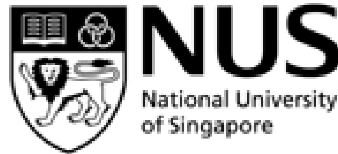
T H E   N A T I O N A L   U N I V E R S I T Y
o f   S I N G A P O R E



S c h o o l   o f   C o m p u t i n g
Computing 1, 13 Computing Drive, Singapore 117417

TRA7/11

Extracting Significant Specifications from Mining through Mutation Testing (Technical Report)

Anh Cuong Nguyen and Siau-Cheng Khoo

July 2011

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

Extracting Significant Specifications from Mining through Mutation Testing (Technical Report)

Anh Cuong Nguyen and Siau-Cheng Khoo

Department of Computer Science, National University of Singapore
{anhcuong, khoosc}@comp.nus.edu.sg
7 July 2011

Abstract. Specification mining techniques are used to automatically infer interaction specifications among objects in the format of call sequences, but many of these specifications can be meaningless or insignificant. As a consequence, when used in program testing or formal verification, the presence of these leads to false positive defects, which in turn demand much effort for manual investigation. We propose a novel process for determining and extracting significant specifications from a set of mined specifications using *mutation testing*. The resulting specifications can then be used with program verification to detect defects with high accuracy. To our knowledge, this is the first fully automatic approach for extracting significant specifications from mining using program testing. We evaluate our approach through mining significant specifications for the Java API and use them to find real defects in many systems.

Keywords: Specification mining, mutation testing, formal specifications.

1 Introduction

Specification mining is a process that enables the inference of candidate interaction protocols between objects in a program from its execution traces or source code. One important type of these specifications is the class of temporal logic properties over function or method call sequences. These specifications can be efficiently used to describe interesting 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)`."

Specifications reflecting legitimate usage protocols can be used in program testing or formal verification to detect defects in systems. A system is said to have defects if it does not respect one or more legitimate protocols. A common issue pertaining to specification mining approach, however, is that there are typically many meaningless and insignificant specifications discovered by the mining process. As a consequence, when insignificant specifications are used for detecting defects, they usually lead to false reports and demand expensive effort for manual investigation. To illustrate this problem, consider the experiment done

by Wasylkowski et al. for detecting anomalies in AspectJ, a compiler for the AspectJ language, using *object usage model* mined from a specification miner called JADET [18]. Among 276 anomalies and 790 violations detected, only 7 violations from 6 anomalies lead to real system defects. Aside from Wasylkowski experiment, many other experiments also show that verify systems against specifications obtained from mining may lead to many violations, but only a few of them actually associated with real defects. [16, 12, 10].

How can one mine specifications against violations of which can lead to real defects rather than false ones? In this paper, we propose a novel technique that employs *mutation tests* to determine significant specifications from a set of mined specifications. The resulting specifications can then be used in program verification to detect defects with high precision.

Specifically, given a mined specification that attempts to describe a behavior of a method, we perform a mutation operation on the method body so that it intentionally violates the given specification. We then execute the method with specific input to determine if such a violation of specification will lead to exception being thrown at appropriate program points. When that happens, we deem the mined specification to be significant. Otherwise, we deem it insignificant.

We have implemented a prototype to test our proposal on the Java API, and discovered significant specifications with 100% of precision and at least 80% of recall. We then supplied the resulting significant specifications to a model checker to help find real defects. Our experiment shows that specifications deemed as significant can drastically expedite the discovery of real defects in programs that use the corresponding API.

Our main contributions are as follows:

1. We put the definition of *specification significance* on a firmer theoretical ground. We assess the significance of a specification by its ability to exhibit bad system behaviours when it is violated (Sec. 2). To our knowledge, this is the first work looking into this issue.
2. We introduce a novel approach for determining significant specifications using *mutation testing*.
3. We demonstrate the effectiveness of this approach by implementing a prototype to test on Java API.

The outline of this paper is as follows: In Section 2, we provide an overview of specification mining and formal representation of specification. We then provide a theoretical formulation of the notion of "significant specification" in Section 3. This is followed by a detailed description of our "mutation test" approach (Section 4) and an experiment on its effectiveness (Section 5). Finally, we discuss related work in Section 6 before concluding in Section 7.

2 Background

We begin with a brief overview on how specification mining can produce an insignificant specification and then detect false defects from it. Suppose that we

want to mine *call sequence usages* of the **Stack** object in each method body, using either their execution traces or source codes. Consider the following code snippet, which is taken from APACHE FOP, 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  }
```

The execution flow from line 4 to line 8 in the above snippet introduces a specification stating that a stack must be peeked before it is popped: `pop` $\hookrightarrow$ `peek`[1]. In addition, lines 11 and 12 introduce another specification stating that a stack must be checked for its emptiness before it can be popped: `pop` $\hookrightarrow$ `empty`. We shall refer to these two rules as $\mathbf{R_1}$ and $\mathbf{R_2}$, in that respective order.

Intuitively, $\mathbf{R_2}$ is a more preferable candidate for the process of checking and finding defects. Still, a closer analysis on the code reveals that $\mathbf{R_1}$ is also a meaningful specification in the current context. It can be interpreted that a stack should check (by peeking) for the existence of an object before it actually executes a pop operation. However, this usage protocol is not universally required for well-behaved stack objects. Different from $\mathbf{R_1}$, $\mathbf{R_2}$ states a rule that a stack should strictly follow. A precondition for a stack to perform a pop is that the object stack must be at a non-empty state. Therefore a call to `empty` checks for the legal state of stack object before it can perform a pop. Without checking the object stack for emptiness, the program may crash when trying to perform a pop operation. For this reason, defects found by a violation to $\mathbf{R_2}$ are likely to be true defects, whereas defects found by a violation to $\mathbf{R_1}$ are likely to be false ones.

The specification miner is unfortunately clueless about this intuition. Currently, a common solution to this problem is to determine the specification significance based on the notion of confidence and support, which are linked to the number of occurrences of the pattern in the given sequences. However, this solution can be ineffective in many cases because an insignificant rule may have a high occurrence. For example, $\mathbf{R_1}$ can have a very high occurrence if the code accidentally uses the rule frequently throughout an execution (which is indeed the case for FOP where our experiment discovered 122 occurrences of $\mathbf{R_1}$.)

[1] Section 2.1 explains the notation in detail.

In this paper, we propose a novel technique to ascertain the significance of system specifications: A specification is deemed significant when violating the specification during program execution can make the participating objects misbehave, which lead to bad system behavior, exhibited by a system crash. It should be noted that our definition of significance centers on the behavior of the participating objects found in the specification, and referenced in the code.

For the remaining of this section, we describe the type of system specification we use throughout the paper (Sec. 2.1). In Sec. 3, we formalize various definitions of specification significance and give an overview of our solution for identifying significant specifications (Sec. 3.1).

2.1 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" [14]. Specifications of this format are commonly found in practice and useful for program testing and verification. Indeed in our empirical study with the Java API, we found a large number of significant rules obeying past-time temporal logic, for example, `next` $\hookrightarrow$ `hasNext`, `pop` $\hookrightarrow$ `empty` or `get` $\hookrightarrow$ `size`.

In our work, past-time temporal specifications discovered is always constructed from two components, a *consequence* and the *premise*, each consists of a series of events. A specification is denoted as `consequence` $\hookrightarrow$ `premise` and states that whenever a series of consequence events occurs it must be preceded by another series of premise events. Each past-time temporal specification can be mapped to its corresponding Linear-time Temporal Logic (LTL) expression. Examples of such correspondences are shown in Table 1, which we borrow from [14]. In addition, all past-time temporal specification in our work must obey

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

Table 1. Specifications and their Past-time LTL Equivalences

another condition, which is all events in the specification must be method calls coming from the same object. This condition is not strictly required for our technique for detecting significant specifications, but it is appropriate for a preliminary study. A technique that handles specifications across multiple objects can be extended straightforwardly from ours.

Finally, a violation of a past-time temporal specification can happen when one or more events in the premise are missing while all events in the consequence occur in the correct order. We use the notation $\overset{v}{\hookrightarrow}$ to symbolize violation. Examples of violations of specifications used in Table 1 are shown in Table 2.

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

Table 2. Specification Violations and their Past-time LTL Equivalences

3 Specification Significance

To have a sense of how specification significance can be defined, let's consider the following scenario. Suppose that a programmer makes use of a new API library and she must follow some call sequence usage specifications specified by the library, which are sometimes documented, sometimes not. She would be prone to make mistake by not obeying some of the usage specifications. When this happens, one or more objects defined in the specification will behave wrongly, and this leads to bad program behaviours.

The definition of specification significance simulates this real life scenario. Firstly, when determining the significance, we look at three components that make of a specification: the *method calls* that make of the event series, the *participating objects* and the relevant *code* that the specification resides in. Then the significance of a specification can be viewed as its ability to cause a participating object to misbehave when the call series does not occur correctly, and that leads to bad code behaviours. Particularly in this work, we consider an object misbehaviour as a thrown exception, and a bad behaviour as a code crash due to the exception. We will discuss further in Sec. 7 how we can determine other kinds of bad code behaviour by incorporating more code checking systems. Concretely we define 3 levels of specification significance as follows, of which the latter two are goals attainable by our method.

Usage Significance. Rules belonging to this set are meaningful and important for use in some specific contexts. An example is `File.delete()` $\hookrightarrow$ `File.isDirectory()`, which is used to delete system directories. Violations of these rules are though harmful in some specific usages, they are safe in other usages. Therefore when these rules are used for testing software, their violations may lead to many false positive defects. We do not deal with these rules here.

Object Significance. Rules belonging to this set are not only important for the usage code but also important for the object participating in the usage. An example is `Stack.pop()` $\hookrightarrow$ `Stack.empty()`. When these rules are violated, some calls in the rule will likely throw exceptions. The rule premise can be seen as a check for the object state validity, before the object can use a call in the

rule consequence. Therefore if the check fails (the object state is invalid) but the object still uses a call in consequence, the call will trigger an exception.

Definition 1 (Object significance). *A specification* Spec *is object significant if and only if there exist a code* 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 consequent component of* Spec*.*

$$\texttt{Spec}.\mathit{sig}() \Leftrightarrow \exists\ \texttt{C},\texttt{I}\ \exists\ \texttt{c} \in \texttt{Spec}.\mathit{cons}: \big(\texttt{C}.\mathit{exec}(I) \wedge \texttt{Spec}.\mathit{violate}()\big) \wedge \lightning@\texttt{c}$$

The notation $\lightning@\texttt{c}$ denotes throwing of an exception at call c.

Is one code crash sufficient to determine the object significance of a specification? Generally, not all programs that violate an object significant specification will lead to crashes. For example, one may not need to call `Stack.empty()` before `Stack.pop()` if she knows that the stack is not empty. Nevertheless, we found that the definition using the existence of at least one code crash is useful enough to detect object significant specifications in practice.

Universe Significance. Rules belonging to this set are dictators for every code. An example is `InputStream.reset()` $\hookrightarrow$ `InputStream.mark(int)`, which states that the `InputStream` need to be marked before "repositioning" can be performed on the input stream to the most previous mark (by calling `reset`). The rule premise in this case is meant to bring the object to a valid state for using a call in the rule consequence. Violation of these rules will most likely cause exceptions (unless, for example, the call in premise is replaced by its definition).

Definition 2 (Universe significance). *A specification* Spec *is universe significant if and only if for all code* C *and its input* I*, when* C *is executed with* I *and the specification* Spec *is violated, the code will crash by an exception thrown by a call* c *occurred in consequent component of* Spec*.*

$$\texttt{Spec}.\mathit{sig}() \Leftrightarrow \forall\ \texttt{C},\texttt{I}\ \forall\ \texttt{c} \in \texttt{Spec}.\mathit{cons}: \big(\texttt{C}.\mathit{exec}(\texttt{I}) \wedge \texttt{Spec}.\mathit{violate}()\big) \rightarrow \lightning@\texttt{c}$$

3.1 Identifying Significant Specifications

In this work we introduce a method for identifying object and universe significant specifications in an API library. An overview of our method is shown in Fig. 1. Concretely the method works as follows.

- In the first step, the API-client (back-end software) is fetched into a past-time temporal specification miner. The miner returns a set of raw specifications of the API, which contains both significant and insignificant specifications.
- For each specification, we simulate the specification violation by *suppressing* one or more calls occurring in its premise; the suppression is achieved by creating mutated programs. We use codes of the API-clients as candidate programs for mutation.

– Mutated programs are executed. If one of the programs crashes by an exception caused by a call in the specification consequence component, we deemed the corresponding specification to be significant, otherwise it is insignificant. Finally we collect all significant specifications and output to the user.

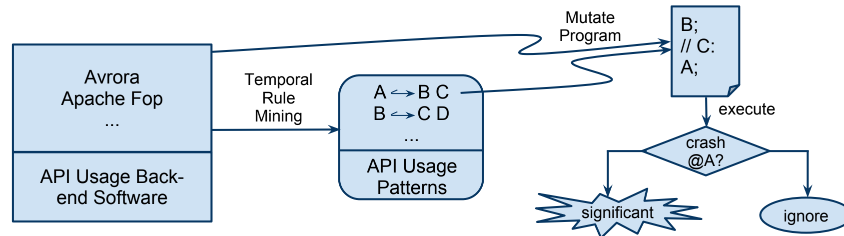


Fig. 1. Extract Significant Specifications from an API Library

4 Mutation Testing

We are now ready to discuss the realization of our mutation testing technique as a practical and effective tool for extracting significant specifications. Our goal is to express opinions about which design choices we prefer and discuss both advantages and shortcomings of these decisions.

We begin by employing a temporal specification miner called LM [6]. LM takes in a set of execution traces. Based on the user-specified minimum thresholds of *support* and *confidence*, LM generates a set of past-time temporal specifications, such as the example below:

```
1  <org/apache/fop/fo/XMLWhiteSpaceHandler.handleWhiteSpace(Lorg/apache/fop/
       fo/FObjMixed;Lorg/apache/fop/fo/FONode;Lorg/apache/fop/fo/FONode;)V
       :97,106 ...>
2  (122, 1.0)
3  PREMISE:
4          java/util/Stack.empty()Z
5  CONSEQUENCE:
6          java/util/Stack.pop()Ljava/lang/Object;
```

The specification states that any `Stack` object must follow the rule `pop` $\hookrightarrow$ `empty`. The rule has a support of 122 (instances from the collection of traces) and a full confidence of 1.0, and one instance of the rule occurs inside the method body of `XMLWhiteSpaceHandler.handleWhiteSpace()` at source line 97 and 106. We call the method body *a container of the specification*.

4.1 Simulating Specification Violation through Program Mutation

We simulate a specification violation by mutating the original program. We use containers of the specification as candidate codes for injecting mutations. The idea behind mutation is to suppress specific calls in the containers that also appear in the specification premise, so that when the container is executed, it causes the program to crash due to violation of the specification.

There are several challenging design issues in program mutation:

Issue 1 Each past-time temporal specification can be violated in many ways. A challenge here is to efficiently represent and manipulate all these violations.

Issue 2 Method call under suppression may occur multiple times in a container. Which occurrence should we suppress to effectively violate the specification?

Issue 3 We mutate the suppressed call by replacing it with an object of identical return type to ensure smooth running of the code. There can be many candidate objects for replacement, which one should we use?

Generally a specification premise can contain several calls. However, an error may occur only when a particular combination of calls are suppressed. For example, the specification `pop` $\hookrightarrow \langle$`empty peek`$\rangle$ about a `Stack` object only yields error at `pop` when `empty` is missing (suppressing `peek` yields no errors). Simulating violation by suppressing all combinations of calls in the premise can be expensive: we need to generate $2^n - 1$ mutated programs when a premise contains n calls. This complexity can be mitigated in two ways. First, we check only *closed temporal rules*[14], which are intuitively rules with "maximal" length. Second, we mutate the container of a closed rule by simultaneously suppressing all calls in the premise. For example, for all three rules `pop` $\hookrightarrow \langle$`empty peek`$\rangle$, `pop` $\hookrightarrow$ `peek` and `pop` $\hookrightarrow$ `empty` (the first rule is closed while the other two can be *subsumed* by the first), we need only one check by suppressing both `empty` and `peek` at the same time. Intuitively, if the closed rule is detected as insignificant (call to `pop` does not yield error), we can safely claim that the rule itself and all subsumed rules are insignificant. On the other hand, if the closed rule is detected as significant, the rule itself or some of its subsumed rules may also be significant (here `pop` $\hookrightarrow$ `empty` is also significant). Later when we have detected a succinct set of significant rules, we can analyze them further to see which subsumed rules are also significant. This will be studied in Sec. 4.4.

For design issues 2 and 3, we illustrate our design decisions through an example. Consider the specification given in the beginning of Sec. 4 and its container shown in Sec. 2. Suppose that we want to simulate a violation `pop` $\hookrightarrow \neg$`empty`. Following is a suitable mutation for the container.

```
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 (!false) % Replaces (!nestedBlockStack.empty())
12           nestedBlockStack.pop();
13       ...
14 }
```

We mutate the container by replacing the call `nestedBlockStack.empty()` at line 11 to a concrete value `false`. This example exposes two problems: the call under suppression (e.g. `empty`) can appear many times (issue 2) and it can be replaced by many concrete values (e.g. `true` or `false`) (issue 3).

We address the second issue by relaxing the definition of violation in the following way: a specification is violated when one (rather than all) of its instances is violated. Thus, when there are many candidate calls for suppression, we only need to suppress those calls that appear in the specification instances. In case of LM, the specification instances are *iterative patterns* [15], which can be uniquely identified from the traces. It becomes natural to use instances to determine a specification violation. In our example above, assuming that line 8 is not executed, the only iterative pattern instance that supports the specification is `empty@11 pop@12`. This instance is thus subject to mutation.

Furthermore, when the mutated code crashes due to an exception, we check whether the exception is raised by a call that is (i) defined in specification consequence, and (ii) part of the instance used for mutation. Only when both conditions are satisfied can we conclude that the specification is significant. In our example above, we need to check that the exception is raised by `pop@12` but not by other occurrences of `pop`.

Finally, a suppressed call can be replaced by many different concrete values, and some might be more suitable than others. For example, in the mutation above, it is more suitable to use `false` as a replacement value because using `true` will make the call `pop` at line `12` non-executable. Ideally one would want to try all possible replacement values, but this is infeasible in practice. In our implementation, we try to simulate as many values as possible using randomly created objects. We discuss the mechanism for creating objects further in Sec. 4.2. Quantitatively, we create one mutation for one replacement object. Thus, if a specification has i instances, its premise contains n calls and each call has maximum r replacement objects, we generate at most ir mutated programs. In Sec. 4.3 we described how the number of mutated programs can be further reduced, thus improving the performance of our technique.

We implement our mutation technique for Java language to work on Java bytecode programs. Bytecode manipulations are done using the ASM bytecode manipulation framework [3]. We omit the details due to space constraint. The algorithm is two phases.

– **Phase 1:** we generate inputs for the *class manipulation adapter* used in phase 2. Each input states which calls and which occurrences of calls the adapter need to suppress; and for each suppression, which replacement values the adapter should use. We also generate expected output for each input. Each output states which calls and which occurrences of calls are expected to raise exceptions.
– **Phase 2:** the class manipulation adapter visits each method container and uses the input generated from phase 1 as guidances for mutating the container. Finally, each mutated class is loaded and runs together with the original system. All exceptions that raised by a call defined in the expected output are recorded. As long as the number of exceptions is greater or equal to 1, we conclude that the specification under checking is significant.

4.2 Generating Replacement Objects

Our mutation technique requires replacing the suppressed call by an object of its return type (except for `void`, which we simply skip the call). We generate these objects by first creating an *object pool* for each object type. An object pool is a set of bytecode instruction sequences that generate objects on the fly. For each type, the content of object pool is randomly created. Generation mechanism can be characterized by three types of input object: primitive types, classes with public constructors and arrays.

We partition the object pool for any primitive type with wide value range into distinct pre-defined sets. For `int`, these are: large negative, small negative, 0, small positive and large positive numbers. For each value range, we only create a small number of objects. Bytecode instructions are used to construct objects within each value range of the corresponding primitive type, as shown in Table 3.

Primitive Type	Bytecode Instructions	Value Ranges
`byte`, `short`, `int`	`{bipush $val}`	`[MIN,-100];[-50,-1];0;` `[1,50];[100,MAX]`
`long`, `float`, `double`	`{ldc $val}`	`[MIN,-100.0];[-50.0,-1.0];0.0` `[1.0,50.0];[100.0,MAX]`
`boolean`	`{iconst_0}`, `{iconst_1}`	`false; true`
`char`	`{bipush $val}`	`[a-z]; [A-Z]; [0-9]; [^a-zA-Z0-9]`

Table 3. Object Pool and Value Ranges for Primitive Type

Object pool of a class that has public constructors is generated inductively and component-wise using object pool of the primitive types. In order to prevent the inductive generation from going into a loop, when initializing an object, we choose constructors that do not involve any *parent objects* of the current object or the object itself as parameters.

For object pool of array type, we firstly assign a random size to each array dimension. Elements of the array are either a primitive type or a normal class and can be randomly generated as usual.

One shortcoming of this approach is that we cannot generate any objects of classes that do not have public constructors, or constructors that require parameters that cannot be generated, as well as array of these classes. These cases rarely occur in our experiment. When they happen, we mark the specification for manual review.

4.3 Exploring Early Pruning

The complexity of our algorithm depends on the run time of containers and the number of mutations under inspection. To reduce the number of mutations and the containers' run time, we utilize two effective pruning strategies.

Duplicated instances avoidance. For two specifications $\mathbf{R_1}$ and $\mathbf{R_2}$ that have the same premise but $\mathbf{R_1}$'s consequence is a super-sequence of $\mathbf{R_2}$, we always check $\mathbf{R_1}$ first. To illustrate this, consider an example with two specifications pertaining to `Rectangle2D` object which are mined from Apache Fop, $\mathbf{R_1} = \langle$ `getWidth, getY, getHeight` $\rangle \hookrightarrow$ `getX` and $\mathbf{R_2} =$ `getY` $\hookrightarrow$ `getX`. Set of instances for $\mathbf{R_1}$ is `{transfrom:22,22,22,22}` while for $\mathbf{R_2}$ is `{transfrom:22,22; generate:56,56}`. Since the instance for $\mathbf{R_1}$ also re-occurs as one for $\mathbf{R_2}$, we remember the instance and only check it once, at rule $\mathbf{R_1}$.

Early termination. Figure 2 shows how a container is being mutated. Aside from suppressing calls in rule premise, we also insert try-catch block around every call in the rule consequence. Each try-catch block does not only record the specification as significant, but also forces early termination of the execution once its significance is determined. In addition, we also force termination of execution (by inserting `System.exit(0)`) when the container exits, if all calls in the instance under check have been executed.

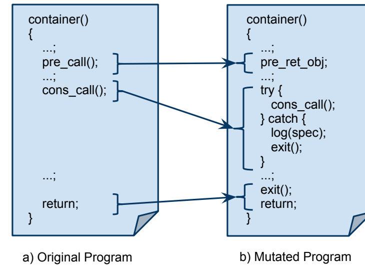


Fig. 2. Mutation of a Container

4.4 Specification Refinement

We show here how the significance of the temporal rules identified by our method can be further improved, leading to a reduction in the number of false positives when used in software testing and/or verification.

Disjunctive Premise. It is common for a call sequence to be preceded by multiple choices of premises. Therefore, given two specifications that have same consequence but different premises, we combine them using disjunction. For example we combine two rules `pop` $\hookrightarrow$ `empty` and `pop` $\hookrightarrow$ `push` into a single, but more precise rule `pop` $\hookrightarrow$ `empty|push`.

Conjunctive Premise. For specifications with multiple calls in the premise, some of these calls may not be required for the consequence, and thus may be dropped. Given such a specification, we re-apply the check on the specification, by suppressing each call in the premise one by one. If a call suppression does not lead to throwing of exceptions, we can safely remove it. For example with the rule `pop` $\hookrightarrow \langle$ `peek, empty` $\rangle$, we can remove `peek` to obtain the rule `pop` $\hookrightarrow$ `empty`.

5 Evaluation

We have built a system described in this paper as a plug-in, called SpecCheck, for the Lm miner. Fig. 3 depicts its system architecture. SpecCheck plugin is implemented as a Java agent that would be invoked by the Java Virtual Machine during load time of the target application. We evaluated SpecCheck on the

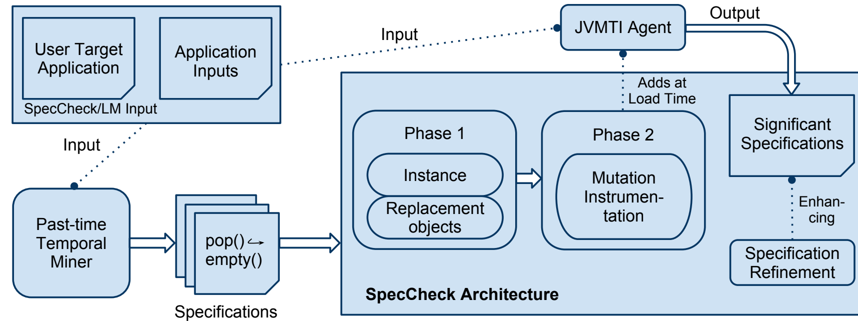


Fig. 3. SpecCheck High-level Architecture

DaCapo 2009 benchmarks [2] to address the following concerns:

– What proportion of the specifications labelled by SpecCheck as significant are indeed significant (precision)?
– What proportion of significant specifications has been labelled by SpecCheck as such (recall)?
– How efficient are the significant specifications, compared to the original specifications, in detecting system defects?

DaCapo 2009 benchmark suite consists of 14 complex Java systems that range over a diverse set of application domains. We select from DaCapo 7

single-thread driven benchmarks to evaluate the effectiveness of SpecCheck (see Table 4). We use the test harness provided by the benchmark suite and perform evaluation for traces of all calls to the Java API library (Sec. 5.1). Finally we evaluate the effectiveness of extracted rules in finding defects in Sec. 5.1.

5.1 Java API Rules

Mining Setup. The set of traces for each Java class varies pretty much in size and some can be very large due to code looping. We keep the traces as they are but use different support and confidence threshold settings for each trace depending on its size. Concretely we set support and confidence thresholds to 5 and 40% respectively for trace files less than 100KB, 50 and 40% respectively for trace files less than 1MB, 100 and 60% respectively for trace files less than 5MB, and ignore all trace files larger than that. Using these settings, the mining process can finish within minutes.

Evaluation Results. Throughout the suite, Lm infers a large number of specifications, among them an average of 6% (38/633) are determined by SpecCheck as significant (Table 4). The ratio of specifications marked as significant to all mined specifications is small, but this is consistent with many past studies that the number of mined specifications which lead to real defects are normally small [16, 12, 10].

Benchmark Project	Mined Spec Considered	Significant Spec Identified	Precision	Recall	Spec Discarded
avrora	28	3	100%	100%	89.2%
batik	369	16	100%	80.0%	95.3%
eclipse	145	10	100%	100%	93.1%
fop	132	11	100%	100%	91.6%
jython	193	14	100%	87.5%	93.2%
luindex	34	4	100%	100%	88.2%
pmd	61	9	100%	100%	85.2%
Total	633	38	100%	86.0%	93.9%
After Spec Refinement:		**No. of Object Spec : 21**		**No. of Universe Spec : 1**	

Table 4. Significant Java API Rules Extracted from DaCapo 2009 Benchmarking Suite

We end up with 22 significant specifications after specification refinement step: 21 of them are object significant and 1 of them is universe significant.

To assess the precision and recall of SpecCheck, we employed three programmers to independently and manually extract significant rules from mined rules. Based on this result, precision and recall achieved by SpecCheck are very encouraging, as depicted in Table 4.

In the assessment of "recall", we have missed a few significant rules (6/44), which correctly reflect some limitations of our mutation techniques. Firstly we may not have generated the correct replacement objects which can cause exceptions. Specifically, we miss the rule `createWritableTranslatedChild` $\hookrightarrow$ `getWidth getHeight` of `WritableRaster` class. To cause an exception in this rule, we need to replace `getWidth/Height` with a very big integer value (e.g. `MAX_VALUE`), which is apparently hard to achieve by generating the value randomly. Secondly we may never reach a designated consequence call which we would like to check during execution, as the program threw exception earlier than expected. We miss three rules of the form `setPixels` $\hookrightarrow$ `getMinX getMinY getWidth getHeigth` of `WritableRaster` class. During the execution of the container methods, `setPixels` is always preceeded by a call to `getPixels`. Unfortunately, by suppressing the call sequence in the premise, we cause `getPixels` to throw exception before the call `setPixels` can ever be reached. Finally we may miss some significant rules that produce errors, but without throwing the anticipated exceptions (e.g. null pointer creation, memory leak, *etc.*)

A sampling of significant specifications is displayed in Table 5. The sample shows that we can detect rules whose significance *cannot* be trivially judged by looking at name similarities (e.g. `pop` $\hookrightarrow$ `empty`). We can also detect rules of length greater than 2 (e.g. `getPixels` $\hookrightarrow$ $\langle$ `getMinX, getMinY, getHeight` $\rangle$). Finally we find one universe significant rule. It is `reset` $\hookrightarrow$ `mark`, which has been explained in Sec. 3.

Performance. Without applying any optimizations described in Sec. 4.3, the checking step does not complete even after several hours for most benchmarks.[2] This is expected, for example in ECLIPSE case, where a single run of the application can take up to 2 minutes. The dataset consists of 145 specifications for checking, each specification has at least two instances and each instance generates at least two mutated programs. The number of mutated programs can be up to 580 and running all of them takes approximately 1160 minutes (which is 19 hours!). Fortunately, with the pruning techniques described in Sec. 4.3, checking of specifications mined from ECLIPSE took only around 15 minutes (6 seconds per rule in average). This confirms that our pruning strategy is necessary and efficient.

5.2 Verification using Java API Rules

In order to ascertain that the specifications we have identified are indeed significant, we compare their ability to detect defects or code smells against those we have classified as insignificant. We only choose those insignificant specifications with high confidences ($> 80\%$). There are 368 of them in total. We employ JFTA [4] to perform static verification.

[2] We conduct the experiment in an Intel Core i5 M460 computer with 2GB memory running Linux Mint 10 Julia.

Java Class	Specification
	Object Signifiance
FlatteningPathIterator	currentSegment([F)I ↪ isDone()Z
Raster	getPixels(IIII[I)[I ↪ getMinX()I getWidth()I| getMinX()I getMinY()I getHeight()I
ByteBuffer	get()B ↪ hasRemaining()Z
Stack	pop()Object ↪ empty()Z|push(Object)Object
LinkedList	get(I)Object ↪ size()I|add(Object)Z
String	substring(II)String ↪ length()I|indexOf(C)I
	Universe Significance
InputStream	reset()V ↪ mark(I)V

Table 5. A Sampling of Java API Significant Specifications from the DaCapo suite.

		Significant Specs			Insignificant Specs		
Benchmark	**Classes**	**Anomalies**	**True**	**False**	**Anomalies**	**True**	**False**
avrora	1838	3	2	2	19	0	61
batik	2430	3	2	1	34	0	154
eclipse	527	3	1	2	22	0	108
fop	1314	2	2	0	11	0	57
jython	2816	1	0	1	24	0	103
luindex	536	1	1	2	10	0	62
pmd	727	2	5	3	13	0	68
Total	9652	7	13	11	60	0	613

Table 6. Significant Specifications Show More True Positives and Less False Positives.

For those identified significant specifications, running JFTA over the seven DACAPO benchmarks reported only a few violations; many of them were true positives. This is in line with our hypothesis that the specifications are significant. On the other hand, verifying using insignificant specifications reported a large number of anomalies and violations (nearly 1000 violations for most projects). However, we observed that majority of violations came from certain Java API classes (e.g. `StringBuilder` and rules `toString` ↪ `append`). Thus we only selected and reported the maximum 10 violations of different anomalies from each API class in case of insignificant specifications. Nevertheless, these specifications still produced much more error reports compared to those produced by significant specifications. Table 6 shows the result of our comparison. It reports the number of anomalies (ie., violation of specifications), true and false code smells (codes that indicate something may go wrong [18]) or defects. (Recall that one anomaly may consist of several defects, each in a different method.)

We inspected all violations manually and reported either defects or code smells as true positives and the rest as false positives. When we are not sure if a violation is a defect/code smell, we conservatively count it as false positive. We also do not consider violations that are already handled directly by a try-catch

block in the code. Aside from keeping the amount of false positives small, we are also interested in finding real defects.

```
1  public synchronized void
       accumLogExit( String scope) {
2    ...
3    AccumPerfScope then = (
         AccumPerfScope) scopeStack.
         pop();
4    if (then == null)
5          System.err.println("Accum
                Perf Error: Scope stack
                empty: " + scope);
6    ...
7  }
```

Fig. 4. A Defect in DACAPO ECLIPSE.

```
1  private void printSectionHeader
       (..., String line) {
2    ...
3    StringTokenizer st = new
         StringTokenizer(line);
4    st.nextToken();
5    ...
6  }
```

Fig. 5. Code Smell in DACAPO AVRORA.

Aside from keeping the amount of false positives small, we are also interested in finding real defects. Among six projects, 57% of error reports uncover real defects or code smells, which is a considerably high rate for a specification-based bug detection tool (e.g. compared to [10, 18]). Most of them are code smells, which are codes that are obviously unsafe which may lead to something wrong within the project. In addition, we are able to find one defect in ECLIPSE project, which is shown in Fig. 4. The developer mistakenly used null to check for stack emptiness. After two years of existence, the file was later removed from ECLIPSE code base in 2008. An example of code smell is also given in Fig. 5. Argument `line` is erroneously used in program line 3 without checking for nullness.

6 Related Work

The integration of *data mining* into *software engineering* has attracted much interest in the past decade. The field is dynamically evolving. Many projects on specification mining produce either an automation [1, 13] or frequent patterns of software behaviours [15, 12]. PERRACOTTA [19] mines two-event temporal logic rules that match a given template, which is later extended by JAVERT to mine more complex specifications [7]. On mining object-oriented behaviours, JADET and ADABU respectively use static and dynamic analysis to mine intra-procedural object usage models [18, 5]. Finally, the specification miner LM we used mines *live sequence charts*, inter-object behaviours of arbitrary sizes [6].

Specification mining can infer many specifications but many of them can be meaningless or irrelevant. This limitation creates a big hurdle for introducing specification mining into real practices. Many interesting research has been

proposed to mitigate this hurdle. Thummalapenta and Xie opine that rules exhibiting exceptional conditions can lead to discovery of real defects, and propose to mine exception-handling rules [16]. OCD automatically learns, enforces and determines anomalies using online statistical learning [8]. JADET, and later CHECKMYCODE, determine anomalies based on various heuristic ranking [18, 10]. Goues and Weimer also introduce a method to mine specifications with few false positives. Their method uses software artifacts like repository and source code to select only input traces with acceptable trustworthiness metrics [9]. Our method inherits advantages from all cited methods: we also mine specifications and detect anomalies with few false positives, allow arbitrary temporal specifications and only require the software itself as input. Finally Dallmeier et al. introduce TAUTOKO tool, which also performs mutation operations, but for the purpose of test case generation [4].

7 Conclusion

We have presented the first mutation testing-based algorithm that identifies significant specifications from a set of mined specifications. We implemented our algorithm in an efficient and practical tool called SPECCHECK. Initial evaluation on DACAPO benchmarks shows encouraging result: we are able to identify significant Java API specifications with high precision and recall; significant specifications can be used to detect defects with high accuracy. In comparison, use of those specifications not identified as significant leads to a large number of false positives.

The evaluation involves two validity threads that we try to mitigate. Firstly the sample of 6 projects in this study may be too small to yield statistically significant results. To reduce this threat, we select 6 matured projects that largely varies in term of application domains. Secondly although we have carefully inspected the source code and do cross-validation in case of determining significant specifications, we are not experts involved in the development and may have some misclassifications.Thus we stay in conservative side when categorizing anomalies, if we are not sure we count them as false positives.

In future we would like to apply our technique to a wider range of projects to gain better statistical results, and to collect significant specifications as a database for bug detection purpose, probably similar to [10]. We also want to detect significance based on program errors aside from exceptions; one way to do this is to incorporate with verification tools such as CORK [11] or JPF [17], which can detect memory leaks and concurrency errors. Finally we are still not sure how to apply our technique for future-time logic (since a violation leads to errors occurring in uncertain points in the future), which is also an interesting direction to work on.

The SPECCHECK source code, and all data and results cited in this article are available at: `http://www.comp.nus.edu.sg/~anhcuong/tools/speccheck.html`.

Acknowledgements We thank Zhao Lin, Shafeeq Ahmed and Quang Huynh for their help on the evaluation of SPECCHECK. We thank the anonymous reviewers

for their valuable feedbacks. Additional thanks go to Hugh Anderson, David Lo, Sandeep Kumar, Chengnian Sun, Narcisa Milea and Zhiqiang Zuo for their inputs and discussions on the preliminary versions for this work. This research is partially supported by the research grants R-252-000-403-112 and R-252-000-318-422.

References

1. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: POPL '02. pp. 4–16 (2002)
2. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: java benchmarking development and analysis. In: OOPSLA '06. pp. 169–190 (2006)
3. Bruneton, E., Lenglet, R., Coupaye, T.: Asm: A code manipulation tool to implement adaptable systems. In: Adaptable and Extensible Component Systems. Grenoble, France (2002), asm.objectweb.org/current/asm-eng.pdf
4. Dallmeier, V., Knopp, N., Mallon, C., Hack, S., Zeller, A.: Generating test cases for specification mining. In: ISSTA '10. pp. 85–96 (2010)
5. Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining object behavior with adabu. In: WODA '06. pp. 17–24 (2006)
6. Doan, T.A., Lo, D., Maoz, S., Khoo, S.C.: Lm: a miner for scenario-based specifications. In: ICSE'10. pp. 319–320 (2010)
7. Gabel, M., Su, Z.: Javert: fully automatic mining of general temporal properties from dynamic traces. In: SIGSOFT '08/FSE-16. pp. 339–349 (2008)
8. Gabel, M., Su, Z.: Online inference and enforcement of temporal properties. In: ICSE '10. pp. 15–24 (2010)
9. Goues, C., Weimer, W.: Specification mining with few false positives. In: TACAS '09. pp. 292–306 (2009)
10. Gruska, N., Wasylkowski, A., Zeller, A.: Learning from 6,000 projects: lightweight cross-project anomaly detection. In: ISSTA '10. pp. 119–130 (2010)
11. Jump, M., McKinley, K.S.: Cork: dynamic memory leak detection for garbage-collected languages. In: POPL '07. pp. 31–38 (2007)
12. Livshits, B., Zimmermann, T.: Dynamine: finding common error patterns by mining software revision histories. In: ESEC/FSE-13. pp. 296–305 (2005)
13. Lo, D., Khoo, S.C.: Smartic: towards building an accurate, robust and scalable specification miner. In: SIGSOFT '06/FSE-14. pp. 265–275 (2006)
14. Lo, D., Khoo, S.C., Liu, C.: Mining past-time temporal rules from execution traces. In: WODA '08. pp. 50–56 (2008)
15. Lo, D., Khoo, S.C., Wong, L.: Non-redundant sequential rules-theory and algorithm. Inf. Syst. 34, 438–453 (June 2009)
16. Thummalapenta, S., Xie, T.: Mining exception-handling rules as sequence association rules. In: ICSE'09. pp. 496–506 (2009)
17. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering 10, 203–232 (2003)
18. Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: ESEC-FSE '07. pp. 35–44 (2007)
19. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal api rules from imperfect traces. In: ICSE '06. pp. 282–291 (2006)