%	**1**	<0.0001	−9.7%
Socket	34.0%	81.9%	**0**	<0.0001	+140.9%
StackAr	100.0%	100.0%	0.5	1	0%
StringTokenizer	58.7%	57.2%	**0.84**	<0.0001	−2.6%
ToHTMLStream	22.0%	22.1%	0.53	0.48	+0.5%
ZipOutputStream	12.6%	33.2%	**0**	<0.0001	+163.5%

Statistical significance (<0.05) is highlighted in bold.

RQ#1.3: How does the best enabledness-based criterion from **RQ#1.1** compare to a purely random approach in terms of *general* failure detection?

Table 5 shows the average mutation score for general failures and effect sizes of Randoop compared to Evo+EPAxp. In 7 of 12 subjects, Evo+EPAxp outperforms Randoop with statistical significance. In 3 of 12 it is Randoop that outperforms Evo+EPAxp with statistical significance. There are 2 subjects for which there is no clear winner: `StackAr` for which both achieve 100% and `ToHtmlStream` where differences are marginal. Results of comparing Evo+EPAxp to Randoop are more mixed than previous comparisons.

The improvements of Evo+EPAxp over Randoop are very significant in many cases with various subject improving 2× or more. In contrast, the three improvements of Randoop over Evo+EPAxp were only 2.6%, 7.7%, and 9.7%.

Understanding the aspects that explain why one tool does better than the other in the different subjects is not simple as Randoop over Evo+EPAxp work quite differently. There are several features that could make the difference. As an example, Randoop and EvoSuite rely on different patterns for assertion generation, which could influence the ability of a generated test case to detect regressions. Similarly, they apply different techniques for generating string values.

RQ#1.3: Evo+EPAxp *fairs significantly better than* Randoop *on the majority of subjects.*

RQ#2.1: Can enableness-based test generation improve *protocol* failure detection compared to a vanilla search-based approach?

Similarly to general failures in **RQ#1.1**, we first compare different EPA/xEPA fitness functions to determine which is more effective at detecting protocol failures. We then use the best one in a statistical comparison against EvoSuite. Note that to avoid confusion with the standard definition of mutation score (μ), which is given as a percentage, we report absolute numbers of mutants for which we detect protocol failures (i.e., #pf) instead of the percentage with respect to the total number of mutants.

Table 6. Average Protocol Failure (#pf) Score for Different EvoSuite Configurations (RQ#2.1)

Subject	EvoSuite	Only-EPA	Only-EPAx	Only-EPAxp	Evo+EPA	Evo+EPAx	Evo+EPAxp
JDBCResultSet	122.4 (5)	91.6 (7)	121.5 (6)	135.7 **(1)**	123.3 (4)	123.5 (3)	134.6 (2)
ListItr	20.6 (7)	23.2 (6)	25.7 (3)	28.1 **(1)**	24.5 (5)	25.1 (4)	28.1 **(1)**
NFST	9.3 (5)	6.0 (7)	7.6 (6)	10.7 (2)	9.5 (3)	9.4 (4)	10.8 **(1)**
SftpConnection	20.6 (2)	8.8 (7)	9.0 (6)	19.4 (5)	20.6 (2)	20.3 (4)	21.9 **(1)**
Signature	19.4 (7)	20.1 (6)	24.4 (3)	29.1 **(1)**	24.1 (4)	24.1 (4)	28.4 (2)
SMTPProcessor	121.1 (7)	131.4 (6)	156.5 (3)	186.1 **(1)**	140.7 (5)	152.8 (4)	181.4 (2)
SMTPProtocol	22.1 (3)	16.6 (7)	17.0 (6)	17.4 (5)	22.2 (2)	22.1 (3)	22.6 **(1)**
Socket	53.0 (6)	48.5 (7)	57.2 (3)	61.2 **(1)**	55.1 (5)	55.4 (4)	58.5 (2)
StackAr	10.8 (5)	10.4 (7)	10.7 (6)	12.8 **(1)**	11.0 (3)	10.9 (4)	12.8 **(1)**
StringTokenizer	27.8 (4)	14.7 (7)	23.5 (6)	32.5 (2)	27.2 (5)	29.3 (3)	33.2 **(1)**
ToHTMLStream	22.7 (2)	6.7 (6)	6.5 (7)	15.8 (5)	21.2 (4)	22.2 (3)	23.3 **(1)**
ZipOutputStream	42.0 (2)	19.5 (7)	20.2 (6)	25.2 (5)	41.9 (3)	41.7 (4)	42.9 **(1)**
Average Rank	4.58 (5)	6.67 (7)	5.08 (6)	2.5 (2)	3.75 (4)	3.67 (3)	1.33 **(1)**

Table 7. Average Protocol Failures and Effect Sizes of EvoSuite versus Evo+EPAxp (RQ#2.1)

Subject	#pf EvoSuite	Evo+EPAxp			
		#pf	A_{12}	p-value	Improvement
JDBCResultSet	122.4	134.6	**0**	<0.0001	+9.97%
ListItr	20.6	28.1	**0**	<0.0001	+36.41%
NFST	9.3	10.8	**0.14**	<0.0001	+16.13%
SftpConnection	20.6	21.9	**0.29**	<0.0001	+6.31%
Signature	19.4	28.4	**0**	<0.0001	+46.39%
SMTPProcessor	121.1	181.4	**0**	<0.0001	+49.79%
SMTPProtocol	22.1	22.6	**0.37**	0.01	+2.26%
Socket	53.0	58.5	**0.03**	<0.0001	+10.38%
StackAr	10.8	12.8	**0.04**	<0.0001	+18.52%
StringTokenizer	27.8	33.2	**0.12**	<0.0001	+19.42%
ToHTMLStream	22.7	23.3	0.47	0.63	+2.64%
ZipOutputStream	42.0	42.9	0.44	0.27	+2.14%

Statistical significance (<0.05) is highlighted in bold.

Table 6 shows the protocol failure score (#pf) for various EvoSuite configurations. The table also shows (in parenthesis) how each configuration ranks when compared to others. For all subjects, the best ranked configuration was either Only-EPAxp or Evo+EPAxp. However, Evo+EPAxp ranked first for 8 of 12 subjects and for the remaining 4, it ranked second.

RQ#2.1: Evo+EPAxp *is the best performing* EvoSuite *configuration.*

Having identified again Evo+EPAxp as the best performing criterion we statistically analyse average number of detected protocol failures and effect sizes of EvoSuite versus Evo+EPAxp/ Table 7 shows that Evo+EPAxp outperforms with statistical significance EvoSuite in 10 of 12 subjects and achieves improvements of up to 50% when focusing exclusively in protocol failures.

Table 8. Average Protocol Failures and Effect Sizes of Best Enabledness-based Criterion from RQ#2.1 (Evo+EPAxp) versus Other Search-based Approaches for Protocol Failures (RQ#2.2)

Subject	Evo	Evo+SMC (2)				EvoMOSA (3)			
	EPAxp (1)	#pf	A_{21}	p-value	Improvement	#pf	A_{31}	p-value	Improvement
JDBCResultSet	134.6	95.7	**0**	<0.0001	+40.65%	97.5	**0**	<0.0001	+38.05%
ListItr	28.1	20.1	**0**	<0.0001	+39.80%	20.0	**0**	<0.0001	+40.50%
NFST	10.8	9.3	**0.11**	<0.0001	+16.13%	8.3	**0.05**	<0.0001	+30.12%
SftpConnection	21.9	19.7	**0.14**	<0.0001	+11.17%	15.8	**0.05**	<0.0001	+38.61%
Signature	28.4	19.6	**0**	<0.0001	+44.90%	20.7	**0**	<0.0001	+37.20%
SMTPProcessor	181.4	107.5	**0**	<0.0001	+68.74%	102.1	**0**	<0.0001	+77.67%
SMTPProtocol	22.6	20.8	**0.12**	<0.0001	+8.65%	17.4	**0**	<0.0001	+29.89%
Socket	58.5	53.5	**0.06**	<0.0001	+9.35%	52.5	**0.02**	<0.0001	+11.43%
StackAr	12.8	10.3	**0.02**	<0.0001	+24.27%	10.6	**0.03**	<0.0001	+20.75%
StringTokenizer	33.2	27.9	**0.1**	<0.0001	+19.00%	28.7	**0.08**	<0.0001	+15.68%
ToHTMLStream	23.3	14.4	**0.04**	<0.0001	+61.81%	11.3	**0.01**	<0.0001	+106.19%
ZipOutputStream	42.9	33.0	**0.02**	<0.0001	+30.00%	29.4	**0**	<0.0001	+45.92%

Statistical significance (<0.05) is highlighted in bold.

We observe that only for subjects `ToHTMLStream` and `ZipOutputStream` Evo+EPAxp failed to achieve an improvement with statistical significance. Perhaps not surprisingly, these two subjects were among the three subjects where Evo+EPAxp failed to achieve an improvement of *general* failure detection as discussed in **RQ#1.1**. Therefore, we can conjecture that the fact that EvoSuite achieved an already high amount of adjacent pairs coverage led to a very limited profit in terms of protocol failure detection.

RQ#2.1: *Criterion* Evo+EPAxp *is statistically better than* EvoSuite *for protocol failures and can achieve protocol failure detection improvement of up to 50%.*

RQ#2.2: How does the best enabledness-based criterion from **RQ#2.1** compare to other state-of-the-art search-based approaches for *protocol* failures?

Table 8 shows the average score for protocol failures (i.e., #pf) and effect sizes when executing the state of the art approaches Evo+SMC and EvoMOSA, compared to Evo+EPAxp.

Results show that for all subjects Evo+EPAxp outperformed Evo+SMC and EvoMOSA with statistical significance with improvements of up to 68% over Evo+SMC and up to 77% over EvoMOSA.

When contrasted to results observed in **RQ#1.2**, we can also conclude that Evo+EPAxp increased its effectiveness when restricted to protocol failures over both state-of-the-art approaches in search-based test generation.

RQ#2.2: *For protocol failures* Evo+EPAxp *outperforms* Evo+SMC *and* EvoMOSA *for all subjects and can provide significant improvement of protocol failure detection rate.*

RQ#2.3: How does the best enabledness-based criterion from **RQ#2.1** compare to a purely random approach in terms of *protocol* failure detection?

Table 9 shows the average mutation score for protocol failures (i.e., #pf) and effect sizes when executing Randoop compared to Evo+EPAxp

The table shows, with statistical significance, that Evo+EPAxp outperformed Randoop for 7 of 12 subjects, while underperforming agaisnt Randoop for the remaining 5. Note that for subjects

Table 9. Average Protocol Failures and Effect Sizes of Best Enabledness-based Criterion from RQ#2.1 (Evo+EPAxp) versus a Purely Random Approach for Protocol Failures (RQ#2.3)

Subject	Randoop	Evo+EPAxp			
		#pf	A_{12}	p-value	Improvement
JDBCResultSet	111.2	134.6	**0**	<0.0001	+21.04%
ListItr	1.0	28.1	**0**	<0.0001	+2710%
NFST	11.0	10.8	**0.59**	<0.0001	−1.82%
SftpConnection	0.0	21.9	**0**	<0.0001	—
Signature	7.0	28.4	**0**	<0.0001	+305.71%
SMTPProcessor	104.4	181.4	**0**	<0.0001	+73.75%
SMTPProtocol	25.1	22.6	**0.99**	<0.0001	−9.96%
Socket	29.0	58.5	**0**	<0.0001	+101.72%
StackAr	13.0	12.8	**0.57**	<0.0001	−1.54%
StringTokenizer	39.0	33.2	**1**	<0.0001	−14.87%
ToHTMLStream	25.0	23.3	**0.64**	<0.0001	−6.80%
ZipOutputStream	13.0	42.9	**0**	<0.0001	+230%

Statistical significance (<0.05) is highlighted in bold.

where Randoop does better, the average improvement is 7% and never greater than 15% (with effect sizes from 0.57 to 1). However, for subjects in which Evo+EPAxp beets Randoop, it achieves very significant improvements.

Although there could be multiple factors that explain why for some subjects Evo+EPAxp does not outperform Randoop (e.g., different mechanisms for generating string values), the enabledness-based criterion still offers significant improvements over a purely random approach for most of the studied subjects.

RQ#2.3: *For protocol failures* Evo+EPAxp *outperforms* Randoop *in most subjects and achieved very significant improvements in terms of protocol failure detection.*

RQ#3: Is there a negative impact in terms of structural coverage when including the best enabledness-based criterion from **RQ#1.1** and **RQ#2.1** to a search-based approach?

It could happen that a fitness criteria may worsen structural code coverage [77]. In Table 10 we show the number of subjects for which Evo+EPAxp performed better or worse in terms of structural coverage than EvoSuite with statistical significance (p-value < 0.05).

Results show that only for two subjects (`JDBCResult` and `ToHTMLStream`) Evo+EPAxp achieved worse structural coverage than EvoSuite with statistical significance. In both cases, the loss of structural coverage is less than 1 percentage point.

For the three studied structural criteria (i.e., line, branch, and exceptional coverage) overwhelmingly there is almost no observable noticeable difference between EvoSuite and Evo+EPAxp.

RQ#3: *The impact of* Evo+EPAxp *in terms of structural coverage when compared to* EvoSuite *is not negative in most subjects and at most negligible.*

6.1.5 Manual Effort. Our approach requires each subject to come with enabledness Boolean queries, one for each public method. We report on how we produced these Boolean queries to mitigate the possibility of these queries being incorrect. We also provide some anecdotal data to argue that the manual effort involved is not significant.

Table 10. Summary of Code Coverage Comparison: EvoSuite versus Evo+EPAxp (RQ#3)

Subject	EvoSuite			Evo+EPAxp								
	line	branch	excep	line	A_{12}	p-value	branch	A_{12}	p-value	excep	A_{12}	p-value
JDBCResultSet	88.9%	80.6%	27.0	88.8%	**0.55**	0.025	79.9%	**0.84**	<0.0001	27.0	0.5	0.99
ListItr	82.4%	86.1%	6.4	82.6%	0.48	0.163	86.4%	0.48	0.163	6.4	0.46	0.215
NFST	100.0%	94.3%	3.0	100.0%	0.5	1	94.4%	0.49	0.325	3.0	0.51	0.655
SftpConnection	93.5%	92.2%	19.4	94.5%	**0.07**	<0.0001	93.2%	**0.28**	<0.0001	18.5	**0.67**	0.001
Signature	90.2%	96.4%	19.0	95.1%	**0**	<0.0001	100.0%	**0**	<0.0001	19.0	0.5	1
SMTPProtocol	78.7%	67.2%	15.0	78.6%	0.52	0.312	67.0%	0.52	0.312	15.0	0.51	0.325
SMTPProtocol	77.8%	63.4%	14.0	77.9%	0.49	0.325	63.4%	0.49	0.325	14.0	0.5	1
Socket	86.8%	82.2%	17.2	86.7%	0.52	0.738	82.2%	0.5	1	17.2	0.5	1
StackAr	100.0%	100.0%	4.3	100.0%	0.5	1	100.0%	0.5	1	4.2	0.51	0.876
StringTokenizer	77.4%	55.9%	8.0	77.5%	0.48	0.159	55.9%	0.49	0.325	8.0	0.5	1
ToHTMLStream	81.0%	76.2%	11.6	80.8%	**0.61**	0.026	75.3%	**0.62**	0.017	10.7	**0.67**	0.001
ZipOutputStream	93.3%	89.9%	11.9	93.3%	0.5	1	90.0%	0.45	0.186	11.7	0.55	0.316
Better than EvoSuite				2/12			2/12			0/12		
Worse than EvoSuite				2/12			2/12			2/12		

We had authors *A* and *B* separately perform the task of writing their own enabledness queries for all the subjects previously used in the literature. The authors did not use the existing EPA models from previous works [28, 54]. Author *A* was very familiar with the case studies while author *B* was not. However, *B* is well experienced in program analysis and source code walkthroughs. The total effort for this task was 180 minutes for *A* and 150 minutes for *B*. Averaging the effort of *A* and *B* over the total number of methods (114) results in 1.44 minutes per method per person. The most laborious class was `StringTokenizer` for which the average required *permethod* was 2.78 minutes.

Once completed, both authors spent 50 minutes comparing queries and agreeing on corrections when discrepancies were found. Of the 114 methods, 36 were initially flagged as having discrepancies of which 5 were simple typos (e.g., missing a negation). Thus, there were 31 differences (27%) because of incorrect understanding of the code. Of these 20 were all the methods of `SftpConnection` due to one same misinterpretation of the code. For all methods but one, at least one reviewer wrote, to the best of our knowledge, the correct query.

An important note is that for 98 methods (85%), the Boolean query corresponds to code that appears explicitly as part of a conditional statement in the first few lines of the method. In other words, the information for determining enabledness was part of the defensive code that appears at the start of methods. The queries for 15 methods were also code found in conditional statements but dispersed throughout the method code.

Thus, for over 99% of the methods, the reviewers only needed to pinpoint a subset of the actual SUT code to capture the enabledness queries. The subset corresponds to conditional statements that guard statements that throw exceptions. An example of such cases can be seen in Figure 6.

Only one method required a Boolean query with conditions that did not appear in the method code. This was the case for an unguarded access to positions in a String in `nextIsSet()` of class `StringTokenizer`.

6.2 Evaluation on Real Object Protocol Implementation Defects

In Section 6.1, we report on the effectiveness for failure detection using synthetic defects on selected third-party classes that implement object protocols. In this section, we study effectiveness on real defects. Note that we do not make a distinction between general and protocol failures as in

```java
public InputStream getInputStream() throws IOException {
  if (isClosed())
    throw new SocketException("Socket is closed");
  if (!isConnected())
    throw new SocketException("Socket is not connected");
  if (isInputShutdown())
    throw new SocketException("Socket input is shutdown");
  final Socket s = this;
  InputStream is = null;
  try {
    is = AccessController.doPrivileged(
      new PrivilegedExceptionAction<InputStream>() {
        public InputStream run() throws IOException {
          return impl.getInputStream();
        }
      });
  } catch (java.security.PrivilegedActionException e) {
    throw (IOException) e.getException();
  }
  return is;
}

@EpaActionPrecondition(name = "getInputStream")
private boolean isGetInputStreamEnabled() {
  return !isClosed() && isConnected() && !isInputShutdown();
}
```

Fig. 6. Socket method example where Boolean query corresponds to code that appears explicitly as part of a conditional statement in the first few lines of the method.

Section 6.1 due to the limited number of real defects we study: In Section 6.1, we studied hundreds of synthetic defects while here we focus on 25 real defects.

6.2.1 Subjects. As mentioned before, there are no datasets of classes with real defects that also implement object protocols. We approach this lacking in two ways. On one hand, we use a well known dataset of real defects although it is not geared toward classes implementing object protocols. On the other hand, we use real defects we found for classes that are known to implement object protocols.

We consider the Defects4J[3] dataset, which contains 438 real defects from six Java open source projects (Char, Lang, Math, Mockito and Closure) [51]. Defects4J offers a dataset in which each fault is represented by a buggy and fixed version. Given the manual effort required to apply the test generation approach described in this article, we selected a subset of these subjects.

As the target classes for the test generation techniques under evaluation are those that implement object protocols we used the following procedure to select subjects from the Defects4J dataset. For each project, we looked at the five classes with most reported defects. We then manually inspected each class to remove classes with trivial preconditions (i.e., $True$) in all their public methods. We were left with one class per project except for the Mockito project that consisted of five classes containing only static methods. We analysed all buggy versions for each of the remaining classes. Table 11(a) provides an overview of the Defects4J subjects. Note that each subject has an ID representing the Defects4J defect number.

As the Defects4J benchmark was not constructed with object protocols in mind, we also created a benchmark with subjects found in the wild that both had real defects and also non-trivial object protocols. The criteria we used for the latter is that classes should be alternative implementations of those studied in Section 6.1; the rationale being that others (i.e., References [24, 28, 33, 54, 56, 74]) considered that these classes exhibit interesting object protocols.

[3]Version 1.5.0.

Table 11. Real Bugs Subject Summary

(a) Defects4J subjects summary.

Project	Subject	LOC	|B|	|M|
Chart	TimeSeries_3	439	186	56
	TimeSeries_9	330	130	46
	TimeSeries_17	338	128	46
Math	SimplexTableau_33	225	124	36
	SimplexTableau_42	231	128	36
	SimplexTableau_83	196	96	37
	SimplexTableau_87	198	92	40
	SimplexTableau_88	197	90	40
Lang	StrBuilder_47	870	472	132
	StrBuilder_59	804	440	110
	StrBuilder_60	804	440	110
	StrBuilder_61	804	440	110
Closure	Compiler_18	1107	354	158
	Compiler_31	1081	338	153
	Compiler_59	903	270	134
	Compiler_64	882	262	131
	Compiler_140	742	192	114
	Compiler_160	882	260	131

(b) Subjects in the wild summary.

Subject	LOC	|B|	|M|
ListIterator [4]	246	24	12
HashIterator [5]	44	22	06
FilterListIterator [6]	107	32	21
SelectionListIterator [7]	34	08	11
StringTokenizer_tokens [8]	86	42	12
StringTokenizer_delimiter [9]	86	42	12
FilterIterator [10]	51	12	1

LOC is lines of code, $|B|$ is number of branches, $|M|$ is number of methods.

To find real bugs for the subjects of Section 6.1, we searched in three bug tracking systems [11–13]. We only considered issues tagged as bugs for which both the buggy and fixed source code versions were available. As a result we found seven real bugs covering the ListIterator and StringTokenizer subjects. Table 11(b) provides an overview of the subjects. `StringTokenizer`, `HashIterator`, `SelectionListIterator`, `StringTokenizer_tokens`, and `StringTokenizer_delimiter` were taken from the Java Development Kit implementation. `FilterListIterator` and `FilterIterator` were taken from the apache commons collections implementation. It is worth noting that `StringTokenizer_tokens` and `StringTokenizer_delimiter` are the same `StringTokenizer` class but we add a suffix for each defect.

For all subjects with real defects (i.e., Defects4J and the new benchmark) it was not necessary to write mocks, because the classes do not contain network or database operations.

6.2.2 Research Questions. Although conceptually we only have one research question for this Section, we split it into two (one for each dataset). The very different nature of the analysed benchmarks (see Section 6.2.1) makes it of interest to see the conclusions separate.

- **RQ#4.1:** How does the best enabledness-based criterion from **RQ#1.1** compare to a vanilla search-based approach in terms of *general* failures detection for real defects from Defects4J dataset?
- **RQ#4.2:** How does the best enabledness-based criterion from **RQ#1.1** compare to a vanilla search-based approach in terms of *general* failures detection for bugs in the wild?

6.2.3 Experimental Design.

Tool Comparison. For both **RQ#4.1**, **RQ#4.2** we compare Evo+EPAxp, the best performing tool in the previous section, to the standard configuration of EvoSuite. Observe that we explicitly discarded Randoop from this comparison, since we chose to focus on the effects on the technique, independently of different implemenation details that arose when comparing different tools.

Measuring test suite effectiveness. Each subject is represented by a buggy and a fixed version. The test suite is generated always using the fixed version and executed over the buggy version. If the test suite fails on the buggy version, then the test suite is capable of detecting the bug. In practice, however, test may fail on the fixed version too. To remove flaky tests, we followed the procedure described in Shamshiri et al. [83], iteratively re-executing all tests over the fixed version and removing failing ones. To avoid false positives, we manually validated the exceptions generated by a test and checked that the exception message or assertions generated are equivalent to those included in Defects4J. Flaky tests were identified in the tests for **RQ#4.1** but not for **RQ#4.2**. No false positives were found. Finally, no statistical analysis is performed as the number of real defects analysed is insufficient to do so.

Parameter Settings and Statistical Tests. As before, we kept EvoSuite's default values for all parameters such as population size, type of selection mechanism, type of crossover, archive, and so on. For both generation algorithms, we executed the configuration with a budget of 10 minutes on each subject. We followed the guidelines described in Arcuri et al. [18] to statistically compare non-deterministic test generators. We repeated each execution of every generation algorithm 60 times with different random seeds. In total, this lead to 25 defects $\times$ 2 algorithms $\times$ 60 repetitions $\times$ 10 minutes = 500 processing hours ($\approx$21 days).

6.2.4 Experimental Results. We now discuss results for each research question.

RQ#4.1: How does the best enabledness-based criterion from **RQ#1.1** compare to a vanilla search-based approach in terms of *general* failures detection for real defects from Defects4J dataset?

Table 12 shows the summary of results for each subject in our Defects4J dataset when executing EvoSuite compared to Evo+EPAxp.

Results show that EvoSuite is more likely than Evo+EPAxp to detect defects only in three subjects: SimplexTableau_42, StrBuilder_47 and StrBuilder_60 with improvements of 5, 1, and 2 defects found respectively. However, the likelihood of Evo+EPAxp detecting defects is greater than that of the vanilla approach in seven subjects with improvements from 1 to 24 more defects found. For one subject, EvoSuite and Evo+EPAxp always found the defect (60/60). For the remaining seven subjects neither EvoSuite nor Evo+EPAxp found the defect.

It is worth noting that the Closure project, and the Compiler class in particular, are challenging for automatically generating tests. As Shamshiri et al. [83] discuss, most of the classes in the Closure project require the creation of complex objects. This requires not only a particular sequence of method calls prior to exercising the defect, but also the creation of complex strings (e.g., a javascript program). Evo+EPAxp may outperform EvoSuite for generating the sequence of calls, but is as weak as EvoSuite for generating the complex input strings. As a result, there is no significant difference in the results between EvoSuite and Evo+EPAxp.

RQ#4.1: Evo+EPAxp *is more likely to detect defects than* EvoSuite *in 7 subjects, while the opposite happens in 3 subjects.*

RQ#4.2: How does the best enabledness-based criterion from **RQ#1.1** compare to a vanilla search-based approach in terms of *general* failures detection for bugs in the wild?

Table 12. Summary of Bug Finding Results for Each Subject for Defects4J Dataset: EvoSuite versus Evo+EPAxp

Bug ID	EvoSuite	μ EvoSuite	Evo+EPAxp	μ Evo+EPAxp
TimeSeries (Chart Project)				
3	0/60	0.00%	2/60	3.33%
9	0/60	0.00%	2/60	3.33%
17	60/60	100.0%	60/60	100.0%
SimplexTableau (Math Project)				
33	0/60	0.00%	0/60	0.00%
42	25/60	41.66%	20/60	33.33%
83	1/60	1.66%	8/60	13.33%
87	38/60	63.33%	44/60	73.33%
88	0/60	0.00%	0/60	0.00%
StrBuilder (Lang Project)				
47	59/60	98.33%	58/60	96.66%
59	18/60	30.00%	42/60	70.00%
60	2/60	3.33%	0/60	0.00%
61	3/60	5.00%	15/60	25.00%
Compiler (Closure Project)				
18	6/60	10.00%	7/60	11.66%
31	0/60	0.00	0/60	0.00%
59	0/60	0.00%	0/60	0.00%
64	0/60	0.00%	0/60	0.00%
140	0/60	0.00%	0/60	0.00%
160	0/60	0.00%	0/60	0.00%

μ column shows the average defect detection for each criterion (RQ#4.1).

Table 13. Summary of Bug Finding Results for Each Subject for Bugs in the Wild Dataset: EvoSuite versus Evo+EPAxp

Subject	EvoSuite	μ EvoSuite	Evo+EPAxp	μ Evo+EPAxp
ListIterator	0/60	0.00%	0/60	0.00%
HashIterator	0/60	0.00%	0/60	0.00%
FilterListIterator	60/60	100.0%	60/60	100.0%
SelectionListIterator	60/60	100.0%	60/60	100.0%
StringTokenizer_tokens	10/60	16.66%	15/60	25.00%
StringTokenizer_delimiter	11/60	18.33%	11/60	18.33%
FilterIterator	1/60	1.66%	58/60	96.66%

μ column shows the average defect detection for each criterion (RQ#4.2).

Table 13 shows an overview comparing EvoSuite to Evo+EPAxp for our second dataset. Results show that Evo+EPAxp is more likely to detect defects than EvoSuite in two subjects: StringTokenizer_token and FilterIterator with improvements of 5 (50%) and 57 (570%) more bugs found, respectively. However, Evo+EPAxp never underperforms EvoSuite.

Note that the defects for subjects ListIterator and HashIterator were not found be either tool. The ListIterator defect requires calling the ListIterator constructor with a List object as a parameter, then the list object must be modified directly (i.e., without calling the ListIterator methods) and finally the ListIterator remove() method must be called. This is a *multi-object protocol* sequence of calls that is not rewarded by the fitness criterion used in Evo+EPAxp, thus unlikely to be covered. This is a direction that is worth exploring in the future.

The HashIterator defect requires calling the next() method repeatedly until the end of the collection is reached and a NoSuchElementException is thrown. A subsequent call to remove() should remove the last element returned by next(). However, the HashIterator implementation instead throws an IllegalStateException. This behavior is inconsistent with other Iterator implementations, such as those returned by ArrayList and LinkedList. So, to detect the defect it is necessary to catch the first exception and then do a subsequent call to remove(). However, automatically generated test cases stop test case generation immediately after an exception is thrown as there is no oracle that asserts what a consistent state after an exception would be. Interestingly, the inferred EPA while testing the defective class differs from the intended EPA protocol exhibiting the failure introduced by the defect. This further suggests that comparing the expected EPA protocol could guard against un-intended regressions. We elaborate about this as future work in the corresponding section.

RQ#4.2: Evo+EPAxp *is more likely to detect defects than* EvoSuite *in 2 subjects, and is never outperformed by* EvoSuite.

6.2.5 Manual Effort. As mentioned previously, our approach requires each subject to come with enabledness Boolean queries, one for each public method. We provide some new anecdotal data to argue that the manual effort involved is not significant.

For subjects with real defects, the total effort was 175 minutes for all methods (293), which results in 0.99 minute per method in average. The most laborious class was again `StringTokenizer` for which the average required *permethod* was 1.67 minutes. Note the reduction in time for this class compared to the first effort to build its Boolean queries (2.78 minutes per method, see Section 6.1.5). This provides some anecdotal evidence that the effort in maintaining Boolean queries across versions may decrease as experience with the class is accumulated. On a similar note, multiple defects for the same subject (i.e., in the Defects4J benchmark) do not require a significant effort to update the Boolean queries as changes to public methods is in general small.

7 THREATS TO VALIDITY

For RQ1 to RQ3, the main threat to construct validity is the use of mutation as a proxy of defects. Although this is still a matter of discussion there is evidence suggesting that test suites that are good at finding seeded failures are also good at finding real failures [15, 52]. Nonetheless, mutation analysis is a prevalent experimental methodology in the testing community. For this preliminary study we focused on object protocols previously studied in the literature. Unfortunately there is no corpus of real defects for these kinds of subjects. To mitigate these threat, we include RQ4 that studies real defects.

Multiple comparisons on the same data set inflates the probability of Type I error. Instead of applying an adjustment such as Bonferroni [68, 73], we followed the guidelines given by Arcuri and Briand [17]. More specifically: we report the obtained p-values, not just whether a difference is significant or not at an arbitrarily chosen α level. By doing this, if for some reason a reader wants to evaluate the results using a Bonferroni adjustment or any of its (less conservative) variants, then it is possible to do so.

A major internal threat is the possibility of the tools we have used or implemented being defective. We chose EvoSuite that is a well-known open source tool available online. Our prototype is also available online, as well as the subjects and scripts needed to replicate the results. We used

PITEST for mutation analysis, a well maintained and updated mutation analysis tool widely used by the research community.

A major threat for RQ4 is the subject selection criteria used. The subjects in Defects4J are unlikely to be representative of classes that implement object protocol. Indeed, they do not appear in any of the literature on the topic (including the empirical study [24]). The other real defects for object protocol were taken from public repositories and may not be the most representative of the defects that are common for these classes. For RQ4, the manual task of writing the enabledness Boolean queries was performed by only one author, because for RQ1–3, we only detected marginal differences between the two author's queries. However, these may contain errors.

The use of randomized algorithms in the test generation poses a threat. To mitigate this threat we repeated EvoSuite execution 60 times with different random seeds as suggested by Arcuri and Briand [17].

Another threat is the budget time we used for the tool comparison. We set the same budget time (10 min) for all criteria. However, the techniques presented herein require extra manual effort to add preconditions. We consider it is not necessary to increase the budget time for other criteria to compensate for this, because code coverage in EvoSuite and Randoop usually saturate already within first minutes and then test suites exhibit a very high degree of redundancy [83]. Also, the pre-conditions only are required to be produced once, and can be reused (modulo minor changes) in every regression testing effort.

In terms of external threats, the main point is that we analysed a reduced set of classes with object protocols. Thus, our conclusions on the effectiveness of the coverage criteria and the tool may not generalise to other subjects. This article is aimed at testing software units exhibiting object protocols. The ideas we present may well not be applicable to system-level test generation. We did, however, use subjects collected from different works on automatic mining of EPA models [28, 54] that are considered relevant by the software engineering community in typestate verification [24, 25, 33, 37, 38, 56, 74]. Although subjects might seem rather simplistic due to the limited size of the source code, the corresponding object protocol is still challenging for developers [33]. Scalability of the presented approach is not affected by lines of code directly, there is no code analysis involved during test case generation or xEPA refinement. Execution time of the code under analysis may impact efficiency. If the execution cost is significant, then the evolutionary algorithm will iterate less times within a fixed budget. This may lead to decreased coverage and consequently less failure detection ability. Although the size of the xEPA can grow in the worst case to $2^{|M|}$ where M is the set of public methods, the number of reachable EPA states is usually small in practice. As an example, the largest EPA model we have encountered is the Microsoft WINS Replication and Autodiscovery Protocol that with 33 methods results in an EPA with 38 states and 233 transitions [32]. Our findings may not generalise to non-deterministic API implementations as the notion of test effectiveness will be affected by test flakiness. Nevertheless, this problem could be mitigated by automatic controlling sources of non-determinism [19, 21].

8 RELATED WORK

Automated test generation is an active research area [69]. Approaches most relevant to this article are those that involve generating tests that are call sequences (e.g., References [16, 59, 70, 88, 89]). A noteworthy tool for Java and .NET is Randoop [70], a purely random (and black box) approach that generates method sequences by reusing previously returned values. Randoop produces large numbers of different tests but does not use a coverage criteria to drive test generation. The sheer number of tests it generates makes mutation score analysis challenging. Studies [40, 84] show that genetic algorithm approaches such as those implemented in EvoSuite can outperform random testing.

The work presented herein has significant similarities with the work presented by Liu et al. [59]. Both quotient the state space of an object by calling Boolean observers. However they use parameterless Boolean observers (e.g., $empty()$, $full()$) instead of method preconditions. An abstract state represents all concrete objects states that are undistinguishable by only calling parameterless Boolean observers. Tests are generated aiming at covering every abstract state instead of aiming to cover transitions between abstract states. Finally, the test generation process is based on the use of a Boolean constraint solver, a theorem prover and forward random search. The use of automatically generated abstraction of the class resembles EPAs, however there is no abstraction of the protocol. Indeed, transitions between abstract states are not considered. Although promising results in terms of real defect finding is shown, an empirical analysis on level ground (i.e., equal time budget) with respect to other techniques is not reported, neither report statistical significance of their results.

Marchetto et al. [61] also dynamically generates a state-based abstraction. More specifically, aimed at Ajax Web Applications. This abstraction is iterativelly refined from observed executions, as our approach does. However, this approach does not abstract using enabledness actions, but certain properties on the DOM web instance.

Dallmeier et al. [29] use a typestate model wich is dynamically inferred and used to guide the generation of new test cases that try to increase transition coverage of the type state. Typestates are abstractions of object protocols, similar to EPAs in spirit, but that quotient object states based on their future acceptable behaviour rather than enabledness of the next action. The main difference with our work is that the intent is to detect misuses of the object protocol by client programs, while our interest is in defects in the class that provides the protocol.

The Seeker tool [88] combines dynamic and static code analyses to build an increase branch and def-use coverage. Defect detection is not studied. The tool targets .NET programs hence a direct comparison is not possible. However, there are results [55, 88] that indirectly show that it is capable of achieving comparable results to evolutionary approaches in terms of branch coverage.

Work on search based unit test generation for Java classes includes Reference [89] for increasing branch coverage, and Reference [16] where genetic algorithms are used to search for parameters that can increase code coverage of a random test case generator.

In summary, the techniques mentioned above vary from systematic white-box to search-based approaches aiming at structural coverage and defect detection. None attempt to exploit covering automatically generated protocol abstractions for test case generation.

A highly related area is that of system level testing at the graphical user interface (GUI) level. Here the GUI enforces an interaction protocol between the user and the system. An important difference is that the GUI can strictly enforce interactions by simply disabling interface elements while an object method can always be called on and must raise exceptions when called upon incorrectly. Various approaches that follow a similar strategy to ours exists (e.g., References [14, 61, 65, 66]) in which abstractions of the graphical user interface are built and used to define coverage criteria that can aide the construction of more effective tests. All use significantly finer grained abstractions than the enabledness criteria used in this article. For instance, GUI objects are considered equivalent if they have the same property values [14].

There has also been work on defining coverage criteria and generating tests from formal specifications and models (e.g., References [45, 48, 91]). Compared to the our work, these approaches require significant human involvement in producing an appropriate abstraction written in a different language than the implementation, and which is the key input to the test generation technique. In contrast, we only require methods that return true if it possible to call a specific method and have it return normally in the current object state. The precondition for a method typically is a subset of the lines of code of the method.

There are experiments [75] that analyse the effectiveness of test case generation from models against manual construction from specifications and random generation. Model coverage, implementation coverage and code and requirements defect detection are studied. Models used for test case generation include extended finite state machines. Effectiveness of model coverage against other coverage criteria is not studied.

Santiago et al. [81] present a tool for test case generation from statecharts. As in our paper, they also study an all-pairs criterion. However, effectiveness is not studied in comparison with other criteria. Jan and Tretmans [50] proposed a behaviour model test generation technique based on the notion of IOCO [94] that is used to verify for conformance.

One problem identified with test case generation from human constructed models or specifications is that accidental syntactic aspects of the description can influence the effectiveness of the coverage criteria used (e.g., References [28, 47, 67, 76, 79, 80, 95]). To address this, Czemerinski et al. [28] studied coverage criteria based on EPAs. EPAs abstract away from syntactic aspects of code or specification by quotienting the state space of the object protocol. Thus, two semantically equivalent (yet syntactically different) descriptions or implementations of an object protocol yield the same EPA. Czemerinski et al. [28] showed a correlation between transition coverage in the EPA of the intended call protocol and the ability of the test suites to detect protocol conformance failures. In this article, not only do we shift the focus to test case generation and provide an additional (and more effective) coverage criterion (Evo+EPAxp), we infer EPAs dynamically rather than requiring them upfront and also do not restrict our interest exclusively to protocol defects, rather we are also interested in understanding how to exploit EPAs to detect all types of defects.

Other relevant work in this line is that of automatic under approximation of infinite behaviour from concrete [62] or symbolic [44] executions that can be later used for regression testing. Finitisation [45] is also addressed using unfolding domain bounding, slicing and state pruning. However, no statistical studies on coverage and its effectiveness are available [42].

Related work on EPA includes automated static EPA construction from source code [33] and [31, 32] from contract specifications.

There has been significant effort in studying specification, inference and analysis of object protocols (e.g., References [32, 35, 36, 45]). An area of particular interest, both in academia (e.g., References [35, 81]) and industry (e.g., Reference [23]), is that of protocol conformance in which the goal is to check if an implementation correctly implements a given protocol, in other words verifying that the code accepts or rejects sequences of method calls correctly according to the intended protocol.

9 CONCLUSIONS AND FURTHER WORK

In this article we study test case generation techniques that build and exploit an abstraction of object protocols to find failures. We study various coverage criteria over enabledness EPA and also xEPA. We define search-based test case generation techniques that attempt to achieve high abstraction coverage. We implemented a prototype extending the EvoSuite search-based test generator that is available at https://github.com/j-godoy/epa-evosuite.

Experimental results suggest that the xEPA coverage criteria and fitness functions related to covering pairs of transitions can provide increased failure detection capabilities (for both protocol and general failures) when compared to search-based test generation for standard structural coverage, strong mutation coverage and multi-objective search-based algorithms. The approach also outperforms a random approach (Randoop) for most subjects.

Currently, the techniques require manual intervention to produce an enabledness Boolean queries for each public class method. We believe this manual step could be revealed by using automated method preconditions inference.

Our work focuses on classes that implement object protocols. Experimentation is centred on single-object protocols but the test case generation approach is applicable to multi-object protocols too. Although classes with object protocols represent a relevant portion of classes in open source software they are far from being a majority. In this work we show that for xEPAs, significant improvement can be achieved. We believe a relevant line of future work is to identify abstractions of the state space of classes that do not have object protocols that can be exploited to improve test case generation. Another line for future work is the reduction of the manual identification of Boolean queries via automation. Additonally, we would like to investigate how sensible are the inferred EPA abstractions generated by the test suites to detect subtle regressions and if the inferred EPA can be used as an oracle.

REFERENCES

[1] [n.d.]. Oracle. Retrieved from https://docs.oracle.com/javase/8/docs/api/java/util/ListIterator.html and https://docs.oracle.com/javase/8/docs/api/java/util/ListIterator.html.

[2] [n.d.]. Tcases: A Model-Based Test Case Generator. Retrieved from https://github.com/Cornutum/tcases and https://github.com/Cornutum/tcasesl.

[3] [n.d.]. Apache. Retrieved from https://xalan.apache.org and https://xalan.apache.org.

[4] [n.d.]. Retrieved from https://bugs.openjdk.java.net/browse/JDK-4308549 and https://bugs.openjdk.java.net/browse/JDK-4308549.

[5] [n.d.]. Retrieved from http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6529795 and http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6529795.

[6] [n.d.]. Apache. Retrieved from https://issues.apache.org/jira/browse/COLLECTIONS-360 and https://issues.apache.org/jira/browse/COLLECTIONS-360.

[7] [n.d.]. Retrieved from http://hg.openjdk.java.net/openjfx/10-dev/rt/rev/572a70fabb47 and http://hg.openjdk.java.net/openjfx/10-dev/rt/rev/572a70fabb47.

[8] [n.d.]. Retrieved from https://bugs.openjdk.java.net/browse/JDK-4238266 and https://bugs.openjdk.java.net/browse/JDK-4238266.

[9] [n.d.]. Retrieved from https://bugs.java.com/bugdatabase/view_bug.do?bug_id=4135670 and https://bugs.java.com/bugdatabase/view_bug.do?bug_id=4135670.

[10] [n.d.]. Retrieved from https://issues.apache.org/jira/browse/COLLECTIONS-61 and https://issues.apache.org/jira/browse/COLLECTIONS-61.

[11] [n.d.]. Retrieved from https://bugs.openjdk.java.net/ and https://bugs.openjdk.java.net/.

[12] [n.d.]. Retrieved from http://bugs.sun.com and http://bugs.sun.com.

[13] [n.d.]. Retrieved from https://issues.apache.org and https://issues.apache.org.

[14] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. 2015. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Softw.* 32, 5 (September 2015), 53–59. DOI: https://doi.org/10.1109/MS.2014.55

[15] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng.* 32, 8 (2006), 608–624. DOI: https://doi.org/10.1109/TSE.2006.83

[16] James H. Andrews, Tim Menzies, and Felix C. H. Li. 2011. Genetic algorithms for randomized unit testing. *IEEE Trans. Softw. Eng.* 37, 1 (2011), 80–94.

[17] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. IEEE, 1–10.

[18] Andrea Arcuri and Lionel C. Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test. Verif. Reliabil.* 24, 3 (2014), 219–250. DOI: https://doi.org/10.1002/stvr.1486

[19] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated unit test generation for classes with environment dependencies. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 79–90. DOI: https://doi.org/10.1145/2642937.2642986

[20] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2015. Generating TCP/UDP network data for automated unit test generation. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 155–165. DOI: https://doi.org/10.1145/2786805.2786828

[21] Andrea Arcuri, Gordon Fraser, and René Just. 2017. Private API access and functional mocking in automated unit test generation. In *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST'17)*. IEEE Computer Society, 126–137. DOI: https://doi.org/10.1109/ICST.2017.19

[22] Cyrille Valentin Artho, Armin Biere, Masami Hagiya, Eric Platon, Martina Seidl, Yoshinori Tanabe, and Mitsuharu Yamamoto. 2013. Modbat: A model-based API tester for event-driven systems. In *Proceedings of the Haifa Verification Conference*. Springer, 112–128.

[23] Thomas Ball and Sriram K. Rajamani. 2001. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*. Springer-Verlag, Berlin, 103–122.

[24] N. Beckman, D. Kim, and J. Aldrich. [n.d.]. An empirical study of object protocols in the wild. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'11)*. 2–26.

[25] Kevin Bierhoff and Jonathan Aldrich. 2007. Modular typestate checking of aliased objects. *SIGPLAN Not.* 42, 10 (Oct. 2007), 301–320. DOI: https://doi.org/10.1145/1297105.1297050

[26] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Inf. Softw. Technol.* 104 (2018), 207–235. DOI: https://doi.org/10.1016/j.infsof.2018.08.010

[27] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. Pit: A practical mutation testing tool for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 449–452.

[28] H. Czemerinski, V. Braberman, and S. Uchitel. 2016. Behaviour abstraction adequacy criteria for API call protocol testing. *Softw. Test. Verif. Reliabil.* 26, 3 (2016), 211–244. DOI: https://doi.org/10.1002/stvr.1593

[29] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Gordon Fraser, Sebastian Hack, and Andreas Zeller. 2012. Automatically generating test cases for specification mining. *IEEE Trans. Softw. Eng.* 38, 2 (2012), 243–257. DOI: https://doi.org/10.1109/TSE.2011.105

[30] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. [n.d.]. Program abstractions for behaviour validation. In *Proceedings of the International Conference on Software Engineering (ICSE'11)*. 381–390.

[31] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. [n.d.]. Validation of contracts using enabledness preserving finite state abstractions. In *Proceedings of the International Conference on Software Engineering (ICSE'09)*. 452–462.

[32] Guido de Caso, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. 2012. Automated abstractions for contract validation. *IEEE Trans. Softw. Eng.* 38, 1 (2012), 141–162.

[33] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. 2013. Enabledness-based program abstractions for behavior validation. *ACM Trans. Softw. Eng. Methodol.* 22, 3 (2013).

[34] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.* 6, 2 (2002), 182–197. DOI: https://doi.org/10.1109/4235.996017

[35] Robert DeLine and Manuel Fähndrich. 2004. Typestates for objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'04)*. 465–490.

[36] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. 2006. Session types for object-oriented languages. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer, 328–352.

[37] John Field, Deepak Goyal, G. Ramalingam, and Eran Yahav. 2003. Typestate verification: Abstraction techniques and complexity results. In *Proceedings of the International Static Analysis Symposium*. Springer, 439–462.

[38] Stephen J Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2 (2008), 9.

[39] G. Fraser and A. Arcuri. 2013. Whole test suite generation. *IEEE Trans. Softw. Eng.* 39, 2 (2013), 276–291.

[40] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2, Article 8 (December 2014), 42 pages. DOI: https://doi.org/10.1145/2685612

[41] Gordon Fraser and Andrea Arcuri. 2015. Achieving scalable mutation-based generation of whole test suites. *Emp. Softw. Eng.* 20, 3 (2015), 783–812. DOI: https://doi.org/10.1007/s10664-013-9299-z

[42] Gordon Fraser and Andreas Zeller. 2011. Exploiting common object usage in test case generation. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST'11)*. IEEE Computer Society, 80–89. DOI: https://doi.org/10.1109/ICST.2011.53

[43] G. Fraser and A. Zeller. 2012. Mutation-driven generation of unit tests and oracles. *Trans. Softw. Eng.* 28, 2 (2012), 278–292.

[44] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. 2002. Generating finite state machines from abstract state machines. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02)*. 112–122.

[45] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman. 2011. Model-based quality assurance of protocol documentation: Tools and methodology. *STVR* 21, 1 (2011), 55–71. DOI: https://doi.org/10.1002/stvr.427

[46] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. [n.d.]. Swarm testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'12)*. 78–88.

[47] Mats P. E. Heimdahl, Devaraj George, and Robert Weber. 2004. Specification test coverage adequacy criteria = Specification test generation inadequacy criteria. In *Proceedings of the Eighth IEEE International Conference on High Assurance Systems Engineering (HASE'04)*. IEEE Computer Society, USA, 178–186.

[48] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. 2009. Using formal specifications to support testing. *Comput. Surveys* 41, 2, Article 9 (2009), 9:1–9:76 pages.

[49] Kobi Inkumsah and Tao Xie. 2008. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 297–306.

[50] Jan and Tretmans. 1996. Conformance testing with labelled transition systems: Implementation relations and test generation. *Comput. Netw. ISDN Syst.* 29, 1 (1996), 49–79.

[51] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis, (ISSTA'14)*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. DOI: https://doi.org/10.1145/2610384.2628055

[52] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 654–665.

[53] Teemu Kanstrén and Olli-Pekka Puolitaival. 2012. Using built-in domain-specific modeling support to guide model-based test generation. *Model-Driven Engineering of Information Systems: Principles, Techniques, and Practice* (2012), 295–319.

[54] I. Krka, Y. Brun, and N. Medvidovic. [n.d.]. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE'14)*. 178–189.

[55] Kiran Lakhotia, Phil McMinn, and Mark Harman. 2010. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *J. Syst. Softw.* 83, 12 (December 2010), 2379–2391. DOI: https://doi.org/10.1016/j.jss.2010.07.026

[56] Tien-Duy B. Le, Xuan-Bach D. Le, David Lo, and Ivan Beschastnikh. 2015. Synergizing specification miners through model fissions and fusions (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 115–125. DOI: https://doi.org/10.1109/ASE.2015.83

[57] David Lee and Mihalis Yannakakis. 1996. Principles and methods of testing finite state machines-a survey. *Proc. IEEE* 84, 8 (1996), 1090–1123.

[58] Wenbin Li, Franck Le Gall, and Naum Spaseski. 2017. A survey on model-based testing tools for test case generation. In *Proceedings of the International Conference on Tools and Methods for Program Analysis*. Springer, 77–89.

[59] Lisa Liu, Bertrand Meyer, and Bernd Schoeller. 2007. Using contracts and Boolean queries to improve the quality of automatic test generation. In *Proceedings of the 1st International Conference on Tests and Proofs (TAP'07)*. Springer-Verlag, Berlin, Heidelberg, 114–130.

[60] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. 2014. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Trans. Softw. Eng.* 40, 1 (2014), 23–42. DOI: https://doi.org/10.1109/TSE.2013.44

[61] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. 2008. State-based testing of Ajax web applications. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST'08)*. 121–130. DOI: https://doi.org/10.1109/ICST.2008.22

[62] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. 2011. AutoBlackTest: A tool for automatic black-box testing. In *Proceedings of the International Conference on Software Engineering (ICSE'11)*. 1013–1015.

[63] Raluca Marinescu, Cristina Seceleanu, Hèléne Le Guen, and Paul Pettersson. 2015. A research overview of tool supported model-based testing of requirements-based designs. *Adv. Comput.* 98 (2015), 89–140.

[64] P. McMinn. 2004. Search-based software test data generation: A survey. *Softw. Test., Verif. Reliabil.* 14, 2 (2004), 105–156.

[65] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. 2001. Coverage criteria for GUI testing. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-9)*. ACM, New York, NY, 256–267. DOI: https://doi.org/10.1145/503209.503244

[66] A. Mesbah, E. Bozdag, and A. v. Deursen. 2008. Crawling AJAX by inferring user interface state changes. In *Proceedings of the 2008 8th International Conference on Web Engineering*. 122–134. DOI: https://doi.org/10.1109/ICWE.2008.24

[67] S. Mouchawrab, L. C. Briand, Y. Labiche, and M. Di Penta. 2011. Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments. *IEEE Trans. Softw. Eng.* 37, 2 (March 2011), 161–187. DOI: https://doi.org/10.1109/TSE.2010.32

[68] Shinichi Nakagawa. 2004. A farewell to Bonferroni: The problems of low statistical power and publication bias. *Behav. Ecol.* 15, 6 (11 2004), 1044–1045. DOI: https://doi.org/10.1093/beheco/arh107

[69] Alessandro Orso and Gregg Rothermel. 2014. Software testing: A research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering.* ACM, 117–132.

[70] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07).* 75–84.

[71] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST'15).* IEEE Computer Society, 1–10. DOI: https://doi.org/10.1109/ICST.2015.7102604

[72] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Trans. Softw. Eng.* 44, 2 (2018), 122–158. DOI: https://doi.org/10.1109/TSE.2017.2663435

[73] Thomas V. Perneger. 1998. What's wrong with Bonferroni adjustments. *Br. Med. J.* 316, 7139 (1998), 1236–1238. DOI: https://doi.org/10.1136/bmj.316.7139.1236 arXiv:https://www.bmj.com/content

[74] Michael Pradel, Philipp Bichsel, and Thomas R. Gross. 2010. A framework for the evaluation of specification miners based on finite state machines. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM'10).* IEEE Computer Society, 1–10. DOI: https://doi.org/10.1109/ICSM.2010.5609576

[75] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. 2005. One evaluation of model-based testing and its automation. In *Proceedings of the International Conference on Software Engineering (ICSE'05).* 392–401.

[76] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. 2005. One evaluation of model-based testing and its automation. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05).* ACM, New York, NY, 392–401. DOI: https://doi.org/10.1145/1062455.1062529

[77] J. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. [n.d.]. Combining multiple coverage criteria in search-based unit test generation. In *Proceedings of the Symposium on Search Based Software Engineering (SSBSE'15).* 93–108.

[78] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Emp. Softw. Eng.* 22, 2 (2017), 852–893. DOI: https://doi.org/10.1007/s10664-015-9424-2

[79] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. 1998. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance (ICSM'98).* IEEE Computer Society, Los Alamitos, CA, 34–. http://dl.acm.org/citation.cfm?id=850947.853294

[80] M. J. Rutherford, A. Carzaniga, and A. L. Wolf. 2008. Evaluating test suites and adequacy criteria using simulation-based models of distributed systems. *IEEE Trans. Softw. Eng.* 34, 4 (July 2008), 452–470. DOI: https://doi.org/10.1109/TSE.2008.33

[81] V. Santiago, A. S. Martins Do Amaral, N. L. Vijaykumar, M. D. Fatima Mattiello-francisco, E. Martins, and O. C. Lopes. 2006. A practical approach for automated test case generation using statecharts. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, Vol. 2. 183–188. DOI: https://doi.org/10.1109/COMPSAC.2006.100

[82] Muhammad Shafique and Yvan Labiche. 2015. A systematic review of state-based test tools. *Int. J. Softw. Tools Technol. Transf.* 17, 1 (2015), 59–76.

[83] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 201–211. DOI: https://doi.org/10.1109/ASE.2015.86

[84] Sina Shamshiri, José Miguel Rojas, Luca Gazzola, Gordon Fraser, Phil McMinn, Leonardo Mariani, and Andrea Arcuri. 2018. Random or evolutionary search for object-oriented test suite generation? *Softw. Test. Verif. Reliabil.* 28, 4 (2018), e1660.

[85] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov. 2011. Testing container classes: Random or systematic? In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE'11).* 262–277.

[86] M. Srinivas and Lalit M. Patnaik. 1994. Genetic algorithms: A survey. *IEEE Comput.* 27, 6 (1994), 17–26. DOI: https://doi.org/10.1109/2.294849

[87] Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.* SE-12, 1 (Jan. 1986), 157–171. DOI: 10.1109/TSE.1986.6312929

[88] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. 2011. Synthesizing method sequences for high-coverage testing. In *Proceedings of the ACM Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'11).*

[89] P. Tonella. 2004. Evolutionary testing of classes. *SIGSOFT Softw. Eng. Notes* 29, 4 (July 2004), 119–128.

[90] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. 1993. Mutation analysis using mutant schemata. In *ACM SIGSOFT Software Engineering Notes*, Vol. 18. ACM, 139–148.

[91] Mark Utting and Bruno Legeard. 2010. *Practical Model-based Testing: A Tools Approach.* Elsevier.

[92] M. Utting, G. Perrone, J. Winchester, S. Thompson, R. Yang, and P. Douangsavanh. 2009. Model junit. *Department of Computer Science, University of Waikato, Waikato, New Zealand* (2009).

[93] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliabil.* 22, 5 (2012), 297–312.

[94] Machiel van der Bijl, Arend Rensink, and Jan Tretmans. 2003. Compositional testing with ioco. In *Formal Approaches to Software Testing, Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES'03)*, Alexandre Petrenko and Andreas Ulrich (Eds.), Lecture Notes in Computer Science, Vol. 2931. Springer, 86–100. DOI: https://doi.org/10.1007/978-3-540-24617-6_7

[95] S. Weißleder. 2010. Simulated satisfaction of coverage criteria on UML state machines. In *Proceedings of the 2010 3rd International Conference on Software Testing, Verification and Validation.* 117–126. DOI: https://doi.org/10.1109/ICST.2010.28

[96] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. [n.d.]. Precise identification of problems for structural test generation. In *Proceedings of the International Conference on Software Engineering (ICSE'11).* 611–620.

[97] Dianxiang Xu, Weifeng Xu, Michael Kent, Lijo Thomas, and Linzhang Wang. 2015. An automated test generation technique for software quality assurance. *IEEE Transactions on Reliability* 64, 1 (2015), 247–268.

Received June 2019; revised July 2020; accepted July 2020