Statically Checking API Protocol Conformance with Mined Multi-Object Specifications

Michael Pradel
Department of Computer Science
ETH Zurich, Switzerland

Ciera Jaspan Jonathan Aldrich
Institute for Software Research
Carnegie Mellon University

Thomas R. Gross
Department of Computer Science
ETH Zurich, Switzerland

***Abstract*—Programmers using an API often must follow protocols that specify when it is legal to call particular methods. Several techniques have been proposed to find violations of such protocols based on mined specifications. However, existing techniques either focus on single-object protocols or on particular kinds of bugs, such as missing method calls. There is no practical technique to find multi-object protocol bugs without a priori known specifications. In this paper, we combine a dynamic analysis that infers multi-object protocols and a static checker of API usage constraints into a fully automatic protocol conformance checker. The combined system statically detects illegal uses of an API without human-written specifications. Our approach finds 41 bugs and code smells in mature, real-world Java programs with a true positive rate of 51%. Furthermore, we show that the analysis reveals bugs not found by state of the art approaches.**

***Keywords*-Typestate; Static analysis; Specification mining**

I. Introduction

Many application programming interfaces (APIs) impose constraints on clients, for example by prescribing particular orders of method calls. Violating such constraints can lead to unexpected behavior, incorrect results, and even program crashes. Unfortunately, most API usage constraints are not formally specified, making it difficult for programmers to learn the constraints and challenging for program analyses to find violations.

As a motivating example, consider Java source code we found in a commonly used benchmark program (Figure 1). The program iterates over a list using an iterator and attempts to remove elements from the list. This usage of Java's Collection API leads to a runtime exception: Programmers must not modify a collection while iterating over it. As the example shows, API usage constraints are often subtle, and violations of them can be hard to detect.

Automatic bug finding techniques based on mined specifications [1]–[3] are a promising approach to find API usage bugs with little effort. This paper focuses on specifications of legal method call sequences through finite state machines (FSMs), called *API usage protocols* [4]–[11]. While existing work focuses on checking programs against mined protocols for a single object [12], [13], this paper presents an automatic bug finding technique based on mined *multi-object* protocols.

Considering multiple interacting objects allows our approach to specify and check constraints that are not expressible in single-object protocols. Consider the following examples from the Java standard library:

- While iterating over a `Collection` with an `Iterator`, one must not modify the `Collection`. Figure 1 violates this constraint.
- To retrieve the entries of a `ZipFile`, one must obtain an `Enumeration` over all entries, iterate through it, and finally close the `ZipFile`.
- Threads can synchronize via `Condition`s and `ReentrantLock`s. Before calling the condition's waiting or signaling methods, one must acquire the lock. Eventually, the lock must be released.
- When wrapping resource-like objects, such as a `FileReader` wrapped into a `BufferedReader`, closing the wrapping resource is sufficient, as it will implicitly close the wrapped resource.

All examples are taken from the protocols mined in the evaluation of this work. Despite this variety, we henceforth focus on collections and iterators to simplify the presentation.

Checking programs against multi-object protocols can reveal more bugs than a single-object approach. For example, our approach detects the concurrent modification in Figure 1. This bug cannot be found with a protocol specifying only `Iterator`s or `Collection`s; it is only expressible with a protocol specifying both together.

In addition to finding more bugs, a multi-object approach avoids false positives caused by the limited scope of single-object protocols. For example, calls to `Iterator.next()` should be preceded by `Iterator.hasNext()`, `Collection.size()`, or `Collection.isEmpty()` to avoid requesting a non-existing element. A single-object `Iterator` protocol specifying the first possibility leads to false positives when the other two possibilities are used. In contrast, a multi-object protocol can specify all possibilities.

The bug detection technique presented in this paper is the result of combining two analyses: a dynamic miner of multi-object protocols [8], [10] and a static checker of API usage constraints, Fusion [15]. Figure 2 provides an overview of our approach. The dynamic specification miner runs training programs to infer specifications from program traces. These

```
LinkedList pinConnections = ...
Iterator i = pinConnections.iterator();
while (i.hasNext()) {
  PinLink curr = (PinLink) i.next();
  if (...) {
    pinConnections.remove(curr);
  }
}
```

Figure 1. Illegal usage of a collection and an iterator found in the DaCapo benchmarks [14].

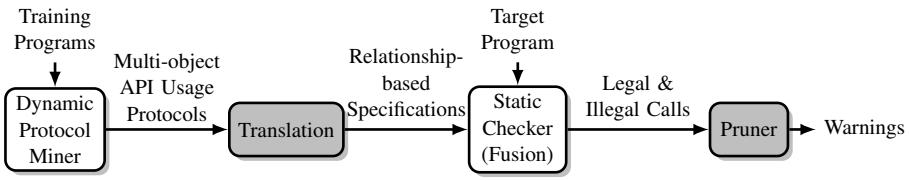


Figure 2. Overview. The components to translate mined protocols into relationship-based specifications and to prune warnings are the primary technical contributions of this paper.

API usage protocols are translated into *relationship-based specifications* suitable for the static checker. The static checker checks the code of a *target program* against the specifications and classifies API method calls as potentially legal or illegal. Some of the potentially illegal calls may be due to incomplete protocols, so a pruner uses heuristics to remove results that are likely false positives. The final set of *warnings* is then presented to the user for manual inspection.

While most related approaches combine mining and checking into a single analysis [16]–[19], we build upon two special purpose analyses. The main reason is that an analysis made for mining specifications is not necessarily the best choice for checking programs. In particular, all-in-one approaches typically do not guarantee that each reported deviation from a common pattern can actually occur at runtime. Our work avoids this source of false positives by using a complete static checker.

The key difficulty in blending the protocol miner with the static checker is to connect the two formalisms used for encoding specifications. While the miner produces FSMs labeled with method calls, the checker requires relationship-based annotations of API methods. We present a translation between these two formalisms, which is the foundation of the presented bug finding technique.

Our approach finds bugs without any a priori known specifications and without human refinement or selection of the mined protocols. The price for this automation is that the combined analysis is neither sound nor complete: Mined protocols can be incomplete, introducing false positives, and incorrect, introducing false negatives. Our results show both problems to be manageable in practice.

To evaluate our work, we analyze the programs of the DaCapo benchmark suite [14], a 1.6 MLOC collection of open-source Java programs. Our system detects 41 issues (26 bugs and 15 suspicious code smells) with a true positive rate of 51%. This precision is higher than that of comparable state of the art approaches (29% in [18], 38% in [16], 38% in [17]), and we show by direct comparison that our approach reveals bugs that these approaches cannot find. To assess which bugs our approach misses, we randomly seed bugs into the programs, of which the system finds 70%. We attribute the remaining 30% to four types of false negatives. Furthermore, we report on a sensitivity analysis of the pruning heuristics. Finally, we find that multi-object protocols are relevant in practice: 61% of the mined protocols are multi-object protocols, and 44% of the issues our system detects are found only with a multi-object protocol.

In summary, this paper contributes the following:

1) A fully automatic technique to find multi-object protocol bugs. It requires neither tests to run the target program nor formal specifications.
2) A translation between FSM-based mined specifications and a relationship-based API specification formalism, which enables the first contribution.
3) Pruning techniques to improve precision.
4) Empirical evidence that multi-object protocols are relevant in practice.
5) Empirical evidence that combining a heuristic, dynamic specification miner with a precise, static checker can reveal serious problems in well-tested programs with a good true positive rate.

II. Background

Our approach uses a dynamic specification miner and a static checker to respectively produce and check specifications of multi-object protocols. Both analyses divide a protocol into two parts. The first part determines the applicability of a protocol, and the second part describes the constraints imposed by the protocol.

A. Specification Mining and API Usage Protocols

We use a dynamic specification miner that extracts API usage protocols from training programs [8], [10]. Any existing API client can serve as a training program. A protocol consists of a deterministic finite state machine and a set of typed protocol parameters. States represent the common state of multiple objects or, for single-object protocols, the state of a single object. Transitions are labeled with method signatures that are annotated with protocol parameters naming the receiver, the method parameters and the return value.

The protocol miner analyzes execution traces, that is, sequences of method calls and returns. At first, the miner extracts subtraces that contain calls to a particular object o, to objects passed to o as an argument, and to objects returned by o. Then, all subtraces for a set of receiver types are summarized into a protocol in such a way that each subtrace is accepted by the protocol.

The mined protocols 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. The miner constructs protocols in such a way that the set of parameters bound at a state is unambiguous. States at which all parameters are bound are liable states; all other states are setup states.

Figure 3 shows an API usage protocol describing the interplay of a collection c and an iterator i. The protocol specifies how to use the iterator (call `hasNext()` before `next()`) and that updating the collection invalidates the iterator. Calling the method `iterator()` on c, which returns i, establishes the interaction between these two objects. Therefore, the later states (3, 4, and 5) are all liable states. The table in Figure 3 gives the set of bound protocol parameters for each state.

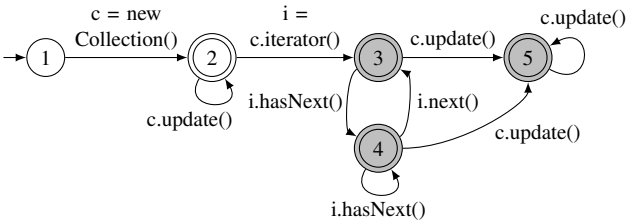
B. Relationship-Based Static Checking

This work uses Fusion [15], a relationship-based, static analysis, to check API clients against specifications over multiple objects. Fusion encodes usage constraints based on *relationships* between objects. A relationship is a user-defined, uninterpreted predicate across objects. For example, a binary relationship $r_{contains}(List, Element)$ can express that an object of type $List$ contains another object of type $Element$, such as $r_{contains}(list, element)$. If a relationship predicate is true, we say that the objects are in the relationship. Likewise, to say that the objects are not in a relationship means that the relationship predicate evaluates to false.

In Fusion, API methods are specified with requirements and effects. A *requirement* is a logical proposition over relationship predicates that describes a precondition on a method. For example, `list.remove(element)` may require that $r_{contains}(list, element)$ holds. An *effect* is a postcondition that describes how the truth of relationship predicates changes after calling a method. For instance, `list.remove(element)` may have the effect to remove $(list, element)$ from $r_{contains}$. Both requirements and effects are guarded by a *trigger*, a logical proposition that describes when the requirement or effect applies. A complete specification of a protocol in Fusion is a set of relationships, a set of effects, and a set of requirements on the relevant methods.

Based on specifications of API methods, Fusion performs an intraprocedural analysis of API clients to check whether they respect the usage constraints. For each call to a method with a specification, the analysis checks whether all triggered requirements are fulfilled and applies all triggered effects. We use a complete variant of Fusion, which guarantees that any bug found will actually occur dynamically.[1]

[1]Fusion can only make this guarantee to the degree that it is given precise aliasing information; see Section V-C for how this affects results in practice.



State	1	2	3	4	5
Bound protocol parameters	$\{\}$	$\{c\}$	$\{c, i\}$	$\{c, i\}$	$\{c, i\}$

Figure 3. Protocol describing how to use a collection and an iterator. The call `c.update()` summarizes calls that modify the collection, for example, `c.add()` or `c.remove()`. Liable states have a gray background.

```
1  Collection c = ...
2  Iterator i = c.iterator();
3  if (i.hasNext())
4     System.out.println(i.next());
5  /* current state unknown */
6  c.update();        // legal -- invalidates iterator
7  if (i.hasNext())   // bug: iterator not valid anymore
8     System.out.println(i.next());
```

Figure 4. The state in line 5 cannot be determined because different paths lead to it.

III. From Mined Protocols to Checkable Specifications

This section presents a translation of FSM-based protocols into relationship-based constraints suitable for checking programs. A more detailed and formal description is given in [20]. The main goal of the translation is to detect protocol violations while reporting few false positives. That is, the generated specifications should cause a warning only if the analyzed source code actually violates a protocol. We use Figure 3 as a running example throughout this section.

A. Challenges

We must deal with two main challenges, the first being common to all static analyses and the second being specific to multi-object checking:

1) Limited Static Knowledge: Static analysis inherently lacks precise knowledge about which program path is taken and as a result may miss bugs or report false positives. For example, Figure 4 shows a piece of source code where static analysis cannot determine the protocol state in line 5. After the call to `hasNext`, the protocol is known to be at state 4. The branch creates two paths, one path on which `next()` is called, leading back to state 3, and another path on which we stay at state 4. Because static analysis cannot determine which path is taken, the state in line 5 is unknown. Our combined analysis should nevertheless find the call in line 6 to be legal and detect the protocol violation in line 7.

2) Object Interactions: Checking multi-object protocols is challenging, because calling a method on one object can

influence the state of other objects. For example, the call to `update()` in line 6 does not refer to the iterator `i`, but directly affects its state. This dependence is implicit in mined multi-object protocols, where a state is the common state of all involved objects.

B. Naive Approach: States as Relationships

An obvious but insufficient way to map protocols to relationships is to have a unary relationship per state and to track each object's state by updating these *state relationships*. For this naive approach, we create an effect for each transition leading from state x to state y: This effect removes the object from r_x and adds it to r_y. To check whether calling a method is legal, we require that the receiver object is in a relationship representing a state where the method is enabled. For example, calling method `i=c.iterator()` requires that $r_2(c)$; the call has the effect of removing c from r_2 and of adding c and i to r_3.

Unfortunately, the approach does not deal well with the challenges illustrated in Figure 4. The lack of static knowledge whether line 4 is executed leads to a loss of knowledge in line 5: Because both $r_3(i)$ and $r_4(i)$ are possible, but not both, Fusion concludes to know nothing about i. Thus, the analysis cannot detect that line 7 is illegal. Furthermore, the approach is oblivious of the relationship between c and i, so that in line 6, i's state is not changed. This limitation is another reason why the naive approach cannot find the bug in line 7.

C. Revised Approach: Triple Bookkeeping

Given a set of protocols, how can we generate relationship-based specifications in a way that addresses the above challenges? We extend the approach described in Section III-B by using relationships to reason about *three* aspects of objects with respect to a protocol. We call the idea underlying this revised approach the *triple bookkeeping* principle. First, we keep track of whether calling a method is allowed with a *method relationship*. This is a unary relationship for a method of the protocol. If an object is in a method relationship, it means that one can legally call the corresponding method on this object. The protocol in Figure 3 has four method relationships (the types in parentheses specify which objects can be in a relationship):

$$r_{next}(Iterator)$$
$$r_{hasNext}(Iterator)$$
$$r_{update}(Collection)$$
$$r_{iterator}(Collection)$$

There is no method relationship for constructor calls because a newly created object cannot be in any relationship before the constructor call.

Second, we keep track of the current state of an object or a set of objects with *state relationship*s. The semantics of a state relationship is that if an object or a set of objects is in this relationship, we know these objects to be at the corresponding state. In contrast to the naive approach, the state relationship here is over all objects that are bound at that state. The states of the protocol in Figure 3 translate into four state relationships:

$$r_2(Collection)$$
$$r_3(Collection, Iterator)$$
$$r_4(Collection, Iterator)$$
$$r_5(Collection, Iterator)$$

There is no state relationship for the initial state because no variables are bound at this state.

Third, we keep track of which objects interact as part of a multi-object protocol. For this purpose, we create a *protocol relationship* over all protocol parameters. The semantics of the protocol relationship is that if a set of objects is in this relationship, we know these objects to interact with each other in the protocol. For example, we create a binary relationship $r_{prot}(Collection, Iterator)$ for the two-object protocol in Figure 3.

The three kinds of relationships serve as data structures for the static analysis. The analysis uses them to check whether method calls are allowed and to represent the effects of method calls.

Checking Requirements for Method Calls: To check whether calling a method is legal, we require that the receiver is in the corresponding method relationship. For example, to call `i.next()`, we require that $r_{next}(i)$ is true. The requirements imposed by a protocol are only valid if the current state is a liable state. Therefore, we guard requirements with the protocol relationship, ensuring that the involved objects interact in the protocol and that the protocol is in a liable state. For example, the requirement on calling `i.next()` is guarded by $r_{prot}(c, i)$. This can be thought of as a logical precondition of the form:

$$\forall c \,.\, (r_{prot}(c, i) \implies r_{next}(i))$$

Making Effects: Calling a method can have effects on the current state, on the currently enabled and disabled methods, and on the binding of objects to protocols. We create specifications that reflect these effects by adapting the corresponding relationships. For example, calling `i = c.iterator()` as shown in Figure 3 influences the state maintained by the analysis as shown in Table I.

Knowing that a particular method is called may not be sufficient to determine which effects to apply and which requirements to check because a single method can label multiple transitions. For example, calling `c.update()` at state 2 (leading again to state 2) has different effects than calling the same method at state 3 (leading to state 5). We handle this problem by guarding effects with triggers on state relationships. In the example for `c.update()`, the translation creates two constraints in Fusion, one triggered by $r_2(c)$ and the other triggered by $r_3(c, i)$.

Table I
EXAMPLE EFFECTS FROM CALLING `ITERATOR()`.

Protocol effects of calling `i = c.iterator()`	**Effects on relationships**
Move to state 3.	Set $r_3(c,i)$ to true; set $r_2(c)$, $r_4(c,i)$, and $r_5(c,i)$ to false.
Enable methods `i.hasNext()` and `c.update()`; disable methods `i.next()` and `c.iterator()`.	Set $r_{hasNext}(i)$ and $r_{update}(c)$ to true; set $r_{next}(i)$ and $r_{iterator}(c)$ to false.
Establish the interaction between `c` and `i`.	Set $r_{prot}(c,i)$ to true.

Our revised approach meets the challenges mentioned in Section III-A. We deal with limited static knowledge by maintaining both the current state and the currently enabled and disabled methods. This approach allows the analysis to recover knowledge that has been lost when merging paths. Even when the current state is unknown, the analysis may still know about currently enabled methods and continue the analysis based on this information. The translation addresses the problem of interacting objects by representing states at which n objects are bound by n-ary state relationships. Furthermore, we represent the interaction of objects through the protocol relationship. By using the protocol relationship as a trigger, the analysis can apply effects on all involved objects, even if a method references only a subset of them.

Translating the protocol in Figure 3 gives constraints that detect the bug in Figure 4. Before checking line 6, the analysis knows the relationships $r_{update}(c)$ and $r_{prot}(c,i)$. That is, calling `c.update()` is legal and c is known to interact with i. The effects of the call lead to the following relationships after line 6: $r_5(c)$, $r_5(i)$, and $\neg r_{hasNext}(i)$. Thus, the precondition for calling `i.hasNext()` in line 7 does not hold and Fusion reports a warning.

IV. ENSURING PRECISION

A major concern of a bug finding system is to report relevant warnings to a programmer without overwhelming her with false positives. Our approach must deal with two sources of false positives. The first source are the mined protocols, as these will only describe a subset of all legal method call sequences. The reason for this incompleteness is that the specification miner is limited to the call sequences that it observes during the execution of training programs. Although one can increase the completeness of protocols by analyzing more executions [10], it is difficult to guarantee that a mined protocol contains all legal sequences.

The second source of false positives is imprecision in the static analysis. Analyzing object-oriented languages imposes several challenges, such as aliasing and the need to deal with all possible execution paths. A static analysis that lists all possible protocol violations is likely to emit warnings that cannot happen in a real run of the program.

A. Dealing with Incomplete Protocols

We present three automatic pruning techniques to remove warnings caused by incomplete protocols. They are independent of the API, the mining technique, and the checker, making them generally applicable. The techniques are based on the assumption that most parts of the training and target programs use APIs legally. Incomplete protocols often stand out because they occur rarely during mining or because the static analysis detects many violations of these protocols during checking. We use these observations to identify incomplete protocols and remove the warnings they cause.

1) Pruning by Number of Confirmations: The specification miner counts how often each protocol occurs in the training programs. We call this value the *number of confirmations* of the protocol. Intuitively, a higher number of confirmations indicates a higher confidence in the protocol as there are more concrete examples supporting the protocol. To avoid false positives due to low confidence protocols, we prune all warnings from protocols with a number of confirmations below a configurable threshold.

2) Pruning by Protocol Error Rate: Another useful metric is how many warnings the protocol causes while checking programs. Incomplete protocols often result in many warnings because previously unseen, legal API usages violate the mined protocol. We compute the *protocol error rate* by dividing the number of method calls that are illegal according to a protocol by the overall number of method calls checked for this protocol. To avoid false positives by incomplete protocols that cause unusually many warnings, we prune all warnings caused by protocols with a protocol error rate above a configurable threshold.

3) Pruning by Method Error Rate: The third metric is a variation of the protocol error rate. This metric addresses protocols with unusually many violations caused by a particular method specified in the protocol. Instead of computing the error rate for the entire protocol, we compute it for each method that appears in a protocol. The *method error rate* is the number of calls to a method that are found to be illegal divided by the overall number of checked calls to this method. If a protocol contains a method with a method error rate above a configurable threshold, we prune all warnings caused by this protocol.

We perform a sensitivity analysis to determine reasonable pruning thresholds for these metrics (Section V-G).

B. Dealing with Imprecisions of the Static Checker

To handle imprecisions due to static analysis, we use the complete variant of Fusion, which guarantees to issue a warning only when the analysis knows for certain that the constraint is broken. Fusion is able to do this because it supports three-valued logic; all predicates evaluate to either true, false, or unknown. The complete variant only issues warnings when the trigger predicate is true and the requires

predicate is false. Any loss of precision that leads to an unknown predicate is ignored by this variant, thus ensuring that there are no false positives.

The complete variant is guaranteed to give no false positives only if it has precise aliasing information. As real alias analyses are imprecise, there are still false positives from this variant due to incorrect aliasing information. To reduce these as much as possible, we use a heuristic, flow-based points-to analysis. This analysis reduces false positives by optimistically assuming that methods return a fresh label that is not aliased by anything else and by presuming that fields are not written to outside of the local context.

V. Evaluation

To evaluate the effectiveness of our approach, we use it to check the API usage of twelve real-world Java programs. In summary, we have the following findings:

1) *Does the approach detect relevant issues in mature and well-tested programs?* We find 41 issues (26 bugs and 15 code smells) in the programs of the DaCapo benchmark suite [14]. Detecting relevant problems in these programs is challenging, since most of them are in production use since several years and have been analyzed by various researchers. (Section V-B)
2) *How precise are the reported warnings?* The true positive rate of the reported warnings is 51%. (Section V-C)
3) *How many protocols are multi-object protocols, and how many warnings come from these protocols in practice?* 61% of the mined protocols are multi-object protocols, and 44% of the true positives are from multi-object protocols. (Section V-D)
4) *Do we find bugs missed by existing approaches?* We show through direct comparison that our approach finds bugs missed by existing multi-object bug finding techniques [16], [18], which those approaches cannot find due to inherent limitations. (Section V-E)
5) *Which bugs do we miss?* We randomly seed bugs into the programs and find that our system detects 70% of them. We describe four categories of defects that the system currently misses. (Section V-F)
6) *What are the best filtering thresholds in practice?* We perform a sensitivity analysis and find thresholds for pruning warnings. (Section V-G)
7) *Do the issues found justify the costs of the system?* As the system is fully automatic, the only costs are finding the initial training programs and evaluating the warnings to determine which are false positives. In practice, we found that most false positives were localized and fast to detect manually. (Section V-H)

A complete list of all detected issues is available in a companion report [20].

A. Experimental Setup and Warning Classification

As target programs we use the twelve Java programs of the DaCapo 9.12 benchmark suite (Table II). In total, these programs sum up to 1.6 million non-blank, non-comment lines of Java code.

As training programs for the protocol miner, we use a set of real-world Java programs used in previous work [10]. The training programs are disjoint from the target programs. We gather execution traces by running unit tests and benchmarks of the program, and by manually executing programs (see [10] for more details). In total, we gather 47 million runtime events, each being a method call or a method return. We configure the protocol miner to extract protocols for the entire `java.*` and `javax.*` APIs, giving 223 protocols. We do not filter or change these 223 protocols manually and instead evaluate the combined approach with protocols that are inferred in a *fully automatic way*.

We manually inspect reported warnings to classify them into three categories (similar to [18], [21]):

- *Bug*. An error that can lead to a program crash or unexpected behavior. We only consider a problem a bug if a programmer can trigger it via the public interface of the buggy class and if the Javadoc documentation of this class does not prevent programmers from triggering the bug.
- *Code smell*. A program property that indicates that something may go wrong [22]. We include in this category code that should be changed to improve maintainability or performance. We also consider problems in test classes as code smells if they fulfill the requirements for a bug, but if we do neither find nor expect any use of the test class by other classes.
- *False positive*. All remaining warnings.

We say *issue* to a warning that is either a bug or a code smell.

B. Examples of Issues Found

In total, our approach detects 41 issues in the analyzed programs. Figures 1, 5a, 5b and 5c are four representative examples of warnings reported by our system (slightly edited for conciseness). The method calls found to violate a protocol are highlighted.

The first example (Figure 1 in Section I) is clearly a bug. The programmer iterates over a list to find and remove a particular element. Unfortunately, this code is at risk of throwing a `ConcurrentModificationException` because a collection must not be modified while iterating over it. Interestingly, testing may not find this bug because the code only throws an exception if the removed element is not the last element in the list.

The second example (Figure 5a) is also a clear bug. The programmer illegally uses an iterator by assuming that `next()` returns `null` after the last element. As iterators

```
Map comparators = ...
Iterator i = comparators.values().iterator();
for (Comparator c = (Comparator) i.next();
     c != null;
     c = (Comparator) i.next()) { ... }
```

(a) Bug: Illegal call to `next()`.

```
BufferedReader in = null;
try {
  in = new BufferedReader(...);
  ...
  in.close();
} finally {
  if (in != null) {
    try { in.close(); }
    catch (IOException e) { ... }
  }
}
```

(b) Code smell: Duplicate call to `close()`.

```
LinkedList shingleBuf = ...
if (!shingleBuf.isEmpty()) {
  Token firstShingle = (Token) shingleBuf.get(0);
}
```

(c) False positive caused by an incomplete protocol.

Figure 5. Examples of warnings found. The method calls that violate a protocol are highlighted.

instead throw an exception if `next()` is called after reaching the last element, this code results in an exception (unless the map contains `null` as a value, which is not the case here).

We classify the third example (Figure 5b) as a code smell. The problem is that `close()` is called twice because the `finally` block is always executed, even when no exception is thrown. Closing an already closed stream has no effect, except for the cost of an additional method call. A simple way to enhance the readability of this source code without changing its semantics is to remove the first call to `close()`.

The fourth example (Figure 5c) is a false positive caused by an incomplete protocol. The mined protocol for `LinkedList` does not allow `get()` directly after `isEmpty()`, which is a legal usage of this class. Hence, this "protocol violation" is innocuous and not relevant to a programmer. As for many of the false positives, one can easily identify this warning as false, even without knowing the violated protocol.

While many of the detected issues are related to iterators, our approach applies to a larger range of problems. In fact, 200 of the 223 protocols are unrelated to iterators. We find many iterator-related issues because many calls that can violate a protocol relate to iterators: A recent study of various Java programs [23] shows that at least 51% of all protocol-related calls to the `java.*` and `javax.*` APIs are iterator-related. However, even among the iterator-related issues that our system finds, there are three distinct kinds of problems. First, we find code that does not respect collection boundaries by calling `Iterator.next()` without knowing whether a next element exists (Figure 5a). Second, we find code that fails to prepare an object for a call by invoking `Iterator.remove()` without a preceding `Iterator.next()`. Finally, we find code that calls a method at a state where the call is not allowed by modifying a collection during iteration (Figure 1).

Table II
WARNINGS REPORTED FOR EACH TARGET PROGRAM. MO ARE ISSUES FOUND ONLY WITH MULTI-OBJECT PROTOCOLS.

Program	LOC	Warnings						
		Before pruning	After pruning					
			Total	Bugs		Code smells		True positives
				All	MO	All	MO	
avrora	69,393	45	13	9	8	0	0	69%
batik	186,460	87	1	0	0	0	0	0%
daytrader	12,325	8	0	0	0	0	0	—
eclipse	289,641	188	15	2	2	1	1	20%
fop	102,909	127	13	8	2	1	0	69%
h2	120,821	100	1	0	0	0	0	0%
jython	245,016	70	7	2	1	1	0	43%
lucene	124,105	114	13	3	1	3	1	46%
pmd	60,062	61	15	2	0	8	2	67%
sunflow	21,970	17	0	0	0	0	0	—
tomcat	161,131	165	2	0	0	0	0	0%
xalan	172,300	11	1	0	0	1	0	100%
Total	1,566,133	993	**81**	**26**	**14**	**15**	**4**	**51%**

C. Precision of Reported Warnings

To make our approach applicable in practice, it is critical to provide a reasonable percentage of relevant warnings among all warnings that our system reports. We evaluate the precision of the reported warnings, that is, how many true and false positives they contain, by manually inspecting warnings and by classifying them into bugs, code smells, and false positives. In total, our system reports 81 warnings for the twelve programs. This number allows us to manually inspect all reported warnings and accurately assess how precise our analysis is for these target programs.

Table II summarizes our results. In total, the system detects 26 bugs and 15 code smells, giving a combined true positive rate of 51%. These results show that our analysis is able to pinpoint a serious number of issues in well-tested, deployed programs, while reporting only one false positive for each true positive on average.

To better understand the imprecisions of our approach, we further analyze the false positives that are reported. The 40 false positives fall into three categories:

- 30 false positives (37% of all warnings) are caused by incomplete protocols. Mined protocols are inherently at risk to be incomplete; further progress in specification mining may reduce this number in the future.
- Two false positives (2%) are from imprecisions in the static checker, more precisely, because the heuristic points-to analysis does not detect some intricate aliasing patterns. Again, future improvements of our points-to analysis may reduce this number.
- The final eight false positives (10%) are because of special program semantics that change a protocol in a

way that cannot be captured by our training programs. For example, this occurs when programs assign special semantics to widely used interfaces, such as jython, which contains a custom iterator class that provides an alternative to calling `hasNext()`. Since our system is unaware of the specialized jython semantics but recognizes the class as an iterator, it produces warnings when jython uses the alternative to `hasNext()`.

A complete static checker is crucial to ensure the precision of our approach. The sound variant of Fusion reports significantly more (potential) protocol violations and leads to a higher false positive rate. For example, analyzing avrora with the sound analysis gives 1,805 warnings, compared to 45 from the complete analysis.

D. Multi-Object Bugs

How useful is it to consider multi-object protocols in addition to single-object protocols? 135 of the 223 mined protocols (61%) specify more than one object. The bugs detected by our approach show a similar picture. Table II lists how many of the detected bugs and code smells are found only with multi-object protocols (columns "MO"). In total, 18 of 41 issues (44%) only become apparent due to a multi-object protocol. We conclude from these results that considering multiple objects is an important advantage over single-object approaches [12], [13], [24].

E. Comparison to Existing Work

We directly compare our approach to recent work on anomaly detection [16], [18] and to FindBugs [25], a widely used static bug checker. While not working with protocols, these approaches are the closest existing ones: They also analyze multi-object interactions and find bugs in the use of the Java API. To compare to GrouMiner [16] and Tikanga [18], we run our analysis on their respective target programs and compare the bugs found. We focus on bugs because [16] uses a slightly different classification for non-bug warnings.

Our analysis identifies six bugs that GrouMiner misses. Likewise, GrouMiner finds four bugs that our analysis misses. There are no bugs found by both approaches. Compared to Tikanga, our analysis finds nine bugs that Tikanga misses and misses two bugs found by Tikanga. Three bugs are found by both analyses.

We identify two main reasons for these differences. First, our approach reveals a class of bugs missed by both existing techniques. While [16] and [18] only find missing calls, our analysis also reports calls that are illegal at a particular state. For example, a bug missed by GrouMiner but found by our analysis is a concurrent modification similar to Figure 1. To the best of our knowledge, no existing technique based on mined specifications can find bugs of this kind. Second, each technique is limited by the mined specifications. Each of the three approaches misses some bugs because the "right" specification to reveal the bug is missing. To address this limitation, one can mine larger sets of programs [10] or automatically enrich mined specifications [13].

To compare our analysis to FindBugs, we run FindBugs on our target programs. Although FindBugs finds various problems, it does not find any of the bugs revealed by our analysis. The reason is that FindBugs is based on a predefined list of bug patterns, including several API usage bugs—none of which covers the bugs our technique finds.

In summary, the comparison with existing work not only shows that our approach finds bugs missed by state of the art approaches, but also confirms an observation by Bessey et al.: "The set of bugs found by tool A is rarely a superset of another tool B." [26].

F. Which Bugs Do We Miss?

Giving an exact answer to this question is nearly impossible for any real-world program, because the set of all bugs in such a program is unknown. We try to give an approximate answer by randomly seeding bugs into the programs from Table II and by trying to find these bugs with our system.

A programmer can violate a protocol by omitting a call or by calling a method in a situation where the call is not allowed. We consider these two cases and insert bugs by adding and removing calls. To seed a single bug, we use the following procedure. Let C be the set of all method calls that match a transition of at least one protocol.

1) Randomly decide between adding a new call and removing an existing call.
2) a) To add a call, randomly select a method m containing a call $c \in C$. Randomly select a call c' from the union of the alphabets of protocols that c is checked against. Randomly select a statement of m and try to insert c' after this statement. If the insertion is syntactically impossible (for example, if there is no receiver for c'), repeat Step 2a.
   b) To remove a call, randomly select a method m containing a call $c \in C$ and remove c from m.
3) If the modification does not result in a bug according to the definition from Section V-A, go back to Step 2.

We applied this procedure 50 times, giving 50 known protocol violation bugs. Our system detects 35 of them (70%). This rate gives us some confidence in the effectiveness of our approach in finding bugs. The 15 seeded bugs that the system misses illustrate some limitations of our work:

- The system misses five bugs because a protocol allows an illegal sequence of method calls. As we mine protocols from real-world programs, we cannot guarantee that all specified behavior is legal.
- The system misses four bugs because the static checker conservatively assumes all objects to be in an unknown state at method entry. That is, if the first call on an object is illegal, the static checker might miss it.

- The system misses two bugs where an illegal call occurs only during the first pass through a loop. While the current implementation of the static checker misses those problems, this limitation can be addressed by checking the first pass through a loop independently from the loop's fixpoint.
- The system misses four bugs that go beyond the expressiveness of the inferred protocols. For instance, illegally calling `Iterator.next()` after `Iterator.hasNext()` has returned `false` does not violate a protocol requiring that `next()` is preceded by `hasNext()`. One can address this limitation by extending protocol transitions with invariants [27]. The existing static checker can handle these more expressive protocols.

G. Sensitivity Analysis

As described in Section IV-A, we prune warnings from protocols with few confirmations or a high error rate. Setting reasonable thresholds for the pruning metrics is critical to reduce false positives without removing true positives. To find reasonable thresholds, we perform a sensitivity analysis by manually classifying a random sample of warnings and by evaluating how different thresholds affect these warnings. We inspected 200 warnings, of which 50 were chosen randomly from all warnings that our system reports without any pruning. The other 150 were chosen by starting to prune with strict thresholds that remove many warnings, and by gradually considering more and more warnings. That is, our sample includes a fair number of warnings representative for the entire space of warnings, and a denser sample of warnings in which we expect to find most true positives.

The results of evaluating the warnings that our system reports with different thresholds are shown in Figure 6. Each graph shows, for different thresholds, the number of reported warnings along with the number of issues they contain. The respective other thresholds are kept at their extreme values (zero confirmations and 100% maximum error rates) to measure the effect of each pruning metric separately. All three thresholds reach a point where no additional issues are found by including further warnings. We chose these points as the default configuration for our system. Although we cannot guarantee that no relevant warning is pruned with these thresholds (without inspecting all 993 warnings), we consider the selected thresholds to be reasonable for the analyzed programs. Programs with a different level of maturity may require different thresholds.

H. Costs of the System

Our system has a true positive rate of 51%, which raises the question: Is this high enough to justify using the system? To answer this, we consider the three primary costs of using the system:

- *The cost of finding and running training programs for mining protocols.* The most costly aspect in terms of human time is finding other programs that use the same API, building them, and producing traces of these programs. This cost would also be higher for APIs that are not as common as the Java Standard Library. However, this cost can be amortized over all future uses of the system.
- *The cost of running the static analysis.* The two factors that increase the runtime of the static checker are the size of the target program and the number of constraints to check. Checking 1.6 MLOC with 223 protocols requires a standard PC to run the analysis overnight. However, as the analysis is intraprocedural, it can be run modularly at a method-level, and it scales linearly when the time to analyze a method is bounded.
- *The cost of investigating warnings.* The programmer must investigate each warning to determine whether it is a true positive or not. In our experience, true positive warnings and false positives due to incomplete protocols are generally fast to check. The warnings that took us the most time were the false positives due to special program semantics (10%); these warnings required some knowledge of the analyzed program. However, we expect that a programmer more familiar with the code would classify these warnings much faster.

VI. Related Work

This work relates to two strands of research: bug finding via anomaly detection and static protocol checking. Our approach is unique in combining these approaches for multi-object specifications.

A. Bug Finding via Anomaly Detection

There are various approaches to mine common usage patterns and detect anomalies. In contrast to the techniques that we discuss below, our approach builds upon two independent program analyses for mining and for checking.

Wasylkowski et al. [18] statically extract temporal patterns from programs and report calls that miss one or more preceding calls that establish a certain state. Nguyen et al. [16] propose a static, graph-based analysis that mines usage patterns consisting of method calls and field accesses. Their analysis reports an anomaly if a method's graph is similar to a commonly found graph but misses a particular method call or field access. Our approach finds bugs not detectable by these approaches (Section V-E).

Several approaches use rules saying that calling a set of methods implies calling another method [3], [17], [19]. These approaches ignore the order of calls, a limitation addressed by [28], which however only analyses exception handling code. Other approaches consider pairs of methods

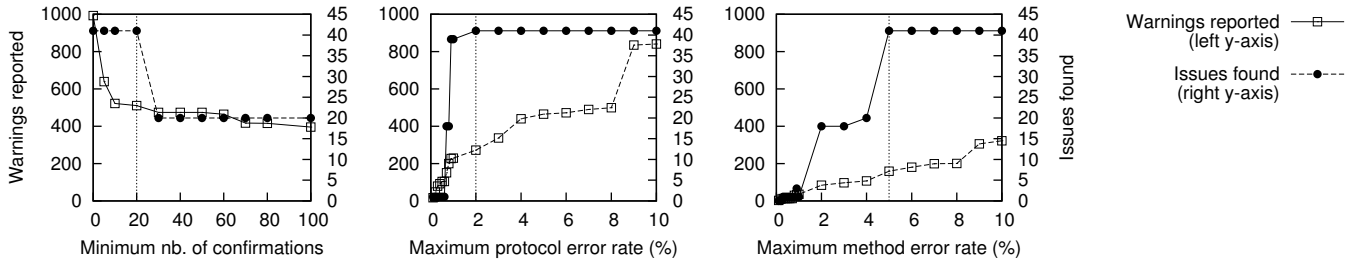


Figure 6. Analysis of the sensitivity to different thresholds for pruning warnings. The vertical lines indicate the thresholds we recommend as default values and that we use for the evaluation.

that are called consecutively [29], program-dependent invariants [2], [30], and properties of method parameters and return values, such as to be non-null or to convey ownership of a resource [31], [32]. Our work differs in the kind of specification used and in the kind of bugs found. Gabel et al. [24] dynamically infer and check method call ordering constraints; their approach is limited to single objects. Engler et al. [1] propose a static analysis that infers programmer "beliefs", such as that two methods are always paired with each other, and finds violations of them. The approach requires a custom checker for each kind of belief.

In [33], we present a dynamic bug finding technique based on mined protocols. It does not report false positives and automatically exercises training programs for protocol mining. In contrast to this work, [33] is limited to bugs that lead to an exception and may not cover the entire program.

B. Protocol Checking with Specifications

We build upon a complete, static checker for multi-object specifications [15]. The main reason for choosing Fusion for this work is that related static protocol checkers [34], [35] are sound but incomplete, that is, reported violations may be infeasible at runtime. In contrast, Fusion is complete and avoids this source of false positives.

Other approaches find protocol violations with runtime monitoring [36], [37] and static analysis [38]–[40]. A prerequisite for them is a specification describing legal and illegal call sequences. Our approach addresses this need by generating checkable specifications from mined protocols.

C. Combining Static and Dynamic Analysis

Whaley et al. [12] statically extract protocols from existing programs and check them dynamically. Instead, we infer protocols dynamically and check them statically. This allows us to leverage runtime information to focus on frequently occurring API usages and to check programs even when no test suites to exercise the program are available. Dallmeier et al. [13] evaluate a technique for refining mined protocols by searching seeded bugs with the refined protocols and a static typestate checker. Contrary to our work, their approach detects bugs only when they lead to an exception. Both [12] and [13] are limited to single-object protocols.

Yang et al. [41] present a dynamic specification miner and validate some of the mined specifications by feeding them into a model checker. Their work focuses on pairs of methods that must be called in alternating order. Yang et al. manually select mined specifications to check, whereas our approach is fully automatic.

D. Fault Localization

There are techniques for localizing faults based on multi-object properties [42], [43]. In contrast to our work, they require program input that triggers a bug.

VII. Conclusions

This work tackles the problem of finding illegal API usages involving multiple objects in an automatic way and with satisfactory precision. To address this problem, we combine a precise, static analysis, which requires specifications of API usage constraints, with a dynamic specification miner, which infers protocols from existing programs. The result of this symbiosis is a *fully automatic* bug detection system for *multi-object protocols*. Considering multiple interacting objects allows our approach to specify and check constraints that are not expressible with single-object protocols.

Experiments with 1.6 MLOC show the effectiveness of the approach. Our technique detects 41 issues with a 51% true positive rate; several of these issues cannot be detected by existing bug detection techniques that use mined specifications. These results are particularly compelling since the analyzed software corpus consists of well-tested and mature programs that are part of a widely used benchmark suite. Our results show that multi-object protocols are relevant in practice: 61% of the mined protocols are multi-object protocols, and 44% of the true positives are from multi-object protocols. We conclude from these results that combining a dynamic protocol miner with a precise, static analysis to find multi-object protocol bugs can reveal serious problems in well-tested programs.

Acknowledgments

The work presented in this paper was partially supported by the Swiss National Science Foundation under grant number 200021-134453, a fellowship from Los Alamos National Laboratory, the U.S. Department of Defense, and U.S. NSF grant CCF-0811592.

References

[1] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *SOSP*, 2001, pp. 57–72.

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

[3] Z. Li and Y. Zhou, "PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *ESEC/FSE*, 2005, pp. 306–315.

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

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

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

[7] H. Zhong, L. Zhang, and H. Mei, "Inferring specifications of object oriented APIs from API source code," in *APSEC*, 2008, pp. 221–228.

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

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

[10] 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.

[11] C. Lee, F. Chen, and G. Rosu, "Mining parametric specifications," in *ICSE*, 2011.

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

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

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

[15] C. Jaspan and J. Aldrich, "Checking framework interactions with relationships," in *ECOOP*, 2009, pp. 27–51.

[16] 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.

[17] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *ASE*, 2009, pp. 283–294.

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

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

[20] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking api protocol conformance with mined multi-object specifications, companion report," ETH Zurich, Tech. Rep. 752, March 2012.

[21] N. Gruska, A. Wasylkowski, and A. Zeller, "Learning from 6,000 projects: Lightweight cross-project anomaly detection," in *ISSTA*, 2010, pp. 119–130.

[22] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 1999.

[23] N. E. Beckman, D. Kim, and J. Aldrich, "An empirical study of object protocols in the wild," in *ECOOP*, 2011.

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

[25] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *OOPSLA Companion*, 2004, pp. 132–136.

[26] 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.

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

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

[29] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *TACAS*, 2005, pp. 461–476.

[30] 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.

[31] B. Hackett, M. Das, D. Wang, and Z. Yang, "Modular checking for buffer overflows in the large," in *ICSE*, 2006, pp. 232–241.

[32] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, "From uncertainty to belief: Inferring the specification within," in *OSDI*, 2006.

[33] M. Pradel and T. R. Gross, "Leveraging test generation and specification mining for automated bug detection without false positives," in *ICSE*, 2012.

[34] E. Bodden, "Efficient hybrid typestate analysis by determining continuation-equivalent states," in *ICSE*, 2010, pp. 5–14.

[35] N. A. Naeem and O. Lhotak, "Typestate-like analysis of multiple interacting objects," in *OOPSLA*, 2008, pp. 347–366.

[36] 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.

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

[38] K. Bierhoff and J. Aldrich, "Modular typestate checking of aliased objects," in *OOPSLA*, 2007, pp. 301–320.

[39] R. DeLine and M. Fähndrich, "Typestates for objects," in *ECOOP*, 2004, pp. 465–490.

[40] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, "Effective typestate verification in the presence of aliasing," *ACM T Softw Eng Meth*, vol. 17, no. 2, pp. 1–34, 2008.

[41] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal API rules from imperfect traces," in *ICSE*, 2006, pp. 282–291.

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

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