Discovering Complete API Rules with Mutation Testing

Anh Cuong Nguyen and Siau-Cheng Khoo
Department of Computer Science
National University of Singapore
Singapore, Singapore
{anhcuong, khoosc}@comp.nus.edu.sg

***Abstract*—Specifications are important for many activities during software construction and maintenance process such as testing, verification, debugging and repairing. Despite their importance, specifications are often missing, informal or incomplete because they are difficult to write manually. Many techniques have been proposed to automatically mine specifications describing method call sequence from execution traces or source code using *frequent pattern mining*. Unfortunately, a sizeable number of such "interesting" specifications discovered by frequent pattern mining may not capture the correct use patterns of method calls. Consequently, when used in software testing or verification, these mined specifications lead to many false positive defects, which in turn consume much effort for manual investigation.**

We present a novel framework for automatically discovering legitimate specifications from execution traces using a *mutation testing* based approach. Such an approach gives a semantics bearing to the legitimacy of the discovered specifications. We introduce the notion of maximal precision and completeness as the desired forms of discovered specifications, and describe in detail suppression techniques that aid efficient discovery. Preliminary evaluation of this approach on several open source software projects shows that specifications discovered through our approach, compared with those discovered through frequent pattern mining, are much more precise and complete. When used in finding bugs, our specifications also locate defects with significantly fewer false positives and more true positives.

***Keywords*-specification mining; formal specifications; mutation testing**

I. Introduction

Specifications are essential for many tasks during software construction and maintenance process. Legitimate specifications (i.e. specifications that programmers should obey when writing code) guide the production of correct code during coding activities and later can be used as system invariants for verifying, testing, debugging and repairing software [5], [6], [22]. They can also be used to facilitate the comprehension, understanding and maintaining of an existing code base [13]. Among these, *temporal logic* properties over function or method call sequences assemble an important and useful specification class [3], [23], [13]. These specifications can be effectively used to describe complex reliability and safety properties of software system such as lock acquisition and release, or resource ownership properties like "all calls to `read(f)` must exist between calls to `open(f)` and `close(f)`".

Specification mining techniques have recently been used to automatically discover temporal logic specifications from execution traces or source code. Many of these techniques share a fundamental hypothesis: "interesting" specifications tend to occur frequently in the set of traces. Based on this hypothesis, there has been some success in discovering specifications from trace sets through mining [9], [13]. Unfortunately, when used for discovering API-related rules, mining often produces many incorrect and incomplete rules. The presence of these *illegitimate* specifications in turn creates many false positives when used in software verification, and consumes much effort for manual investigation [11], [18], [19], [22]. For instance, Wasylkowski *et al.* reported in [22] an experiment using mined rules to find object usage anomalies in ASPECTJ, a compiler for the AspectJ language. Among 790 usage violations (anomalies) found, only 7 violations lead to real bugs (>99% of false positives).

We believe that a legitimate specification should have semantic bearing on identifying faulty call patterns. In other words, a specification is considered legitimate if violation of which can lead to error. In [15], we introduce the idea of *mutation test* based approach to identify legitimate specifications. This approach examines the legitimacy of a mined specification by subjecting it to mutation test. Specifically, given a mined specification that describes the behavior of a code fragment, we mutate the code so that it intentionally violates the specification. We then execute the code with a specific input to determine if such a violation of the specification will lead to exceptions being thrown at appropriate program points. When that happens, we deem the mined specification to be legitimate.

This mutation-testing approach significantly reduces the number of irrelevant mined rules, leading to a signficant reduction in the number of false positives. Specifically, the resulting set of mined rules can detect defects with at least 50% of accuracy [15]. Nevertheless, a large number of false positives remains, as we continue to adopt the hypothesis that legitimate rules must be "interesting" in the sense of having high statistical frequency. Indeed, the notion of "statistical significance" naturally omits call patterns that do not occur frequently in the execution traces, even though

978-1-4673-1761-0/12/$31.00 © 2012 IEEE

violation of such patterns may lead to program crash.

In the current work, we abolish the use of "statistical significance" hypothesis, and adopt the hypothesis that legitimate specifications must demonstrate their semantic significance. This naturally leads to a much bigger pool of specification candidates for legitimacy check. We continue to employ mutation-testing approach to determine the legitimacy of every potential specification. In order to ensure the effectiveness of legitimacy check, we introduce the concepts of *maximal precision* and *completeness* as the goals for constructing legitimate specifications, and propose a novel mutation algorithm called *binary suppression* to speed up the check. At an abstract level, our goal appears similar to that of Alattin presented by Thummalapenta and Xie in mining "alternative patterns" by detecting "neglected conditions" [18]. However, the differences in the underlying hypotheses and the mining objectives have led to entirely different solutions. We will say more about this is in Section VII.

We develop COPPICE to materialize this specification discovery framework. COPPICE can infer, fully automatically, maximally precise and complete *past-time temporal logic* API rules *with respect to a set of execution traces*. We conducted an extensive empirical experiment and discovered Java API rules using several open source software projects. Our results are promising: using COPPICE, we discovered 60-200% more legitimate rules (i.e., rules with 100% precision) than frequent pattern mining approach. When used in finding bugs, these rules drastically reduced 66% of false positives compared to those obtained from our previous mining approach presented in [15].

This paper makes the following contributions:

1) We provide a theoretical foundations for the identification and construction of maximally precise and complete legitimate specifications. We show that complete legitimate specifications are helpful to elimination of false positives in the verification of legitimate specifications (Section III).
2) We propose a general *mutation framework* for discovering legitimate temporal API rules from execution traces (Section IV). We provide a novel instance of this mutation framework for discovering complete past-time temporal API rules for Java programs, which consists of an efficient algorithm for managing mutation called binary suppression. This instance defines an algorithm for discovering, fully automatically, precise and complete API rules and requires no extraneous parameter (such as support and confidence commonly provided in mining process) beyond the program and program inputs (Section V).
3) We realize our framework by constructing an efficient and practical system called COPPICE and perform an extensive evaluation to demonstrate its effectiveness and scalability using several open source software projects (Section VI).

II. MOTIVATING EXAMPLES

We motivate our work by first illustrating two common pitfalls in using frequent pattern mining techniques to infer API rules: (1) such techniques may produce rules which has no semantics significance, and thus considered illegitimate; (2) such techniques may omit rules which have semantics significance, but fails to occur frequently in the execution traces.

In the following two examples, we demonstrate these drawbacks of frequent pattern mining and explain how they constitute a source of false positives when mined rules are used in software verification. These examples also give readers an intuition behind the notion of legitimate rules and complete rules. Here, and throughout the paper, we consider API rules as past-time temporal properties representing sequential usage of method calls, possibly from multiple objects.[1]

Example 1. Consider the following code snippet, which is excerpted from APACHE FOP [1], a print formatter for XSL formatting objects.

```
1  private Stack nestedBlockStack = new Stack();
2  public void handleWhiteSpace(...) {
3    ...
4    if (nestedBlockStack.empty() || fo !=
         nestedBlockStack.peek()) {
5      ...
6      nestedBlockStack.push(currentBlock);
7    } else {
8      nestedBlockStack.pop();
9    }
10   ...
11   if (!nestedBlockStack.empty())
12     nestedBlockStack.pop();
13   ...
14 }
```

Let's suppose that we want to infer usage rules of the `Stack` object in the above code using its method call execution trace. The execution flow from line 4 to line 8 produces the rule `pop` $\hookrightarrow$ `peek`, which states that a stack must perform a call to `peek` before it can perform a call to `pop`, and the execution flow from line 11 to line 12 produces the rule `pop` $\hookrightarrow$ `empty`, which states that "a stack must be checked for emptiness (by performing a call to `empty`) before it can perform a call to `pop`". In our experiment, the rule `pop` $\hookrightarrow$ `peek` occurs frequently during program execution (with at least 122 occurrences); however, the rule is unsuitable for use as an invariant in verification, because it is not legitimate in general. The rule `pop` $\hookrightarrow$ `empty`, by contrast, states a legitimate rule that any stack should firmly follow. Consequently, any violation of `pop` $\hookrightarrow$ `empty` in verification will likely be true positives, whereas any alert due to a violation of `pop` $\hookrightarrow$ `peek` will highly probably be a false positive.

[1]Section III-A explains the representation in detail.

Example 2. Consider the following code snippet, which is excerpted from PMD [2], a bug finding tool for Java programs.

```
1  public String withoutPackageName(...) {
2    int dotPos = name.lastIndexOf('.');
3    return dotPos > 0 ? name.substring(dotPos + 1)
         : name;
4  }
```

The execution flow from lines 2 to 3 produces the rule `substring` $\hookrightarrow$ `lastIndexOf` for the `String` object, which can be interpreted as one should use the method `lastIndexOf` to find the position where the string should be truncated before truncating it by performing the `substring` method. This rule occurs very infrequently (with only 1 occurrence) during the program execution in our experiment and consequently is not inferred by frequent pattern mining technique. Typically, a more frequently occurred rule such as `substring` $\hookrightarrow$ `indexOf` will be produced by the miner. If we verify the rule `substring` $\hookrightarrow$ `indexOf` alone against the above code, we will find a violation. This violation is however a false positive because the call to `substring` is actually safe due to the presence of the call to `lastIndexOf`. Alternatively, one would want to use a more *complete* rule such as `substring` $\hookrightarrow$ `indexOf OR lastIndexOf`, which can help to eliminate the false alarm described.

In order to minimize the number of false positives produced when using mined specifications for software verification purpose, we propose in this work the use of both *precise* and *complete legitimate* specifications, whose formal definitions will be presented in Section III-B. As for what kinds of specifications can be considered as legitimate, our key insight is that legitimate rules like `pop` $\hookrightarrow$ `empty` and `substring` $\hookrightarrow$ `lastIndexOf` are not only used to implement the code functionality but also mandatory for the participating objects to behave correctly. Consequently, the legitimacy of these rules can be determined by observing how the objects respond when the rules are violated. For example, a precondition for a stack to perform a pop is that the stack must be at a non-empty state. Therefore a call to `empty`, in contrast with a call to `peek`, is necessary to check for the legal state of the stack object before it can perform a pop. Generally, *a rule is deemed legitimate when violating the rule during program execution can make the participating objects misbehave, which lead to a bad system behavior, and possibly exhibited by an unexpected system error*.

III. Legitimate Specification

In this section, we will first formalize the syntax of past-time temporal specification and describe how we use it to verify program behavior (Section III-A). This is followed by formal definitions of legitimate specifications, precise and complete legitimate specifications, and an explanation on how complete legitimate specifications can help eliminate false bugs in verification of legitimate specifications (Section III-B). These definitions serve as a theoretical foundation for our mutation framework for discovering legitimate specifications, which we will discuss in Section IV.

Table I
Specifications and their Past-time LTL Equivalences

Specification	LTL Equivalent
$a \hookrightarrow b$	$G(a \rightarrow X^{-1}F^{-1}b)$
$\langle a, b\rangle \hookrightarrow c$	$G((a \wedge XFb) \rightarrow X^{-1}F^{-1}c)$
$a \hookrightarrow \langle b, c\rangle$	$G(a \rightarrow X^{-1}F^{-1}(c \wedge X^{-1}F^{-1}b))$

Table II
Specification Violations and their Past-time LTL Equivalences

Specification	Violation	LTL Equivalent of Violation
$a \hookrightarrow b$	$a \overset{v}{\hookrightarrow} \neg b$	$F(a \wedge X^{-1}G^{-1}\neg b)$
$\langle a, b\rangle \hookrightarrow c$	$\langle a, b\rangle \overset{v}{\hookrightarrow} \neg c$	$F((a \wedge XFb) \wedge X^{-1}G^{-1}\neg c)$
$a \hookrightarrow \langle b, c\rangle$	$a \overset{v}{\hookrightarrow} \langle \neg b, c\rangle$ $a \overset{v}{\hookrightarrow} \langle b, \neg c\rangle$ $a \overset{v}{\hookrightarrow} \langle \neg b, \neg c\rangle$	$F(a \wedge X^{-1}G^{-1}(c \rightarrow X^{-1}G^{-1}\neg b))$

A. Past-time Temporal Specification

Past-time temporal specifications are rules stating that *"whenever a series of events occurs, previously another series of events must have happened"*. Specifications of this format are commonly found in practice and useful for program testing and verification [14].

In this work, a past-time temporal specification represents an interaction protocol between possibly multiple objects in terms of call sequence usage and is always constructed from two components, a *consequence* and a *premise*, each consists of a series of method calls. A specification is denoted as `consequence` $\hookrightarrow$ `premise` and states that whenever a series of consequence calls occurs it must be preceded by another series of premise calls. Each past-time temporal specification corresponds to a Linear-time Temporal Logic (LTL) expression. Examples of this correspondence are shown in Table I, which we borrow from [14]. Each character represents a call. A sequence of characters surrounded by angle brackets represents a series of (not necessarily consecutive) calls that are invoked in that respective order.

Finally, a *violation* of a past-time temporal specification can occur when one or more events in the premise are missing while all events in the consequence occur in the correct order. Violations of specifications used in Table I are shown in Table II. We use the notation $\overset{v}{\hookrightarrow}$ to symbolize violation.

B. Legitimate Specification

As mentioned in Section I, legitimate specifications are rules that programmers must follow when writing code.

In our definition, legitimate specifications do not merely describe code functionalities, they are rules that require the objects participating in the code to obey. When such a specification is violated, the participating objects will likely misbehave (that can be exhibited by throwing exceptions) at some calls stated in the specification. Consequently, legitimate specifications are useful for software verification purpose. A formal definition (Definition 1) is as follows. This definition also appeared in [15] under the name *object-significant specification*.

Definition 1 (Legitimate Specification)**.** A specification $Spec$ is legitimate if and only if there exist a code fragment C and its input I such that code C when executed with I will crash, due to the violation of $Spec$, by throwing an exception at a call c occurred in the consequent component of $Spec$.

$$Spec.leg() \Leftrightarrow \exists\, C, I\ \exists\, c \in Spec.cons : \\ \big(\ C.exec(I)\ \wedge\ Spec.violate(C, I)\ \big)\ \wedge\ \lightning@c$$

We also call (C, I) a *violating instance* of the specification $Spec$.

In this definition, the notation $Spec.leg()$ means the specification $Spec$ is legitimate, $Spec.cons$ denotes the set of calls in the consequence component of the specification, $C.exec(I)$ represents the execution of code C with input I, $Spec.violate(C, I)$ indicates the specification $Spec$ is violated during the execution of code C with I, and finally $\lightning@c$ denotes throwing of an exception at call c. In addition, an instance of a legitimate specification is defined as follows (Definition 2).

Definition 2 (Legitimate Instance)**.** A pair comprising a code fragment and its input (C_{org}, I) is called a legitimate instance of a legitimate specification $Spec$ if and only if the execution of C_{org} with I satisfies $Spec$ and C_{org} can be mutated into C_{mut} such that (C_{mut}, I) is a violating instance of $Spec$.

We now define the concept of precise legitimate specification and complete legitimate specification (Definitions 3-6). The use of these kinds of specifications during software verification can minimize the number of false bugs detected. In these definitions, we use an uppercase character to represent the premise or the consequence component of the specification, which can consist of either a single call or a series of calls. When we want to emphasize the series of calls in each component, we use a sequence of lowercase characters surrounded by angle brackets instead.

Definition 3 (Precise Legitimate Specification)**.** A legitimate specification of multi-event premise $A \hookrightarrow \langle b_1, b_2, \ldots, b_n\rangle$ is precise if and only if there exist a code fragment C and its input I such that (C, I) is a legitimate instance of $A \hookrightarrow \langle b_1, b_2, \ldots, b_n\rangle$ and of each of the single-event premise specifications $A \hookrightarrow \langle b_i\rangle, \forall i = 1 \ldots n$.

Intuitively, a legitimate specification is precise when all calls in its premise are legitimate for the calls in consequence. Notice that an imprecise legitimate specification with some illegitimate calls in premise may lead to the detection of false bugs. For example, consider the specification `pop()` $\hookrightarrow$ ⟨ `peek()`, `empty()` ⟩ of the `Stack` object. The specification is legitimate because a violation of the specification in which the call to `empty()` is missing will cause the `Stack` object to throw an exception at the call `pop()` during execution. However, the specification is not a precise legitimate specification because the specification `pop()` $\hookrightarrow$ `peek()` is not legitimate. Consider a safe code in which a call to `empty`, and without any calls to `peek`, is called before the call to `pop`, such as the code fragment including lines 11 and 12 in Example 1, Section II. Verification of such a code against the specification `pop()` $\hookrightarrow$ ⟨ `peek()`, `empty()` ⟩ will result in a violation (in which the call to `peek` is expected but missing), however the violation implies a false bug.

Usually, we would like to combine all single premise legitimate specifications $A \hookrightarrow \langle b_i\rangle$ which share the same consequence A into one single precise legitimate specification using conjunction operator $A \hookrightarrow \langle b_1, b_2, \ldots, b_n\rangle$ if they all share the same legitimate instance (C, I). Because this combination specifies a temporal ordering of all calls $b_i, i = 1 \ldots n$ as they appear in the execution of code C with input I, the combined specification is more precise than each of the single premise specification alone. For this reason, we usually want to infer the *maximal precise legitimate specification*.

Definition 4 (Maximal Precise Legitimate Specification)**.** A precise legitimate specification $A \hookrightarrow \langle b_1, b_2, \ldots, b_n\rangle$ is maximal if and only if there exists an instance of the specification (C, I) such that there are no other specifications $A \hookrightarrow \langle b_{n+1}\rangle, b_{n+1} \neq b_i, i = 1 \ldots n$ of which (C, I) is a legitimate instance. We also call (C, I) the *maximal legitimate instance* of $A \hookrightarrow \langle b_1, b_2, \ldots, b_n\rangle$.

Sometimes, two legitimate specifications $A \hookrightarrow B_1$ and $A \hookrightarrow B_2$, which share the same consequence A, do not share the same legitimate instance (C, I). We say these specifications are *exclusive* of one another and combine them using disjunction operator.

Definition 5 (Specification Exclusivity)**.** Two legitimate specifications $A \hookrightarrow B_1$ and $A \hookrightarrow B_2$ are exclusive if and only if there exists two pairs of code fragment and input (C_1, I_1) and (C_2, I_2) such that (C_1, I_1) is a legitimate instance of $A \hookrightarrow B_1$ but not $A \hookrightarrow B_2$, and (C_2, I_2) is a legitimate instance of $A \hookrightarrow B_2$ but not $A \hookrightarrow B_1$.

Definition 6 (Complete Legitimate Specification)**.** A set of maximal precise legitimate specifications $\{A \hookrightarrow B_1, \ldots, A \hookrightarrow B_n\}$, which can be expressed as a single specification $A \hookrightarrow (B_1 \vee \cdots \vee B_n)$, is complete if

and only if there are no other maximal precise legitimate specifications $A \hookrightarrow B_{n+1}, B_{n+1} \neq B_{\mathtt{i}}, i = 1 \ldots n$ that are mutually exclusive with the given set of legitimate specifications.

Intuitively, a complete legitimate specification consists of the biggest set of mutually exclusive specifications which share the same consequence component. A concrete example that illustrates this situation has been shown in Example 2. We now show how complete legitimate specifications can help reduce false bugs when they are used in place of incomplete ones to verify software application.

Theorem 1. Given a complete legitimate specification $Spec$ and an incomplete one $Spec'$ obtained by removing one of the exclusive disjunct of $Spec$. The number of false bugs encountered during verification against $Spec$ is bounded above by that during verification against $Spec'$.

In practice, as we will show in Section VI, complete legitimate specifications usually help reduce significantly the number of false bugs when they are used in place of incomplete ones to verify software application.

IV. COPPICE FRAMEWORK

A. Overview of the Framework

We now present COPPICE, the mutation framework for discovering legitimate API rules from a software application and a set of its input. Figure 1 shows an overview of the framework. Input to COPPICE is a software application and a set of its input. The inputs of the software application can be either the inputs for the system tests or the unit tests. COPPICE is most useful when developers wish to extract legitimate rules used in completed software projects, and the system inputs or unit tests are usually widely available in completed projects. Output of COPPICE is a set of legitimate API rules found during the application execution with the set of provided inputs.

COPPICE comprises two customizable components, the *candidate rule extractor* and the *mutation generator*. They respectively characterize the set of rules the system will consider, and determine the appropriate injected mutations for verifying the rules' legitimacy. The last component, *legitimate rule verifier*, is fixed and can be shared by all instances of the framework. Following we describe each component in detail.

Candidate Rule Extractor. The first layer of the candidate rule extractor is the *method call tracer*. The tracer's duty is to execute the software application with its inputs, and record traces of API calls during the application execution. By default, our implementation of the framework provides one inter and one intra procedural tracer. However, the tracer can also be customized, for example, to be a more domain-specific tracer such as the *object collaboration* tracer described in [17]. The second layer is the candidate rule extractor. It accepts the set of traces and produces a set of candidate temporal rules. This layer also affects the degree of completeness which the inferred legitimate API rules can be. In [15], we use a frequent pattern miner as an instance of this second layer, and extract frequent temporal rules by mining the traces. As mentioned in Section I, many infrequent but legitimate rules may be omitted by the miner. In this work, we will use a *maximal rule extractor*, which can extract all possible rules present in the set of traces. We will discuss this new extractor further in Section V-A.

Mutation Generator. The Mutation Generator accepts a set of candidate rules and a software application and produces the mutated versions of the application. Mutations can be injected in the form of suppressing existing method calls or introducing new method calls. Each of the mutated applications will attempt to violate a particular candidate rule at runtime. A challenge here is that the number of ways to mutate a piece of code to effect rule violation can be humongous. On the other hand, we notice that some but not all mutations are required for judging the legitimacy of the candidate rule. For example, consider a past-time temporal specification with premise of length n, $A \hookrightarrow \langle b_1, b_2, \ldots, b_n \rangle$. A possible mutation to violate the rule is to suppress calls in a subset of the premise call set $\{b_1, b_2, \ldots, b_n\}$ (except for the empty set). Thus, there can be $2^n - 1$ different mutations. On the other hand, if it has been found that $A \hookrightarrow \langle b_i \rangle$ (for some i) is not legitimate, there is no need to do mutation test for $A \hookrightarrow \langle b_i, b_j \rangle$ (for the same i and some $j \neq i$). In [15], we introduce *linear suppression* and show that we only need n mutations to judge the legitimacy of the candidate rule. Unfortunately, this suppression technique does not scale well for our current work. We will therefore introduce an even more efficient mutation algorithm called binary suppression. We will elaborate on this further in Section V-B.

Legitimate Rule Verifier. Legitimate rule verifier takes in a set of mutated versions of a software application and executes them using the same set of inputs provided at the beginning. Any abnormal behaviors (e.g. throwing exception, null pointer creation, memory leak, etc.) occurring at some pre-determined program points signify violations of the corresponding rule under verify. When this happens, the rule is deemed to be legitimate and the corresponding mutations are incorporated into other mutations to further refine the precision of the rule. In the end, all legitimate rules are collected, refined and delivered to users.

B. SPECCHECK

In previous work, we introduced SPECCHECK, an instance of COPPICE framework for discovering frequent and precise legitimate API rules [15]. SPECCHECK adopts the hypothesis that "interesting" specifications tend to occur frequently in the set of traces. It therefore uses a frequent pattern miner

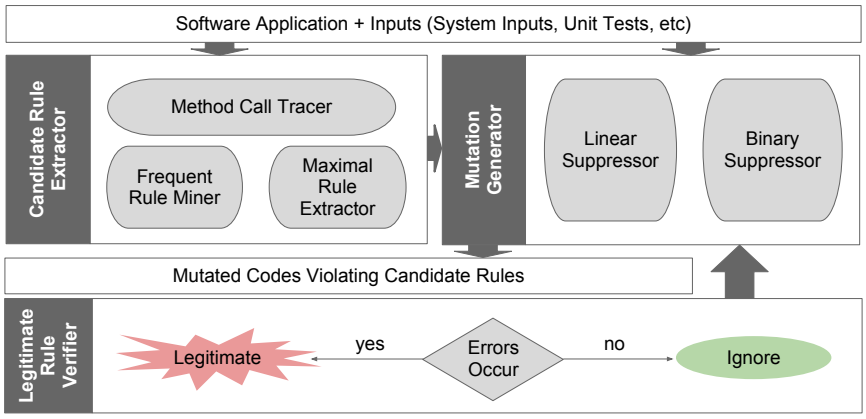

Figure 1. COPPICE Framework for Discovering Legitimate Specifications

to define the candidate rule extractor. It then uses a *linear suppressor*[2] to define the mutation generator component. While SPECCHECK also attempts to infer precise legitimate specifications and combine exclusive specifications, the theory behind the construction of legitimate specifications was immature. Furthermore, as mentioned in Section I, the adoption of "statistical frequency" hypothesis leaves SPECCHECK in a disadvantage position in producing maximally precise and complete specifications.

V. AN INSTANCE OF COPPICE

In this work, we introduce a new instance of COPPICE that can discover complete past-time temporal API rules, *with respect to the given software application and its inputs*. We now describe the candidate rule extractor and mutation generator of this new instance in detail, respectively in Section V-A and Section V-B. Finally, we show how complete legitimate rules are generated from maximal precise legitimate rules in Section V-C. From now on, whenever we use the name COPPICE, we refer to this new instance.

A. Maximal Rule Extractor

In contrast with the frequent pattern miner used in SPECCHECK which only extracts frequent candidate rules from the execution traces, this maximal rule extractor extracts all possible candidate rules. We employ two practical heuristic to enable a compact representation of these candidate rules. Firstly, we consider only those temporal rules whose concatenations of their premise and consequence correspond to some *iterative patterns* in the execution traces (*cf.* Property 1). An iterative pattern captures "the total-ordering and one-to-one correspondence between events in the pattern". Using iterative pattern is a natural and unambiguous way to represent interactions between software components [13]. Secondly, we consider only temporal rules that have one method call in their consequences (and arbitrary number of calls in their premises).

In COPPICE, we employ an intra-procedural and *multiple-object* method call tracer. A typical trace looks like the following:

```
1  org/apache/xmlgraphics/fonts/Glyphs
2  ...
3  java/io/BufferedReader.readLine()Ljava/lang/
       String;:303
4  java/lang/String.indexOf(I)I:305
5  java/lang/String.substring(I)Ljava/lang/
       String;:307
6  ...
```

The first event in the trace, called *container*, identifies the name of the *class* in which calls are traced. Each of the subsequent events is a method call, and is identified by its name (*eg.* `readLine()` in event 3 above) and its source line number (*eg.*, 303 in event 3).

We extract candidate rules from execution traces as follows. We first collect the set of distinct calls from all execution traces. Then for each call, we proceed as follows. We fix the call to be the consequence of the candidate rules we are going to extract. From each trace, we then extract all subsequences that start from beginning of the trace and ends at an occurrence of the call to become candidate rule instances. The following property helps further refine the instances to conform with iterative pattern properties. This property is derived from the definition of iterative pattern instance described in [13].

Property 1. Consider a legitimate rule $\langle c \rangle \hookrightarrow \langle p \rangle$. An instance of the rule satisfies the following QRE expressions $p[-c]^*c$ and $p[-p]^*c$.[3]

The first condition implies that the premise p occurs between either two consecutive occurrences of c or the

[2]We will discuss more about *linear suppressor* in Section V-B

[3]QRE stands for quantitative regular expression. The expression simply says that there are no other occurrences of c or p in between p and c.

beginning of the trace and the first occurrence of c. In addition, when there are multiple occurrences of the premise p, the second condition states that we only need to consider the right most occurrence of p with respect to c. Consider an example in which we want to extract candidate rule instances with call consequence c from the trace $T = [a, b, \underline{c}, a, b, a, b, d, f, \underline{c}, g, h]$. Two raw candidate rule instances are $[a, b, \underline{c}]$ and $[a, b, c, a, b, a, b, d, f, \underline{c}]$. Using Property 1, the second instance can be reduced to $[a, b, d, f, \underline{c}]$.

B. Binary Suppression

Given a candidate rule $\langle c\rangle \hookrightarrow \langle p_1, p_2, \ldots, p_n\rangle$, we need to determine whether it is a legitimate rule. If it is, we need to further reduce it into a precise legitimate rule, and possibly maximal precise legitimate rule. We tackle this problem by first determine those $p_j, j = 1 \ldots n$ such that $\langle c\rangle \hookrightarrow \langle p_j\rangle$ is a legitimate rule. Then the precise legitimate rule reduced is $\langle c\rangle \hookrightarrow \langle p_{j_1}, p_{j_2}, \ldots, p_{j_m}\rangle$ where $\langle c\rangle \hookrightarrow \langle p_{j_k}\rangle$ is legitimate, $k = 1 \ldots m$, $j_k \in [1, n]$.

In SPECCHECK, we use linear suppression to verify and reduce a candidate rule into a precise legitimate rule. For each instance *(C, I)* of $\langle c\rangle \hookrightarrow \langle p_1, p_2, \ldots, p_n\rangle$ and for each call $p_j, j = 1 \ldots n$, we mutate code *C* to suppress the occurrence of p_j in C. The occurrence of p_j is identified by its container name, its name and source line number. The mutated code is then executed to determine whether each rule $\langle c\rangle \hookrightarrow \langle p_j\rangle$ is legitimate with respect to (C, I). Thus, for each instance (C, I), this algorithm takes linear time in the length of the rule under verify.

In COPPICE, the candidate rules we need to deal with are much longer than those dealt by SPECCHECK. We therefore introduce a new mutation algorithm called binary suppression that in average takes only logarithmic time in the length of the rule under verification for each instance (C, I). The idea is to mimic binary search algorithm to find specific calls p_j within $\{p_1, p_2, \ldots, p_n\}$ such that $\langle \subset\rangle \hookrightarrow \langle p_j\rangle$ is legitimate. At each stage, the algorithm mutates code *C* to suppress occurrences of calls in the premise sequence. If the mutated code when executed does not result in an exception throwing at call c, then all calls in the premise sequence are not legitimate with respect to instance *(C, I)* and the algorithm terminates. Otherwise the premise sequence is divided into two equal halves, and the algorithm repeats its action on each half. In the degenerated case where the premise sequence consists of only one call p_j, the algorithm concludes that $\langle c\rangle \hookrightarrow \langle p_j\rangle$ is legitimate with respect to (C, I) and terminates this branch of testing.

Figure 2 illustrates a run of the binary suppression algorithm on a concrete candidate rule in form of a tree. The candidate rule is extracted from FOP application [1]. Each node in the tree represents a premise sequence under verification. A dotted-line node represents an illegitimate premise sequence, which is determined by the fact that suppressing all calls in the premise does not cause `substring` to throw any exception. The suppression algorithm terminates at dotted-line nodes. A dashed-line node represents a legitimate premise sequence, which is signified by the fact that its suppression causes `substring` to throw an exception. Each dashed-line node is divided into two almost equal halves and the algorithm repeats its action on each half. Finally, the suppression of a single call `indexOf` causes an exception being thrown at `substring`. Therefore, one precise legitimate rule is found, which is `substring` $\hookrightarrow$ `indexOf`.

The following theorem (Theorem 2) affirms that by employing maximal rule extractor and binary suppression, COPPICE infers maximal precise legitimate rules from the given software application and its inputs.

Theorem 2. Binary suppression algorithm reduces candidate rules inferred by maximal rule extractor into maximal precise legitimate rules.

Binary suppression performs at its worst when all calls in the premise sequence are legitimate. It is best when all calls in the premise sequence are illegitimate. As in practice, the number of legitimate calls in the premise sequence is usually much smaller than the size of the premise sequence, binary suppression outperforms linear suppression. For example, given the candidate rule shown in Figure 2, in our experiment, binary suppression finishes in 8.78s while linear suppression requires 18.43s.

Call suppression process. We now describe in more detail the call suppression process. As stated previously, each call in our work is identified by its container, name and source line number. During the load time of the call's container, we identify the call's occurrence by its name and source line number. We then suppress the call by replacing it with selection of objects of its returned type. A small number of objects will be created as replacements, and are subjects of mutation tests. In particular, values of primitive type objects will be generated randomly, each taken from a pre-defined value range of the corresponding primitive type. Moreover, other types of objects are generated inductively from their field parameters. Some types of objects cannot be generated by our method (e.g. objects without public constructors). This situation rarely occurs in our experiment; however, when it happens, we simply assume that the call is legitimate and do not perform suppression of the call. Nevertheless, the corresponding inferred rule is marked for manual review.

C. Inference of Complete Legitimate Rules

COPPICE finally combines maximal precise legitimate rules inferred into complete legitimate rules. Given a set of inferred premises $\{P_1, P_2, \ldots, P_n\}$ that shares a same consequence `C`, COPPICE first removes all premises P_j which is a super-sequence of another premise P_i in the

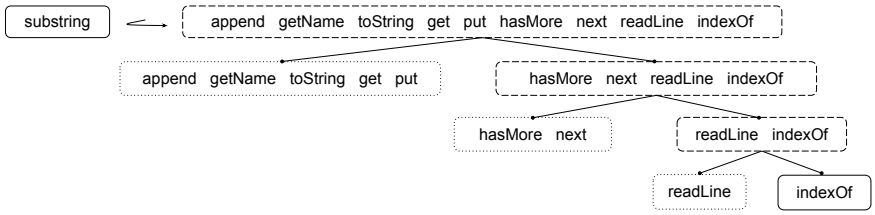

Figure 2. Binary Suppression Tree for a Candidate Rule Instance in FOP

set. The remaining premises $\{P_{i_1}, P_{i_2}, \ldots, P_{i_m}\}$ are used to construct the rule $C \hookrightarrow (P_{i_1} \vee P_{i_2} \vee \ldots \vee P_{i_m})$. Since COPPICE already considers all maximal precise legitimate rules used in the execution traces, the rule inferred is complete with respect to the given set of execution traces.

VI. EVALUATION

We evaluate COPPICE using 6 software applications obtained from DACAPO 2009 benchmark (see Table III) [4]. We use the test harness provided by the benchmark as inputs to execute 6 applications. We collect traces of only calls to the Java standard API library and infer Java API rules. As previously stated, we use an intra-procedural and multiple-object method call tracer. Research questions regarding of the precision and effectiveness of legitimate specifications have been studied in our previous work [15]. In this work, we study the completeness of rules inferred by COPPICE, and the benefits for using them in eliminating false software bugs. We achieve this by comparing COPPICE with our original tool SPECCHECK. We conduct our experiment using an Intel Core i5 M460 computer with 2GB memory running Linux Mint 10 Julia and use all its 4 cores.

A. Inferring Complete Legitimate Rules

We first address the question of whether legitimate rules found by COPPICE are more complete than those found by SPECCHECK. We ran the two tools with the same set of traces collected from 6 target applications and compared their results.

SPECCHECK Setup. SPECCHECK requires setting support and confidence parameters to mine frequent candidate rules from traces. This setting affects the completeness of legitimate rules inferred by SPECCHECK. We set these parameters so that SPECCHECK can mine as many candidate rules as possible, given that the mining process must finish within 1 hour. As such, for each target system, we set the value of support parameter from 5-15 and confidence from 60-80% depending on the size of the traces.

Evaluation Result. Table III compares the number of legitimate rules found by SPECCHECK and COPPICE. For SPECCHECK, we report the number of mined rules that it first considered (Frequent), and the number of legitimate rules it infers from the considered mined rules (Legitimate). For COPPICE, we report the number of legitimate rules it infers (Legitimate), the number of legitimate rules found by SPECCHECK that COPPICE is able to make them more complete (Complete+) and the number of new rules found by COPPICE that are not found by SPECCHECK. In our experiment, there are no legitimate rules found by SPECCHECK that COPPICE cannot find. In total, COPPICE is able to turn 64% (9 out of 14) of SPECCHECK rules into complete rules. Not only does COPPICE discover all legitimate rules discovered by SPECCHECK, it also infers 136% (19 out of 14) more legitimate rules than SPECCHECK.

Table IV shows a sampling of legitimate API rules inferred by COPPICE. This table shows that COPPICE can find rules with many alternative premises (e.g. 6 different premises for `String.substring(II)`) and legitimate rules of multiple objects. We inspect all rules inferred manually and find that they are all meaningful. However, few rules might be arguably not significant because they are used only in rare scenarios. For example, the rule `parseDouble` $\hookrightarrow$ `substring` is used to extract a string of double from a larger string and then parse it. Usually, the input string to `parseDouble` already represents a double, thus the call `substring` is not required. Nevertheless, because these rules are rarely used, they have a small number of occurrences in the set of traces (less than 5 in our experiment). Consequently, this number can guide users to remove these *infrequent rules* based on their preferences.

COPPICE Performance. We observe that in COPPICE the process of inferring legitimate rules from a candidate rule instance is independent from one another. We therefore parallelize these processes and exploit multi-core processors to speed up COPPICE performance. For each target application, there are thousands of candidate rules to consider but COPPICE finishes within 30 minutes to 2 hours.

B. Detecting Defects using Complete Legitimate Rules

We next address the question of whether the complete rules found by COPPICE are more useful than rules found by SPECCHECK. We perform this evaluation by comparing their abilities to detect bugs or code smells in 6 open source applications mentioned in the previous section. For verification of 6 applications against inferred rules, we used an in-house static verification tool, which is based on the SOOT flow analysis framework [20]. We then carefully

Table III
COPPICE INFERS MORE LEGITIMATE RULES.

	SPECCHECK		COPPICE		
Benchmark	**Frequent**	**Legitimate**	**Legitimate**	**Complete+**	**New**
avrora	46	1	3	0	2
batik	338	8	13	4	5
fop	168	7	18	3	11
jython	199	4	18	3	14
luindex	52	3	8	0	5
pmd	51	5	14	0	9
Total	809	14	33	9	19

Table IV
A SAMPLING OF LEGITIMATE API RULES INFERRED BY COPPICE

Java Class	**Legitimate Rules**		
`String`	`substring(II)`	$\hookrightarrow$	`length()` $\vee$ `indexOf(I)` $\vee$ `indexOf(II)` $\vee$ `lastIndexOf(String)` $\vee$ `indexOf(String)` $\vee$ `startsWith(String)`
`Stack`	`pop()`	$\hookrightarrow$	`isEmpty()` $\vee$ `push()` $\vee$ `size()`
$\texttt{Double}_1$, $\texttt{String}_2$	$\texttt{parseDouble(String)}_1$	$\hookrightarrow$	$\texttt{substring(II)}_2$
$\texttt{Rectangle}_1$, $\texttt{BufferedImage}_2$	$\texttt{getData(Rectangle)}_2$	$\hookrightarrow$	$\texttt{intersection(Rectangle)}_1$ $\vee$ $\texttt{translate(II)}_1$

inspected all violations reported and classified them as either true or false positives.

Table V shows the results of our comparison. It reports the number of anomalies (ie., violation of specifications), true and false bugs or code smells (code that indicate something may go wrong [22]). Among 6 applications, SPECCHECK's rules uncover bugs or code smells with 32%(26/79) of precision, while COPPICE's rules achieves 60%(28/46) of precision. Futhermore, COPPICE helps reduce 66%(35/53) of false positive rate and uncover 2 more new true positives. Specifically, *complete+ rules* found by COPPICE help eliminate false positives while *new rules* found by COPPICE help uncover new true positives. Nevertheless, some false positives still exist; we attribute this to the fact that rules are extracted from a finite set of traces, a limitation inherent in dynamic analysis.

VII. RELATED WORK

Our mutation framework, as presented in Section IV, is closely related to works in software specification mining and software verification using mined specifications. Many earlier works mine specifications to aid software comprehension [3], [23], [9], [21]. The mined specifications can be in the form of automata, sequential patterns, etc. Lo et al. propose to mine iterative patterns that better capture software component interactions [13]. This work is later extended to mining live sequence charts, inter-object behaviors of arbitrary sizes [8].

The issue with legitimacy of mined specification has attracted attention lately, with Thummalapenta and Xie focus on mining classes of exception-handling rules to detect unsafe code [19]. Gous and Weimer opine that legitimate rules should come from trustworthy code, and propose to use software metric such as repository, version history and source code to filter input traces with acceptable trustworthiness metrics [12]. OCD infers rules only from traces of small fixed-size windows, as the authors affirm that events in a programming rule are usually invoked closely to one another during execution [10]. JADET, and later CHECKMYCODE, mine arbitrary rules but adopt heuristic ranking to determine anomalies [22], [11]. SPECCHECK was the first to determine rule legitimacy based on semantic reasoning [15]. The current work, COPPICE, extends SPECCHECK to further infer complete legitimate rules. A similar work by Thummalapenta and Xie, ALATTIN, mines both frequent and infrequent programming rules [18]. Their objective is to find alternative for frequent rules and to look at only rules involving conditional checks. Thus given the same set of inputs, there can be legitimate rules found by COPPICE that ALATTIN cannot find. On the other hand, ALATTIN adopts static analysis and online code search, and thus can discover some legitimate rules not available in the input traces. Finally, TAUTOKO tool also performs mutation operations, but for the purpose of test case generation [7].

VIII. CONCLUSION

In this paper, we have presented a general mutation framework, COPPICE, that can fully automatically infer legitimate rules from execution traces. COPPICE extends the previous work SPECCHECK in the following significant ways: It infers all legitimate rules discoverable from a set of traces, and presents those which are maximally precise and complete. Our evaluation of COPPICE on six matured open source projects showed that COPPICE can infer 136% more legitimate rules than SPECCHECK, and help make 64% of rules inferred by SPECCHECK more complete. When used to find bugs, additional rules inferred by COPPICE helped eliminate many false positives, as well as locate new bugs.

Table V
COPPICE RULES SHOW MORE TRUE POSITIVE AND LESS FALSE POSITIVE

		SPECCHECK			COPPICE		
Benchmark	**Classes**	**Anomalies**	**True**	**False**	**Anomalies**	**True**	**False**
avrora	1838	4	3	4	4	4	1
batik	2430	3	3	5	3	4	3
fop	1314	7	4	9	6	4	4
jython	2816	4	7	11	3	7	3
luindex	536	2	1	2	1	1	0
pmd	727	4	8	22	2	8	7
Total	9652	7	26	53	7	28	18

Finally, a version of this paper with proofs is available at [16]. An implementation of COPPICE is also available at http://www.comp.nus.edu.sg/~specmine/coppice.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their valuable feedbacks for improving the final version of this paper. This research is partially supported by the research grants R-252-000-402-112 and R-252-000-318-422.

REFERENCES

[1] Apache fop's homepage. http://xmlgraphics.apache.org/fop/.

[2] Pmd's homepage. http://pmd.sourceforge.net/.

[3] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL '02*, pages 4–16, 2002.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06*, pages 169–190, 2006.

[5] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. In *OOPSLA '07*, pages 569–588, 2007.

[6] C. Csallner, Y. Smaragdakis, and T. Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17:8:1–8:37, May 2008.

[7] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *ISSTA '10*, pages 85–96, 2010.

[8] T.-A. Doan, D. Lo, S. Maoz, and S.-C. Khoo. Lm: a miner for scenario-based specifications. In *ICSE'10*, pages 319–320, 2010.

[9] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *SIGSOFT '08/FSE-16*, pages 339–349, 2008.

[10] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *ICSE '10*, pages 15–24, 2010.

[11] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: lightweight cross-project anomaly detection. In *ISSTA '10*, pages 119–130, 2010.

[12] C. Le Goues and W. Weimer. Specification mining with few false positives. In *TACAS '09*, pages 292–306. 2009.

[13] D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. In *KDD '07*, pages 460–469, 2007.

[14] D. Lo, S.-C. Khoo, and C. Liu. Mining past-time temporal rules from execution traces. In *WODA '08*, pages 50–56, 2008.

[15] A. C. Nguyen and S.-C. Khoo. Extracting significant specifications from mining through mutation testing. In *ICFEM '11*, 2011.

[16] A. C. Nguyen and S.-C. Khoo. COPPICE: Discovering complete api rules through mutation testing. Technical Report TRC3/12, Department of Computer Science, National University of Singapore, March 2012.

[17] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *ASE '09*, pages 371–382, 2009.

[18] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *ASE '09*, pages 283–294, 2009.

[19] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *ICSE '09*, pages 496–506, 2009.

[20] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99*, pages 13–, 1999.

[21] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *ICDE '04*, page 79, 2004.

[22] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *ESEC-FSE '07*, pages 35–44, 2007.

[23] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. In *ICSE '06*, pages 282–291, 2006.