Leveraging Test Generation and Specification Mining for Automated Bug Detection without False Positives

Michael Pradel
Department of Computer Science
ETH Zurich

Thomas R. Gross
Department of Computer Science
ETH Zurich

***Abstract*—Mining specifications and using them for bug detection is a promising way to reveal bugs in programs. Existing approaches suffer from two problems. First, dynamic specification miners require input that drives a program to generate common usage patterns. Second, existing approaches report false positives, that is, spurious warnings that mislead developers and reduce the practicability of the approach. We present a novel technique for dynamically mining and checking specifications without relying on existing input to drive a program and without reporting false positives. Our technique leverages automatically generated tests in two ways: Passing tests drive the program during specification mining, and failing test executions are checked against the mined specifications. The output are warnings that show with concrete test cases how the program violates commonly accepted specifications. Our implementation reports no false positives and 54 true positives in ten well-tested Java programs.**

***Keywords*-Bug detection; Specification mining; False positives**

I. Introduction

Programs often access libraries and frameworks via application programming interfaces (APIs). While this approach reduces development effort by reusing existing functionality, it entails challenges for ensuring the program's correctness. Many APIs impose *protocols*, that is, temporal constraints regarding the order of the calls of API methods. For example, calling `peek()` on `java.util.Stack` without a preceding `push()` gives an `EmptyStackException`, and calling `next()` on `java.util.Iterator` without checking whether there is a next element with `hasNext()` can result in a `NoSuchElementException`. API clients that violate such protocols do not obtain the desired behavior and may even crash the program.

This paper presents a fully automatic approach to reveal violations of API protocols. Our work builds upon three existing techniques that—on their own—do not fully address the problem. First, we build upon *automatic test generation* [1]–[3]. While test generators achieve good coverage, their usefulness is bounded by the test oracle that decides when to report a potential bug. For example, the Randoop test generator [2] comes with a small set of manually specified, generic test oracles that do not detect all bugs triggered by the generated tests.

Second, we build upon *dynamic protocol mining* [4], [5]. Protocol miners infer likely API protocols from executions of API clients. The usefulness of a dynamic protocol miner is bounded by the executions it analyzes [6]. A protocol miner can infer a protocol only if there is input that exercises paths of an API client that expose the protocol.

Third, we build upon *dynamic protocol checking* [7]–[9], which verifies whether a program execution violates a given protocol. Similar to a protocol miner, the usefulness of a protocol checker is bounded by the executions it analyzes. The checker can report a protocol violation only if some input triggers the illegal behavior.

As a motivating example, consider a simplified version of a problem our analysis found in *xalan*, a program shipped with the DaCapo benchmarks [10]: A class `C` offers a public method `report()`. Calling `report()` on a fresh instance of `C` gives an `EmptyStackException`, which is not declared in the public interface of `report()`. The problem is that `C` uses the `Stack` API in an unsafe way that exposes users of `C` to exceptions caused by violations `Stack`'s API protocol. A class using an API should ensure that the API is used safely, that is, without the risk to throw an exception due to a protocol violation. A user of `C` cannot foresee that `C` may crash the program, unless she inspects `C`'s implementation, which however, contradicts the idea of modular, object-oriented programming.

Can test generation, protocol mining, and protocol checking help finding the problem in class `C`? A test generator can produce tests that cover many paths through `C`, including a path that triggers the `EmptyStackException`. Unfortunately, an exception alone is an unreliable indicator for a bug: Generated tests produce many exceptions due to illegal usage, for example, by passing illegal method arguments. A protocol miner can infer the protocol for using `Stack`s but requires input that drives clients of the `Stack` class. Finally, a runtime protocol checker can detect that `C` violates the `Stack` protocol but requires the protocol and input that drives `C` into the protocol violation.

We present a fully automatic approach to reveal API protocol violations by combining test generation, protocol mining, and protocol checking. Our approach consists of three steps. First, we generate tests for the classes of a program. Some tests fail, because they result in an uncaught exception, while other tests pass. Second, we analyze executions of passing tests with a protocol miner, giving likely specifications of how to use API classes. Third, we verify executions of failing tests against the mined protocols and report a warning if a test fails because of a protocol violation.

The approach is easily applicable in practice because it has a high benefit-cost ratio. All the input required by the analysis is the program itself. The approach does not require specifications, as these are extracted by the protocol miner. Furthermore, it does not require any input to run the program, because all the input is automatically generated. The output of the analysis is precise and concrete. It is precise, because each warning reveals a protocol violation that certainly leads the program to an exception. It is concrete, because each warning comes with a test case demonstrating how to trigger the problem.

A main benefit of the approach is that all reported warnings are true positives. The analysis reports a warning only if a protocol is violated and if the protocol violation causes an uncaught exception. As a result, each warning exposes an unsafe API usage where a class uses an API in a way that exposes clients of the class to the risk of unintentionally violating the API's protocol. In principle, the approach might report a false warning when two conditions hold. First, an API method `m` must be implemented incorrectly and throw an exception despite being called legally. Second, a mined protocol must forbid calling `m` at a state where a permissive protocol [11] would allow it. One of the two conditions alone does not lead to false warnings. In particular, incomplete protocols alone, which may result from dynamic protocol mining, do not cause false positives. In practice, both conditions together are very unlikely to occur and we did not encounter this situation during our experiments.

We implement the approach based on two existing tools: a dynamic protocol miner to extract API protocols [5], [6] and a random test generator, Randoop [2]. To evaluate the approach, we apply it to ten real-world Java programs [10]. The analysis reports no false positives and 54 true bugs, all of which are illegal API usages that can be triggered via public methods of the programs. Applied to the same programs, the best existing approach to dynamically find violations of API protocols without a priori known specifications has reported three warnings, two of which are false positives [12].

As an optional, heuristic optimization of our approach, we modify Randoop by adding a static analysis that guides the random test generation towards those methods that use the API in question. Without this guidance, the test generator randomly chooses the methods to call from all methods of a program. If only a small part of the program uses the API, most of the testing effort is wasted. Our analysis statically prioritizes methods that are likely to contribute to calls from the program to the API, so that API usage bugs are revealed faster. Compared to standard Randoop, this optimization increases the number of API calls by a factor of 57 and reveals bugs with five times less testing effort, on average.

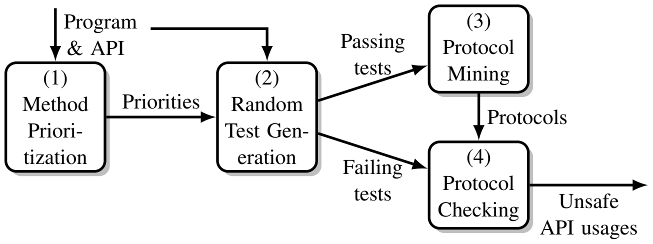


Figure 1. Overview.

In summary, this paper makes the following contributions:

- *No false positives.* An analysis to detect bugs based on mined specifications without reporting false positives.
- *Mining and checking driven by generated tests.* Our approach drives both specification mining and program checking with automatically generated input.
- *Guided random test generation.* We reduce the time to trigger an API usage bug by guiding random test generation towards a set of API methods.

II. Learning from the Good to Find the Bad

In this section, we describe our approach to leverage generated tests to reveal API protocol violations without reporting false positives. Figure 1 illustrates the major steps. Given a program and an API as input, we use a random test generator (2) to generate tests for the program. The generated tests can be classified into two groups: *failing tests* that lead the program to an exception or that violate an assertion, and *passing tests* that execute without any obvious error. We analyze executions of passing tests with a dynamic protocol miner (3), which produces protocols describing common API usage patterns (Section II-B). Then we analyze executions of failing tests with a dynamic protocol checker (4) (Section II-C). This last step filters the various failing tests by selecting those that fail because the program violates a protocol, removing all tests that fail for other reasons, such as illegal input (Section II-D). Finally, the protocol violations are reported to the developer.

Figure 1 also shows *method prioritization* (1), a static analysis that heuristically computes how relevant each of the program's methods is for triggering calls into the API. Method prioritization is an optional optimization and can be ignored for the description of our general approach. Section III explains how method prioritization makes our technique even more useful in practice.

A. Background

1) Protocol Mining: Specification mining extracts formal specifications from existing software. This work focuses on protocols that describe in which order API methods

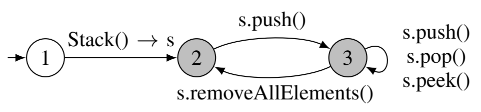


Figure 2. Mined protocol for `java.util.Stack`. States with gray background are liable states.

are commonly called. Several techniques for mining such protocols from an API [13] or from programs using an API [4], [14]–[18] have been proposed. We use a scalable protocol miner that infers protocols involving one or more interacting objects [5], [6]. Figure 2 shows an example of a mined protocol for `Stack`. An empty stack allows only calls to `push()`. Once elements have been pushed on the stack, one can call `push()`, `pop()`, and `peek()` in an arbitrary order, until calls to `removeAllElements()` empty the stack again. This protocol is a regular language approximation of the more precise property that `push()` and `pop()` are balanced. Despite this approximation, the protocol can reveal illegal uses of stacks, as we show subsequently.

A protocol consists of a finite state machine and a set of typed protocol parameters that represent the objects involved in the protocol. The states of a protocol represent states of the involved objects. The transitions represent method calls, where receiver and, optionally, method parameters and the return value are protocol parameters. A call $r.m(a_1, \ldots, a_n) \rightarrow o$ encodes the method identifier m, the receiver r, the arguments $a_1, \ldots, a_n$, and the return value o. The protocols created by the protocol miner distinguish two kinds of states: *setup states* and *liable states*. The setup states establish which objects interact in the protocol by binding objects to protocol parameters. The liable states describe constraints on method calls that a programmer ought to respect. A protocol must be followed only when the current state is a liable state.

The protocol miner requires an *execution trace*, that is, a sequence of method calls observed during an execution. The protocol miner splits the execution trace into small *subtraces* that contain calls to one API object or to a set of interacting API objects. For a subtrace sub, $types(sub)$ denotes the set of receiver types occurring in sub. The miner summarizes all subtraces with the same $types(sub)$, for example all subtraces for `Stack`s, into a protocol. Whenever a subtrace contains a method call sequence not yet accepted by the protocol, the protocol is extended to accept the subtrace. The resulting protocol accepts all observed subtraces.

2) *Random Test Generation:* There are various approaches to automatically generate tests (Section VI-B). This work builds upon feedback-directed random test generation, implemented in Randoop [2]. We use Randoop because it requires no input except the program under test, scales to large programs, and generates tests with high coverage. Randoop randomly selects methods and chains them into call sequences. If executing a call sequence neither produces an exception nor violates an assertion, the sequence is used to construct further call sequences. The output are passing and failing JUnit tests.

B. Driving Protocol Mining with Passing Tests

Dynamic protocol mining is inherently limited by the execution traces given to the miner, which in turn is limited by the input used to execute the program. Most programs come with a finite set of inputs, for example, given as unit tests. Producing more input requires manual effort, reducing the overall usefulness of protocol mining.

We address this limitation by combining protocol mining with test generation. The idea is to analyze an execution of a generated test in the same way as traditional specification mining analyzes executions driven by otherwise available input. A benefit of this approach is that generated tests are a large and diverse source of inputs. Random test generation provides fresh inputs for different random seeds, which can trigger behavior not yet triggered with other random seeds [19].

To mine the API usage of a program driven by generated tests we proceed as follows. At first, we use AspectJ to instrument the program so that it writes an execution trace with all API calls to a file. We create one file per generated sequence of method calls. Then, the protocol miner analyzes the execution traces and extracts protocols describing in which order the program calls API methods. Call sequences of different tests that trigger the same API class contribute to the same protocol. The main difference to traditional specification mining is that we drive the program with generated tests instead of other input.

A potential problem of generating tests for protocol mining is that artificial input may trigger illegal API call sequences that can lead to incorrect protocols. We address this problem in two ways. First, we feed only execution traces of passing tests into the protocol miner, not execution traces of failing tests. While a passing test does not guarantee to use the program in the intended way, we found that in practice, most illegal inputs are filtered, because they would lead to an exception. Second, the execution trace contains only API calls from the program, not calls from the generated tests (a generated test can call API methods, for example, to create an API object before passing it as a parameter). That is, each API call the miner learns from is part of the program and not a call that was randomly generated by Randoop.

C. Checking Failing Tests against Mined Protocols

A dynamic protocol checker can verify whether a program violates commonly accepted constraints by checking the program against mined protocols. For a given program execution, a checker guarantees soundness and completeness: All reported protocol violations provably occur in the execution, and all protocol violations in the execution are reported. Unfortunately, this approach can only reveal a problem if the problem occurs in the analyzed program execution, that is, if the problem is triggered by the given input. For example, checking the execution of a manually created unit test suite against a set of protocols is unlikely to reveal many protocol violations, because the test suite exercises well-tested paths of the program.

We address this limitation by driving the program with generated tests. In contrast to the mining step of our approach, we now use generated tests that fail. These tests are classified as failing because they trigger a problem in the program that leads to an exception or an assertion violation.

Many failing tests do not expose any bug but fail because the generated test uses the program incorrectly. For example, a test may fail with a `NullPointerException` or an `IllegalArgumentException`, because the test passes `null` or another illegal value as a method argument. Such tests are not relevant for programmers, as they do not reveal bugs in the program. To focus on tests that fail because of a bug in the program, we check whether the program violates a protocol during test execution, as those tests expose an unsafe API usage in the program and therefore are relevant for programmers. That is, we use protocol checking as a filter that identifies true bugs among all failing tests.

The checker verifies the execution trace of each failing test against the set of mined protocols. Conceptually, the approach is similar to existing runtime verification tools, such as JavaMOP [9]. Our implementation uses the infrastructure built for mining protocols. At first, the execution trace is split into subtraces in the same manner as for protocol mining [6]. Then, each subtrace sub is verified against all protocols that have protocol parameters with types equal to $types(sub)$. The checker assigns each receiver object in the subtrace to a protocol parameter of matching type. For subtraces with multiple receiver objects of the same type, the checker creates multiple protocol instances, one for each possible assignment of objects to protocol parameters.

At the beginning of a subtrace, the protocol is in the initial state. The checker goes through all calls in the subtrace and analyzes all calls that appear in the alphabet of the protocol. Each such call is matched against the outgoing transitions of the current state. As all mined protocols are deterministic, each call matches at most one transition. If a transition matches, the target state of the transition becomes the current state. If no transition matches and the current state is a liable state, the checker reports a protocol violation. Non-matching transitions in setup states are ignored.

For example, consider the source code in Figure 3a, which is the problem described in Section I. The class `C` uses a stack in an unsafe way, because a client of `C` can trigger a violation of the stack protocol via `C`'s public interface.

```
class C {
  private Stack s = new Stack();
  public String report() {
    return get().toString();
  }
  private Object get() {
    s.peek();
  }
  public void fill() {
    s.push(..);
  }
}
```

(a) Source code

```
new Stack() → s
s.peek()
# EmptyStackException
# at Stack.peek()
# at C.get()
# at C.report()
# ...
```

(b) Execution trace

Figure 3. Example code that can violate the `Stack` protocol and execution trace exposing the violation.

We consider this to be an API usage bug, because `C`'s public interface does not reveal this risk, and understanding it requires knowledge of `C`'s implementation.

Our approach finds this problem because Randoop generates a test that triggers an API protocol violation by calling `C`'s methods. The test calls `report()` without a preceding call to `fill()` and therefore triggers a call to `peek()` while the stack is empty. This call causes an `EmptyStackException` and fails the test. Figure 3b shows the execution trace of the failing test. The trace contains all calls to `Stack` that were observed during the test execution. The checker finds a violation of the protocol in Figure 2 because the second call in the trace tries to call `peek()` at state 2, but `peek()` is only allowed at state 3.

D. Warnings without False Positives

Reporting few (ideally no) false positives is crucial to make a bug finding technique applicable in practice [20]. By using runtime checking, our approach avoids a problem typically encountered with static checkers, namely false positives due to approximations of execution paths. Another potential source of false positives are mined protocols that disallow legal behavior because this behavior was not observed during protocol mining. Our approach eliminates this problem by only reporting those protocol violations that certainly lead to an exception thrown by the API, that is, to undesired behavior. For example, the call to `peek()` in Figure 3 results in an `EmptyStackException`, a clear indication that the API is used incorrectly.

To filter protocol violations that cause an exception, we need several pieces of information. For each failing test, the analysis logs the reason for the failure, which consists of two parts: the type T_{exc} of the thrown exception and the stack trace $S = (loc_1, loc_2, \ldots, loc_n)$ when throwing it. Each stack trace element loc_i indicates a source code location. For each protocol violation, the protocol checker reports the source code location loc_{viol} where the violating API call occurs. Furthermore, the analysis statically extracts the set of declared exception types D_{API} of the API method that is called at loc_{viol} and the set of declared exception types

$D_{program}$ of the method containing loc_{viol}. An exception can be declared using the `throws` keyword or using the `@throws` or `@exception` Javadoc tags.

Our analysis reports a protocol violation to the user only if the following three conditions are true:

1) $T_{exc} \in D_{API}$
2) $loc_{viol} \in S$
3) $T_{exc} \notin D_{program}$

The first two conditions ensure that the call that violates the protocol is responsible for failing the test. The third condition avoids warnings for methods that deliberately pass on exceptions thrown by the API to their own clients. In this case, the protocol-violating class is implemented safely, because it explicitly passes on responsibility for using the API correctly to its clients. For Figure 3, all three conditions hold and our analysis reports an unsafe API usage, along with a test case that illustrates how to trigger the problem.

III. API-Guided Test Generation

Section II describes how randomly generated tests can drive a program for mining API protocols and for checking the program against mined protocols. With a random test generation tool, such as Randoop, many generated tests do not trigger API calls. The reason is that even though only a subset of all methods of a program uses an API, Randoop chooses which method to call randomly from all available methods. This section presents a heuristic to optimize our approach by guiding random test generation towards methods that trigger API calls.

Given infinite time, standard Randoop triggers the same behavior as a guided approach focusing on methods relevant for calling the API. In practice, increasing the number and variety of API calls triggered by generated tests within a finite amount of time is important for two reasons. First, random test generators require a stopping criterion—typically, wall clock time or the number of generated method call sequences. Second, random test generators execute arbitrary code, making them susceptible to crashes that are hard to avoid in general [21]. Whatever stops the test generator, triggering API calls earlier than a purely random approach not only saves time, but also is likely to trigger more API usage bugs before the test generator terminates.

To guide Randoop towards methods relevant for triggering API calls, we statically analyze the program and prioritize its methods. The priorities influence the random decisions of the test generator so that calling a method with higher priority is more likely. Methods that do not contribute at all to calling the API get a priority of zero and are ignored altogether by the test generator.

A. Computing Method Priorities

We build upon two well-known graph representations of the program under test: an inverted, context-insensitive call graph and a parameter graph [1]. The inverted call graph is a directed graph where vertices represent methods and edges represent a "may be called by" relationship between methods. We extract call graphs with Soot [22]. The parameter graph is a directed graph where vertices also represent methods and where edges represent a "can use the result of" relationship between methods. There is an edge from m_1 to m_2 if m_1 requires an object of type T (as receiver or as argument) and if m_2's return type is equal to T or a subtype of T.

Test generators call methods for three reasons. First, because a method is among the *methods under test*. In our case, these are all methods that call the API. Second, because the method returns an object that can be passed as an argument to another method, that is, the method is a *parameter provider*. Third, because the method may change the internal state of an object that is used for another call afterwards, that is, the method is a *state changer*. Our prioritization technique considers these reasons and computes for each method three priority values that indicate how relevant the method is for each reason. Afterwards, the priorities are combined into a single priority value per method.

Algorithm 1 describes how we compute the priority of each method. The algorithm takes as input the call graph G_c, the parameter graph G_p, the set of all methods M, and the set M_{API} of API methods. There are four main steps.

Algorithm 1 Compute priorities for calling methods during random test generation

```
Input: Inverted call graph $G_c$, parameter graph $G_p$, methods $M$, API methods $M_{API}$
Output: Map $p : method \to \mathbb{R}$ assigning priorities to methods
    /* priorities for methods under test */
 1: initialize $p_{mut} : method \to \mathbb{R}$ to zero
 2: for all $m \in M$ do
 3:     for all $m_{API} \in M_{API}$ with $d_c(m_{API}, m) \leq d_c^{max}$ do
 4:         $p_{mut}(m) \mathrel{+}= \frac{relevance(m_{API})}{d_c(m_{API}, m)}$
    /* priorities for parameter providers */
 5: initialize $p_{param} : method \to \mathbb{R}$ to zero
 6: for all $m \in M$ with $p_{mut}(m) > 0$ do
 7:     for all $m' \in reachable(G_p, m)$ do
 8:         $p_{param}(m') \mathrel{+}= \frac{p_{mut}(m)}{nbProviders(retType(m'))}$
    /* priorities for state changers */
 9: initialize $p_{state} : method \to \mathbb{R}$ to zero
10: for all $m \in M$ with $p_{mut}(m) > 0$ do
11:     for all $t \in inputTypes(m)$ do
12:         for all $m' \in methods(t)$ do
13:             $p_{state} \mathrel{+}= \frac{p_{mut}(m)}{|methods(t)|}$
14: $p = merge(p_{mut}, p_{param}, p_{state})$
```

1) Priorities for Methods Under Test: At first, we compute how relevant a method is depending on whether it calls any of the API methods. Methods calling an API method directly are the most relevant ones. In addition, methods calling API methods indirectly, that is, via other methods, are given a lower priority. The reason for considering indirect calls is that some methods should be called at a state built up by another method. For instance, a method m_1 may create a file and call another method m_2 that writes into the file. By calling m_1, the test generator can trigger successful writes into the file, even though m_1 calls the file API only indirectly. We limit the level of indirection up to which to consider indirect callers of API methods to d_c^{max}, which is set to three in our experiments. The priority gain of a method calling an API method depends on two factors. First, it increases with the relevance of the called API method:

$$relevance(m_{API}) = \frac{1}{|\{m \mid (m_{API}, m) \in G_c\}|}$$

Less frequently called methods are more relevant so that all API methods get an equal chance of being triggered. Second, the priority gain decreases with the minimal distance $d_c(m_{API}, m)$ in G_c between the API method m_{API} and the calling method m. The rationale for this decrease is that the call graph contains may-call and not must-call edges; fewer indirections to an API call increase the chance that the API is actually called during an execution.

2) Priorities for Parameter Providers: We compute how relevant a method is for providing parameters for calling methods under test. Therefore, we consider all methods $reachable(G_p, m)$ that are reachable from a method under test m in the parameter graph. These are all methods that directly or indirectly contribute parameters for calling the method under test. The gain of priority for such a parameter provider depends in two factors. First, it increases with the priority of the method it provides parameters for. Second, the gain of priority decreases with the overall number of providers for the type required by the method under test. The rationale for this decrease is to give higher priority to methods providing an uncommon type than to methods providing a type returned by many other methods.

3) Priorities for State Changers: The third kind of priority is for methods changing the internal state of objects used for calling methods under test. The types $inputTypes(m)$ calling a method m are the receiver type and all argument types of m. We consider all methods of input types for methods under test as potentially state-changing and increase their priority. The priority gain depends on two factors. First, it increases with the priority of the method under test that the state changer may influence. Second, the priority gain decreases with the overall number of methods of the input type. The rationale for this decrease is to give objects of each input type an equal chance of being changed, independently of the number of methods of the type.

```
class A {
  API api;
  A(B b) { .. }
  void init() { api = APIPool.some; }
  void doIt() { api.use(); }
}
class B {
  B(C c) { .. }
  void m(D d) { .. }
}
class C {}
class D {}
```

Method	Priority
`A.doIt()`	107
`A()`	33
`B(C)`	27
`C()`	27
`A.init()`	7
`B.m(D)`	0
`D()`	0

Figure 4. Example for prioritizing methods towards using the API.

4) Merging: The final step is to normalize the three kinds of priorities and to combine the three priorities of each method in a weighted sum. The weights indicate how important calling methods under test, calling parameter providers, and calling state changers are, respectively. In our experiments, the test generator devotes 20% of all calls to calling state changers and 40% to each calling methods under test and to calling parameter providers.

B. Example

Figure 4 shows a minimal example to illustrate how prioritizing methods guides random test generation towards calling API methods. The example contains seven methods and constructors, of which one, `doIt()`, calls the API. Algorithm 1 finds this method as the only method under test and assigns a high priority to it. To call `doIt()`, the test generator requires an object of type `A`, which in turn requires objects of types `B` and `C`. Our algorithm finds the constructors of these classes as parameter providers and gives them a medium priority. The API call in `doIt()` results in a NullPointerException unless `init()` is called beforehand. This method is found to be a state changer and also gets a non-zero priority. The remaining two methods, `B.m()` and `D()`, do not contribute to calling the API and get a priority of zero. Given the priorities, the random test generator selects only methods that eventually contribute to calling the API and probabilistically leads to more calls to the API than a purely random approach. We show this claim to be valid for real-world programs and APIs in Section IV-C.

IV. Evaluation

Our evaluation was driven by two main questions.

1) *How effective is our approach in finding unsafe API usages?* We find 54 unsafe API usages in ten well-tested Java programs. Manual inspection shows all of them to be true positives.
2) *How much does the testing effort reduce compared to a purely random approach by guiding test generation towards an API?* Our heuristic optimization reveals bugs with five times fewer generated call sequences than the unguided approach, on average. This improvement is

Table I
ANALYZED PROGRAMS, NUMBER OF FAILING TESTS (OUT OF 10,000), AND API USAGE BUGS FOUND.

Program	Classes	Failing tests		Bugs found		
		(Average per run)		Average per run		Total
		Coll/Iter	Vector/Enum	Coll/Iter	Vector/Enum	
antlr	199	1,429	1,695	0	0	0
bloat	331	4,098	4,042	0	0	0
chart	481	2,336	2,396	1.4	1	4
eclipse	384	2,817	3,375	0	0	0
fop	969	3,148	2,948	3.2	2.8	9
hsqldb	383	2,246	2,264	0	0.1	1
jython	890	4,314	5,169	1.4	0	2
lucene	316	3,573	3,372	1.9	1.1	5
pmd	524	2,155	4,158	12.8	5.9	16
xalan	535	4,440	4,395	6.6	5.9	17
Sum	5,012	3,056	3,381	27.3	16.8	54

Table II
ANALYZED APIS.

API	Classes
Coll/Iter	All public methods of `java.util.Collection`, `java.util.Iterator`, and their subclasses.
Vector/Enum	All public methods of `java.util.Vector`, `java.util.Enumeration`, and their subclasses.

possible because method prioritization increases the number of API calls by a factor of 57 and decreases the sequences required to call a particular API method by a factor of seven.

A. Setup

We run our analysis on all programs from the DaCapo benchmarks, version 2006-10-MR2 (Table I) [10]. The analysis focuses on API classes of the Java library (Table II).

We run the analysis in two modes: *unguided* and *guided* towards an API. Unguided runs use Randoop's standard test generation technique. Guided runs use method prioritization (Section III). To evaluate guided runs, we run Randoop for each program-API pair. To evaluate unguided runs, we run Randoop on each program. Since Randoop is based on random decisions, we perform each experiment with ten different random seeds [2]. Unless otherwise mentioned, the reported results are average values of all ten runs.

Randoop requires a stopping criterion. We use a maximum number of generated method call sequences of 10,000. We prefer this criterion over a maximum running time, because it is easily reproducible, whereas Randoop's execution time depends on the environment used for the experiments.

B. Detection of Protocol Violations

The analysis finds a total of 54 protocol violations. Each static source code location with a protocol violation counts only once, even if it occurs in multiple failing tests. Table I

```
public class MIFDocument {
    protected BookComponent bookComponent;
    public MIFDocument() {
        bookComponent = new BookComponent();
    }
    public void setTextRectProp(int left, int top,
                                int width, int height) {
        (bookComponent.curPage()).curTextRect()
            .setTextRectProp(left, top, width, height);
    }
    public void createPage() {
        bookComponent.pages.add(new Page());
    }
    class BookComponent {
        ArrayList pages = new ArrayList();
        private Page curPage() {
            return (Page)pages.get(pages.size() - 1);
        }
    }
}
```

(a) Incorrect usage of `ArrayList` in *fop*.

```
public class InternalTables2 {
    protected Iterator iter;
    public void _beginCanonical() {
        iter = ((TableProvid2)classes).values().iterator();
    }
    public Object _next() {
        if (iter.hasNext()) {
            cur = iter.next();
            return (PyJavaClass)cur;
        }
        return null;
    }
    public void _flushCurrent() {
        iter.remove();
        classesDec(((PyJavaClass)cur).__name__);
    }
}
```

(b) Incorrect usage of `Iterator` in *jython*.

Figure 5. Examples of bugs found.

shows the number of violations detected for each program-API pair, along with the total number of violations found over all runs. The table also shows the number of failing tests in which the protocol checker finds the bugs. On average, 3,219 of 10,000 tests fail—too much for a programmer to inspect. The protocol checker filters these tests and presents only those that expose a protocol bug.

We inspect all protocol violations manually to verify that no false positives are reported. Indeed, all reported violations show an illegal API usage that can be triggered via the public methods of a class in the program. Furthermore, we manually check whether the comments of the buggy methods contain any natural language warnings about the exceptions that may result from these methods. None of the reported methods has such a comment. That is, all reported bugs are real problems in the programs.

Figure 5 shows two examples of bugs found. The first example, Figure 5a, was found in *fop*. The class `MIFDocument` contains an inner class `BookComponent` that stores pages in a list, which initially is empty. The inner class provides a method `curPage()` that tries to return the last page in the list. This method is called in the public method `setTextRectProp()`, which raises an `IndexOutOfBoundsException` when the list of pages is empty. This risk is not apparent to clients of `MIFDocument`, and hence, the class uses `ArrayList` in an unsafe way. Our analysis finds this bug because the mined protocol for `ArrayList` disallows `get()` directly after creating the list.

The second example, Figure 5b, is taken from *jython*. The class has an `Iterator` field that is accessed by multiple public methods. One of them, `_flushCurrent()`, calls `remove()` on the iterator. The iterator protocol allows this call exactly once after each call to `next()`. This constraint is not respected in the class, causing a potential `IllegalStateException` when calling `_flushCurrent()`. Similar to the first example, the class uses an API in an unsafe way and does not make the potential problem apparent to its clients. Our analysis infers the iterator protocol and finds this violation of it.

Descriptions of all detected bugs are available at http://mp.binaervarianz.de/icse2012-leveraging.

An obvious question about bug reports is how to fix the bugs. During our experiments, we have seen two strategies to ensure safe usage of the API and conformance to its protocols. First, an API-using method `m()` can ensure to follow the protocols of all API objects that `m()` uses and not to propagate any exception from the API to callers of `m()`. Often, such methods return a special value, for example `null`, when they cannot return another value. Second, an API-using method `m()` can propagate API exceptions and declare this in its signature. This approach indicates to callers of `m()` that calling the method at an illegal state or with illegal input may result in an exception, and hence, `m()` explicitly passes the responsibility on to its callers.

Interestingly, many of the detected bugs are surrounded by attempts to shield users of a class from protocol violations. For example in Figure 5b, the `_next()` method checks by calling `hasNext()` whether a call to `next()` is legal and returns `null` otherwise. That is, the programmers protect clients of the class from some violations of the iterator protocol but, unfortunately, forgot about another violation.

Comparison to Existing Approaches: We directly compare our analysis to OCD [12], a combined dynamic protocol miner and checker. OCD also searches for protocol violations by mining and checking protocols at runtime. Two main differences are that OCD relies on existing input to drive the program and that the analysis combines mining and checking into a single run of the program under test. Gabel and Su used OCD to analyze the usage of the Java standard library in the DaCapo programs. Their technique reports three warnings, matching Gabel and Su's expectation to find few problems in these well-tested benchmarks. Manual inspection showed two warnings to be clear false positives and one warning to be a potential problem.

Applied to the same programs, our analysis reports 54 warnings that are all true positives. What are the reasons for these differences? First, our analysis does not report false positives by construction, as explained in Section II-D. Second, OCD is limited to the program paths triggered by available input (here, DaCapo's benchmark input). In contrast, the generated tests used by our analysis exercise less frequently tested parts of the programs and trigger API usage bugs not exposed with the benchmark inputs.

C. API-Guided Test Generation

The goal of guiding test generation towards an API is to reduce the overall time required to reveal API usage bugs. The most important factor for reducing this time is the number of generated call sequences, which controls the running times of test generator, protocol miner, and protocol checker. Therefore, we use the number of generated call sequences as a metric for testing effort.

We compare guided runs of our analysis to unguided runs (Table III). Do guided runs trigger more API calls than unguided runs within a fixed number of generated call sequences? The first block of columns in Table III compares the number of calls to an API. Each value is the average over ten runs of Randoop with different random seeds. For all but one program-API pair, guided runs trigger more API calls. For example, the number of calls to Vector/Enum triggered in tests for *antlr* increases from 6,146 to 58,438. On average over all programs, APIs, and runs, guidance improves the number of API calls by 56.7x (median: 5.2x).

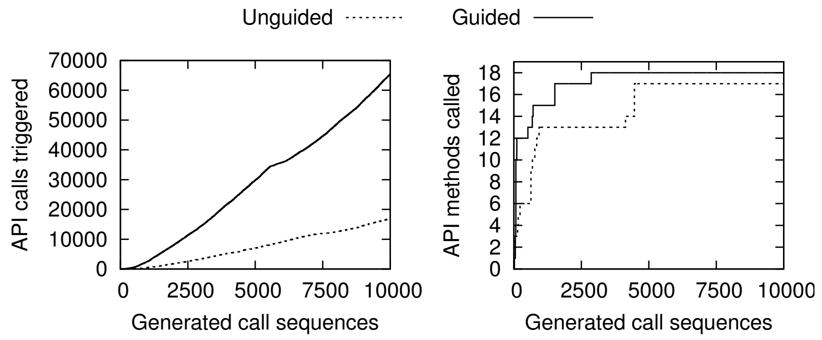
Increasing the number of API calls is important to give the miner different examples to learn from and to increase the chance to hit a bug. It is also important to trigger calls to *different* API methods to expose a broad variety of API protocols and their violations. The second block of columns in Table III compares the number of distinct API methods called. In most cases, there is no significant difference between guided and unguided runs. The reason is that our stopping criterion is large enough to give even unguided runs a realistic chance to hit each API method.

How long does it take the test generator to hit an API method? The third block of columns in Table III shows after how many generated sequences an API method is called the first time. For all but one program-API pair, API methods are called earlier in guided runs than in unguided runs. On average over all programs, APIs, and runs, the guided approach calls a method 6.9 times faster than the unguided approach. That is, even though after enough time the same set of API methods is called, guided runs achieve this goal much faster than unguided runs.

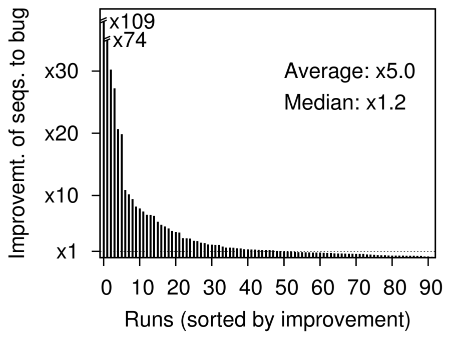
Figure 6a illustrates an unguided and a guided run. The graphs show the cumulative number of API calls and the number of distinct called API methods depending on how many sequences Randoop has generated, respectively. The first graph shows that the number of API calls increases

Table III
COMPARISON OF UNGUIDED AND GUIDED TEST GENERATION. THE LAST ROW SUMMARIZES THE RESULTS FROM ALL RUNS.

Program	API calls				Called API methods				Sequences to first call			
	Coll/Iter		Vector/Enum		Coll/Iter		Vector/Enum		Coll/Iter		Vector/Enum	
	Ung..	Guid.	Ung..	Guid.	Ung..	Guid.	Ung..	Guid.	Ung..	Guid.	Ung..	Guid.
antlr	3,068	7,915	6,146	58,438	3	3	11	11	930	510	651	188
bloat	33,363	47,269	0	0	62	62	0	0	686	470	0	0
chart	1,564,161	2,356,751	278	19,761	77	75	5	6	624	236	5,472	720
eclipse	33,334	164,051	29,285	152,103	45	40	20	20	506	348	804	512
fop	12,291	59,220	1,163	22,078	46	46	12	12	1,016	601	1400	260
hsqldb	1,549	50,730	37,479	656,172	12	28	12	14	2,503	275	1,238	105
jython	261,276	236,718	36,474	122,400	105	89	18	18	444	802	1,414	575
lucene	70,884	136,490	66,169	97,585	38	37	12	12	561	537	942	1,861
pmd	16,371	134,074	2,934	450,228	91	93	19	21	1,264	638	6,051	2,167
xalan	22,289	116,225	38,230	106,600	9	9	17	18	940	459	802	376
Avg./Med.	**Improvement: x56.7 / x5.2**				**Improvement: x1.0 / x1.0**				**Reduction: x6.9 / x1.9**			



(a) Effects of guiding test generation using *xalan* and Vector/Enum as an example.



(b) Improvement of guided runs over unguided runs in the number of call sequences required to trigger a bug.

Figure 6. Graphical comparison of guided and unguided test generation.

much faster with guidance than without guidance. The second graph illustrates that the guided run triggers API methods much faster than the unguided run, even though both arrive roughly at the same number of API methods after about 4,500 sequences.

Finally, does guided test generation trigger bugs earlier than unguided test generation? Figure 6b compares the number of generated sequences required to trigger a bug. The graph shows the improvement through guidance for each run where a bug was found both with and without guidance. In most of the runs, bugs are triggered faster with guidance, with up to 109x improvement. For some runs, our heuristic triggers bugs slower, with a slowdown up to 8x. On average, guided runs improve upon unguided runs by 5x.

There is one outlier in Table III: For *jython* and the Coll/-Iter API, guidance decreases the number of API calls and increases the sequences needed to trigger an API method. The reason is that almost all methods (91%) in *jython* are relevant for using the Coll/Iter API, which skews the method priorities. After all, our guidance technique is a heuristic that, even though successful in most cases, cannot guarantee improvements for all programs and APIs.

D. Scalability and Performance

The total execution time of our analysis is the sum of three components: the time for generating tests and executing them, the time for mining execution traces, and the time for checking execution traces. All three scale well to large programs. Randoop's execution time is the product of the number of sequences to generate and some program-specific factor (because Randoop executes code from the program). The performance of the protocol miner scales linearly with the size of the execution traces [5], which is also true for the checker because it builds upon the same infrastructure. On a standard PC, analyzing a program-API pair takes between less than a minute and several minutes, which we consider to be acceptable for an automatic testing tool.

V. DISCUSSION

Our analysis reveals unsafe API usages in classes of a program but does not guarantee that a crash is feasible when using the program as a whole, for example, via its graphical user interface. Nevertheless, all reported bugs are real problems because a programmer can trigger them via the public interface of a class. That is, even if a bug is infeasible in the current program, changes in using the buggy class can make it feasible. Programmers can avoid this risk by ensuring safe API usage.

Currently, the analysis is limited to detecting violations of API protocols. Future work may apply the approach to other kinds of dynamically inferred specifications, such as invariants [23], [24]. Another limitation is that the analysis focuses on bugs that manifest through an exception. While this approach guarantees all bugs to be severe, we might miss more subtle problems that lead to incorrect but not obviously wrong behavior.

VI. Related Work

A. Bug Finding with Mined Specifications

Mined specifications can be used for bug finding by searching for anomalies within the extracted specifications [25]–[28] or by feeding the specifications into program checkers [14], [29]. In contrast to our work, these approaches either require input to run a program, produce false positives, or both.

We present a static bug finding technique based on mined protocols in [29]. It can analyze those parts of a program that are not covered by generated tests but reports false positives and does not address the problem of exercising programs for protocol mining.

B. Test Generation

This work builds upon random test generation in general [1], [3], and the feedback-directed random test generator Randoop [2] in particular. The bugs found by our analysis complement the bugs reported by Randoop based on built-in, generic test oracles. Our approach extends the built-in oracles with inferred protocols that reveal problems Randoop is oblivious of otherwise.

Jaygarl et al. [30] modify Randoop's random method selection by creating parameters before calling a method and by favoring methods with low coverage in the existing tests. Zheng et al. [31] propose to select methods that are likely to affect the method under test, where influence between two methods is estimated based on whether the methods access a common field. These approaches increase the coverage of methods under test, whereas our guidance technique intensifies the API usage of the tested program. In contrast to them, our guidance technique is more general, as it considers both parameters needed to call a method and methods that may influence the state of objects. A technique for regression testing by Jin et al. [32] guides Randoop towards methods that have been modified between two versions.

Test generation can be enhanced by mining call sequence models [33]–[36]. Xie and Notkin [37] propose to combine test generation and specification mining for iteratively enhancing both activities. Dallmeier et al. [38] concretize this idea by generating tests that explore not yet specified sequences of a protocol and by using the result of executing these tests to refine the protocol. The main goal of these approaches is to enhance test generation with specification mining, or vice versa, whereas we combine test generation and specification mining into an automatic bug finding technique.

Xie and Notkin integrate invariant inference with test generation to guide the test generator and to select generated tests that are more likely to expose a bug [39]. In contrast to their work, our approach does not require an existing test suite and reports bugs without false positives.

C. Comparing Passing and Failing Tests

Two kinds of approaches leverage the differences between passing and failing tests. The first group of approaches compares features of passing and failing tests to identify classes and methods that are likely to cause a known failure [40]–[42]. The second group of approaches compares passing and failing tests to generate fixes for a known problem [43], [44]. In contrast to both approaches, our work discovers unknown bugs and leverages test generation instead of relying on existing input, such as manually written tests.

D. Runtime Protocol Checking

The checking step of our analysis is closely related to runtime verification of protocols [7]–[9]. Our checker is conceptually similar to the online checker JavaMOP [9] but works on execution traces and thus avoids some of the challenges faced by JavaMOP, for example, because we know which objects interact during an execution.

VII. Conclusions

This paper shows how to integrate test generation, specification mining, and runtime checking into a fully automatic bug finding technique. We draw several conclusions. First, generated tests are not only useful to reveal bugs, but they can also serve as input to drive other dynamic analyses. This allows our approach to benefit from the precision of dynamic analysis without requiring any input to run a program. Second, leveraging passing and failing tests for mining and checking specifications, respectively, allows for detecting bugs without reporting false positives. Third, focusing the efforts spent during random test generation on those parts of a program relevant for the problem at hand increases the effectiveness of the overall approach, compared to purely random test generation.

Acknowledgments

Thanks to Eric Bodden, Mark Gabel, Patrick Meredith, Albert Noll, Grigore Rosu, and the anonymous reviewers for their valuable comments on this work, as well as to Carlos Pacheco and Michael Ernst for providing Randoop. This work was partially supported by the Swiss National Science Foundation under grant number 200021-134453.

References

[1] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Software Pract Exper*, vol. 34, no. 11, pp. 1025–1050, Sep. 2004.

[2] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE*, 2007, pp. 75–84.

[3] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "ARTOO: adaptive random testing for object-oriented software," in *ICSE*, 2008, pp. 71–80.

[4] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *POPL*, 2002, pp. 4–16.

[5] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *ASE*, 2009, pp. 371–382.

[6] M. Pradel, P. Bichsel, and T. R. Gross, "A framework for the evaluation of specification miners based on finite state machines," in *ICSM*, 2010, pp. 1–10.

[7] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," in *OOPSLA*, 2005, pp. 345–364.

[8] M. C. Martin, V. B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: A program query language," in *OOPSLA*, 2005, pp. 365–383.

[9] F. Chen and G. Rosu, "MOP: An efficient and generic runtime verification framework," in *OOPSLA*, 2007, pp. 569–588.

[10] S. M. Blackburn *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA*, 2006, pp. 169–190.

[11] T. A. Henzinger, R. Jhala, and R. Majumdar, "Permissive interfaces," in *ESEC/FSE*, 2005, pp. 31–40.

[12] M. Gabel and Z. Su, "Online inference and enforcement of temporal properties," in *ICSE*, 2010, pp. 15–24.

[13] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *ASE*, 2009, pp. 307–318.

[14] J. Whaley, M. C. Martin, and M. S. Lam, "Automatic extraction of object-oriented component interfaces," in *ISSTA*, 2002, pp. 218–228.

[15] S. Shoham, E. Yahav, S. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," in *ISSTA*, 2007, pp. 174–184.

[16] M. Gabel and Z. Su, "Javert: Fully automatic mining of general temporal properties from dynamic traces," in *FSE*, 2008, pp. 339–349.

[17] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *ICSE*, 2008, pp. 501–510.

[18] C. Lee, F. Chen, and G. Rosu, "Mining parametric specifications," in *ICSE*, 2011, pp. 591–600.

[19] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer, "On the predictability of random tests for object-oriented software," in *ICST*, 2008, pp. 72–81.

[20] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[21] C. Pacheco, S. K. Lahiri, and T. Ball, "Finding errors in .NET with feedback-directed random testing," in *ISSTA*, 2008, pp. 87–96.

[22] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *CASCON*, 1999, pp. 125–135.

[23] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE T Software Eng*, vol. 27, no. 2, pp. 213–224, 2001.

[24] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *ICSE*, 2002, pp. 291–301.

[25] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," in *ASE*, 2009, pp. 295–306.

[26] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *ICSE*, 2009, pp. 496–506.

[27] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *ESEC/FSE*, 2009, pp. 383–392.

[28] M. Monperrus, M. Bruch, and M. Mezini, "Detecting missing method calls in object-oriented software," in *ECOOP*, 2010, pp. 2–25.

[29] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking API protocol conformance with mined multi-object specifications," in *International Conference on Software Engineering (ICSE)*, 2012.

[30] H. Jaygarl, K.-S. Lu, and C. K. Chang, "GenRed: A tool for generating and reducing object-oriented test cases," in *COMPSAC*, 2010, pp. 127–136.

[31] W. Zheng, Q. Zhang, M. R. Lyu, and T. Xie, "Random unit-test generation with MUT-aware sequence recommendation," in *ASE*, 2010, pp. 293–296.

[32] W. Jin, A. Orso, and T. Xie, "Automated behavioral regression testing," in *ICST*, 2010, pp. 137–146.

[33] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "MSeqGen: Object-oriented unit-test generation via mining source code," in *ESEC/FSE*, 2009, pp. 193–202.

[34] T. Xie and D. Notkin, "Automatic extraction of object-oriented observer abstractions from unit-test executions," in *ICFEM*, 2004, pp. 290–305.

[35] S. Thummalapenta, J. de Halleux, N. Tillmann, and S. Wadsworth, "DyGen: Automatic generation of high-coverage tests via mining gigabytes of dynamic traces," in *TAP*, 2010, pp. 77–93.

[36] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Combined static and dynamic automated test generation," in *ISSTA*, 2011, pp. 353–363.

[37] T. Xie and D. Notkin, "Mutually enhancing test generation and specification inference," in *FATES*, 2003, pp. 60–69.

[38] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, "Generating test cases for specification mining," in *ISSTA*, 2010, pp. 85–96.

[39] T. Xie and D. Notkin, "Tool-assisted unit test selection based on operational violations," in *ASE*, 2003, pp. 40–48.

[40] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *ASE*, 2003, pp. 30–39.

[41] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for Java," in *ECOOP*, 2005, pp. 528–550.

[42] L. Mariani, F. Pastore, and M. Pezzè, "Dynamic analysis for diagnosing integration faults," *IEEE Trans Sw Eng*, vol. 37, no. 4, pp. 486–508, 2011.

[43] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE*, 2009, pp. 363–374.

[44] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *ISSTA*, 2010, pp. 61–72.