Automatic Testing of Sequential and Concurrent Substitutability

Michael Pradel
Department of Computer Science
ETH Zurich, Switzerland

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

***Abstract*—Languages with inheritance and polymorphism assume that a subclass instance can substitute a superclass instance without causing behavioral differences for clients of the superclass. However, programmers may accidentally create subclasses that are semantically incompatible with their superclasses. Such subclasses lead to bugs, because a programmer may assign a subclass instance to a superclass reference. This paper presents an automatic testing technique to reveal subclasses that cannot safely substitute their superclasses. The key idea is to generate generic tests that analyze the behavior of both the subclass and its superclass. If using the subclass leads to behavior that cannot occur with the superclass, the analysis reports a warning. We find a high percentage of widely used Java classes, including classes from JBoss, Eclipse, and Apache Commons Collections, to be unsafe substitutes for their superclasses: 30% of these classes lead to crashes, and even more have other behavioral differences.**

I. Introduction

A. Substitutability

Classes extend and refine other classes through subclassing. Polymorphism allows a reference of a superclass type to point to a subclass instance. To ensure that using a superclass reference gives the expected behavior, each subclass should behave like the superclass when being used through the superclass type. This requirement is called *substitutability* and can be enforced through behavioral subtyping [1], [2]. We call a subclass that fulfills this property a *safe substitute* for its superclass and an *unsafe substitute* otherwise.

Substitutability is crucial for object-oriented programming. It allows a programmer to reason about the behavior of an object based on the object's statically declared type, that is, without knowing its runtime type. What if a programmer that uses objects of declared type *Super* could not assume subclasses to be safe substitutes? A prudent programmer would have to study all subclasses of *Super*, figure out which of them are compatible with the intended usage, and add runtime type checks before each use of an object with an unknown runtime type. Since this approach would add boilerplate code and runtime overhead, it obviously reduces the benefits of subclassing and polymorphism. To avoid these problems, substitutability is widely accepted as one of the bedrock principles of object-oriented programming.

Despite the importance of substitutability, programmers get little help from existing languages and tools to ensure that subclasses are safe substitutes. Popular languages, such as Java, enforce substitutability only on the type level but not on a deeper, semantic level. While the type system ensures that an overriding method is type-compatible with the overridden method, it does not guarantee that the overriding method semantically substitutes the overridden method. Specification and verification techniques to ensure substitutability through behavioral subtyping have been proposed [2]–[6], but none of them has found its way into mainstream programming. Furthermore, we are not aware of any practical tool that helps programmers find substitutability problems.

```
TreeMap m = new TreeMap() OR new FastTreeMap();
m.put(23, m);
m.pollLastEntry();
m.hashCode();
```

Result:

OK	if `m` is `TreeMap`
`StackOverflowError`	if `m` is `FastTreeMap`

Figure 1. A sequentially used class that is an unsafe substitute.

As a result of this lack of support, programmers are susceptible to creating subclasses that are unsafe substitutes. For example, consider `FastTreeMap` from the Apache Commons Collections library. `FastTreeMap` is a subclass of `TreeMap` and therefore should behave like a `TreeMap` when being referred to as a `TreeMap`. Figure 1 shows a usage of these classes that reveals that `FastTreeMap` is an unsafe substitute for `TreeMap`.[1] Executing the code succeeds if `m` is a `TreeMap` but raises an exception if `m` is a `FastTreeMap`. This difference may crash a client program that expects a `TreeMap` reference to behave like a `TreeMap`. We reported this problem to the developers, who acknowledged our report.[2]

B. Substitutability in Concurrent Programs

Substitutability, which traditionally has been considered in sequential programs, is equally important in concurrent programs. Similar to sequential programs, a subclass instance referenced through a superclass type and shared by multiple concurrent threads should behave like a superclass instance. In particular, subclasses of a thread-safe class must be thread-safe as well. That is, if multiple threads are allowed to call

[1] We use the syntax `new A() OR new B()` to say that either of the constructors can be executed.

[2] See issue 394 in the Commons Collections bug tracker. The class will be removed in the next version of the library.

```
Properties p = new Properties() OR new PropertyMap();
p.setProperty("a", "b");
```

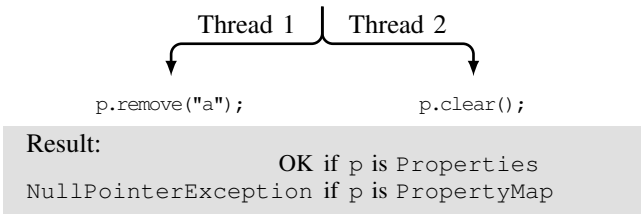

Figure 2. A concurrently used class that is an unsafe substitute.

methods of a shared object without synchronization, then this property must be preserved by subclasses.

Unfortunately, existing languages and tools do not support programmers in ensuring substitutability for thread-safe classes. Figure 2 shows a substitutability problem in `PropertyMap` from JBoss. The class extends `Properties`, a thread-safe class from the Java standard library, and therefore, should also support concurrent usage. However, `PropertyMap` causes an exception in a concurrent usage, whereas the same usage succeeds with the superclass. We reported this problem to the developers, who fixed the bug.[3]

C. Our Approach

How can developers find unsafe substitutes, such as Figures 1 and 2? This paper presents an automatic testing technique that detects subclasses that cannot safely substitute their superclasses. Given two classes *Super* and *Sub*, the technique generates test cases that can test both *Super* instances and *Sub* instances. Then, the analysis uses the behavior of *Super* as an oracle for *Sub* and reports a warning if *Sub* behaves differently from its superclass. We present two variants of this oracle: the *output oracle*, which reports any difference in the visible output, and the *crash oracle*, which reports a warning only if *Sub* leads to a crash that does not occur with *Super*. The analysis checks substitutability for sequentially and concurrently used classes and, among many others, finds the bugs in Figures 1 and 2.

Our analysis is easy to apply, precise, and incomplete. It is easy to apply, because the only input are the classes to test, possibly accompanied by auxiliary classes to create method arguments. The approach is precise, because all reported warnings point to classes that are unsafe substitutes.[4] Furthermore, each warning comes with a concrete test case showing how the subclass behavior differs from the superclass behavior. On the downside, the analysis is incomplete, that is, it cannot guarantee that a subclass is a safe substitute. Incompleteness is the price we pay for an easy to apply and precise analysis.

We apply the analysis to 145 pairs of widely used and well-tested Java classes and have two main findings.

1) We find that surprisingly many subclasses are unsafe substitutes: 30% of the analyzed subclasses lead to crashes that do not occur with their superclass and even more subclasses (42%) produce output that differs from the output of their superclass.
2) We find that developers care about substitutability problems: We reported ten bugs, out of which seven got fixed after a short time. Moreover, at least three other bugs detected by our analysis have been fixed independently of us.

[3]See issue 126 in the JBoss Common bug repository.

[4]Precision is guaranteed only to the degree that constructor mappings (Section III-C) are precise. In practice, the crash oracle (Section III-D2) has no false positives during our evaluation.

D. Contributions

In summary, this paper contributes the following:

- An automatic and precise analysis to find subclasses that cannot safely substitute their superclasses.
- We consider substitutability, which has traditionally been studied in sequential programs, in concurrent programs.
- Empirical evidence that the problem of ensuring substitutability is not yet solved and that our analysis is a step towards solving it.

II. Background: The Perils of Polymorphism

Polymorphism and subclassing are part of most popular, class-based, object-oriented programming languages. This section briefly discusses problems emerging when these two language features interact and describes how substitutability helps to avoid these problems.

Polymorphism allows a reference of type $Super$ to point to an object of any subtype Sub of $Super$. Polymorphic references that point to subtype instances occur in various situations, for example, when a method with declared return type $Super$ returns a Sub instance, when a Sub argument is passed to a method expecting a $Super$ parameter, or when a Sub instance is stored in a field of type $Super$.

In many languages, a subtype is created by creating a subclass and restrictions apply on how the subclass can adapt the superclass. For example, Java imposes type-level restrictions on overriding methods to ensure that an overriding method accepts at least all parameters accepted by the overridden method. Despite these restrictions, a subclass can significantly change the superclass behavior. For example, a subclass may override a method that performs a complex computation with an empty method, or replace a side effect-free method with a method that has side effects.

The ubiquity of polymorphic references combined with the power of subclassing leads to a problem: A subclass Sub that behaves differently from its superclass $Super$ can surprise a client of $Super$ that unconsciously works with an instance of Sub. For example, consider a client that calls a method on a variable with static type $Super$, where the variable is the return value of a call to a third-party library. The client cannot foresee the runtime type of the returned object and decides to call the method based on the variable's static type. If the behavior of the call depends on the object's runtime type, the programmer is caught off guard and may have to deal with a bug that she is not responsible for and that she cannot foresee.

To avoid surprises because of subtype instances hidden by polymorphic references, subclasses should not diverge from

the superclass behavior. This idea, coined as substitutability [1], has become textbook knowledge [7], [8] and is widely accepted by object-oriented programmers.

III. Finding Unsafe Substitutes

Although substitutability is crucial for object-oriented programming, programmers have little support for testing whether a subclass is a safe substitute for a superclass. This section presents an automatic analysis to find subclasses that cannot safely substitute their superclasses.

A. Terminology

In the following, we define when a subclass is a safe substitute for a superclass and describe two ways in which a subclass can be an unsafe substitute. When saying subclass, we mean both direct and indirect subclass relationships. The definitions build upon the notion of a *usage*, which describes how a class is used in a client program. We denote usages that apply to both a superclass and a subclass as *generic usages*.

Definition 1 (Generic usage)**.** *A generic usage w.r.t. classes $Super$ and Sub is a partial order of constructor calls, method calls, and field accesses that use an object of runtime type $Super$ or Sub through the interface offered by $Super$.*

For a generic usage u, we write u_{Super} if the object has runtime type $Super$ and u_{Sub} if the object has runtime type Sub.

A generic usage can describe both sequential and concurrent usages of objects. For a sequential usage, the partial order of calls and field accesses is a total order. For a concurrent usage, the partial order reflects multiple concurrent threads.

Definition 2 (Safe substitute)**.** *A subclass Sub is a safe substitute for a superclass $Super$ if and only if for each generic usage u, the visible behavior of u_{Super} is equivalent to the visible behavior of u_{Sub}.*

By visible behavior we mean application-level behavior, such as return values of method calls, exceptions, text written to the console, and network packets sent to other machines. We ignore machine-level behavior, such as memory accesses and CPU usage. In the following, we define two kinds of unsafe substitutes, which differ in the way the visible behavior of the subclass diverges from the visible behavior of the superclass.

Definition 3 (Output-diverging (unsafe) substitute)**.** *A subclass Sub is an output-diverging substitute for a superclass $Super$ if and only if there exists a generic usage u so that there exists an output of u_{Sub} that is not an output of u_{Super}.*

By output we mean the sequence of return values of method calls on the used object. For sequential and deterministic usages, there exists exactly one output per usage. If the subclass output is different from the superclass output, then the subclass is an output-diverging substitute. For concurrent usages, there can be multiple outputs for a single usage, for example, due to scheduling non-determinism. If the set of subclass outputs contains an output that is not in the set of superclass outputs, then the subclass is an output-diverging substitute.

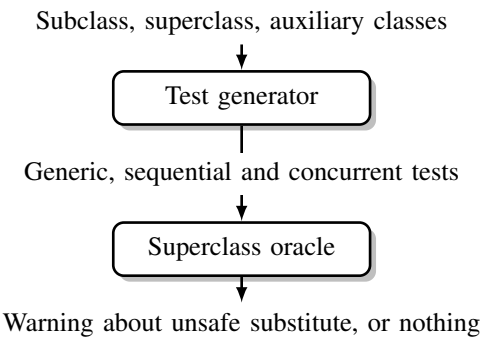

Figure 3. Overview of the approach.

Definition 4 (Crashing (unsafe) substitute)**.** *A subclass Sub is a crashing substitute for a superclass $Super$ if and only if there exists a generic usage u so that u_{Sub} can lead to an uncaught exception or to a deadlock that cannot occur with u_{Super}.*

Of the unsafe substitutes considered here, crashing substitutes are the more severe kind of unsafe substitutes, because they lead to certainly undesired behavior in clients using a Sub instance through type $Super$. Output-diverging substitutes may or may not be a problem in practice (Section V).

B. Overview of the Analysis

How can a programmer check whether a subclass is an unsafe substitute? In the following, we present a fully automatic analysis to reveal subclasses that are unsafe substitutes. The input to our analysis are a subclass Sub and a superclass $Super$, possibly accompanied by classes that the other classes depend on. The output of the analysis are warnings that show with concrete test cases how using a subclass instance through the superclass type leads to visible behavior that is impossible with a superclass instance.

Our approach to reveal unsafe substitutes combines two techniques (Figure 3). First, a *test generator* creates generic tests that exercise sequential and concurrent behavior (Section III-C). The generated tests are generic in the sense that they can test the behavior of Sub as well as the behavior of $Super$. Generic tests simulate how clients of Sub and $Super$ use these classes, that is, they represent generic usages. The test generator considers only public methods, fields, and constructors, because these are accessible to clients.

Second, an analysis called the *superclass oracle* checks for each test whether the test exposes behavior that occurs when using Sub but that cannot occur when using $Super$ (Section III-D). If the superclass oracle detects such behavior, it reports a warning to the developer. By construction, each warning comes with a concrete test case showing how the subclass behavior diverges from the superclass behavior. We present two variants of the superclass oracle that search for output-diverging substitutes (Section III-D1) and crashing substitutes (Section III-D2), respectively.

C. Test Generation

In the following, we present our approach for generating generic tests for sequentially and concurrently used classes. The test generator builds upon a previously presented technique for generating concurrent tests [9]. We adapt the existing technique to generate generic tests and to generate both sequential and concurrent tests.

1) Generic Tests: Tests are assembled from *calls*. A call consists of a method name, a list of input variables, and an optional output variable. For an instance call, the first input variable represents the receiver of the call; the other input variables are the arguments given to the call. If the output variable is defined, it represents the return value of the call. We model constructor calls as calls without a receiver variable and where the output variable refers to the newly created object. Similarly, we model field accesses as calls without arguments and where the output variable represents the value of the field. The calls in a test are centered around a particular object, called the *object under test* (OUT). The OUT is created by one of the calls in a test and subsequently used as the receiver of other calls. Tests can create other objects, for example, to pass them as arguments to methods of the OUT.

To compare $Super$ and Sub, we require tests that can check the behavior of OUTs of both runtime types Sub and $Super$. We call such a test a *generic test*. In a generic test, the static type of the variable v_{OUT}, which represents the OUT, is $Super$. The dynamic type of v_{OUT} can be either $Super$ or Sub. Using the static type $Super$ for the OUT ensures that each call involving the OUT is type-compatible with both $Super$ and Sub. A generic test has two parts. One part creates the OUT and therefore decides on the OUT's runtime type. This part has two variants, one that instantiates $Super$ and one that instantiates Sub. The other part uses the OUT and is independent of the OUT's runtime type.

2) Constructor Mappings: Because we compare the behavior of $Super$ and Sub instances, it is crucial that the constructors used for creating these instances are semantically equivalent. Finding pairs of $Super$ and Sub constructors that create semantically equivalent objects is non-trivial in Java and cannot be fully automated for two reasons. First, constructors are not inherited [10]. For example, Sub may have a single constructor expecting an `int` parameter, while $Super$ offers only the parameterless default constructor. Second, calling a Sub constructor that leads to the same inherited state as calling a $Super$ constructor does not guarantee that the resulting objects are semantically equivalent. For example, a superclass may store a length in meters in a field, whereas the subclass uses the same field to store the length in yards.

To address the problem of finding equivalent constructors, we use *constructor mappings* that specify how calling a constructor of Sub can be transformed into a semantically equivalent call to a constructor of $Super$. We provide a heuristic that automatically generates constructor mappings under the assumption that $Super$ and Sub constructors that expect the same types of arguments are semantically equivalent. Alternatively, a user of the approach can specify constructor mappings explicitly. In the evaluation, we use automatically generated constructor mappings.

For example, consider the following constructor mappings:

$$Sub(Foo, Bar) \rightarrow Super(1, 2)$$

$$Sub(int, boolean) \rightarrow Super(2)$$

The first mapping specifies that calling `new Sub(someFoo, someBar)` is semantically equivalent to calling `new Super(someFoo, someBar)`. In practice, most constructor mappings are similar to the first example, which is why our heuristic generates this mapping automatically. The second mapping specifies that we can replace a call to `new Sub(someInt, someBoolean)` with a call to `new Super(someBoolean)`, that is, we pass the second argument of Sub's constructor as the only argument to $Super$'s constructor and omit the `int` parameter as it is not required by $Super$. If necessary, a user can specify the second mapping in addition to or instead of generated constructor mappings.

3) Generating Sequential and Concurrent Tests: The test generator can be configured to produce either sequential tests, concurrent tests, or both kinds of tests, depending on the kind of usage the tested class is intended for. A *sequential test* is a sequence of calls executed in a single thread. A *concurrent test* is a partially ordered set of calls that are executed in multiple concurrent threads. We focus on concurrent tests that consist of a *sequential prefix* followed by two *parallel suffixes*, where the prefix is a call sequence executed in a single thread, and where each suffix contains a single method call executed in a separate thread that runs concurrently with the other suffix thread. The rationale for this choice is twofold. First, a recent study on real-world concurrency bugs shows that most bugs (96%) can be reproduced with two threads [11]. Second, the execution of concurrent suffixes with a single method call can be efficiently explored and still reveals many real-world bugs [12].

To create a generic test, the first step is to generate calls that create the OUT. The generator selects a constructor mapping and generates two call sequences with calls to instantiate $Super$ and Sub, respectively. These two sequences give two variants of the test, which allows us to test both $Super$ and Sub. The second step is to generate the part of the test that uses the OUT. For a sequential test, the generator repeatedly appends calls to the OUT. For a concurrent test, the generator appends calls to the OUT to obtain the prefix of the test. Then, the generator creates suffixes by spawning two threads and by adding a call to each of them.

Calling methods often requires method arguments. The test generator chooses among three ways to obtain an argument of a particular type: (1) Use an existing variable of the required type, for example, the return value of an existing call in the test; (2) Call a method that returns a value of the required type; (3) For primitive types, randomly create a value. When searching for arguments, the test generator may need further arguments. In this case, it recursively applies the above procedure. If a maximum recursion depth is reached, the test

Input: Generic test t for classes $Super$ and Sub
Output: Warning about an unsafe substitute, or nothing

1: $\mathcal{O}_{Sub} \leftarrow execute(t_{Sub})$
2: $\mathcal{O}_{Super} \leftarrow execute(t_{Super})$
3: $\mathcal{O}_{SubOnly} \leftarrow \mathcal{O}_{Sub} - \mathcal{O}_{Super}$
4: **if** $\mathcal{O}_{SubOnly} \neq \emptyset$ **then**
5: $\quad reportWarning(t, \mathcal{O}_{SubOnly})$

Figure 4. Output oracle. Checks whether Sub is an output-diverging substitute for $Super$.

generator gives up and tries to call a method that requires other types of arguments.

Calls made to obtain arguments for calls in the suffix of a concurrent test are added to the prefix instead of the suffix. For example, to call a method `m(A)` in the suffix, the test generator adds calls to obtain an instance of `A` to the prefix. The rationale for pushing calls back to the prefix is to keep the suffix as short as possible, which in turn, reduces the time for exploring the interleavings of a concurrent test.

Most decisions of the test generator on how to explore the space of possible tests are taken randomly and based on feedback from executions of partial tests [13]. For example, when choosing which constructor or method to call, or how to obtain an argument, the test generator randomly selects from all available options. Whenever a call is added to a partial test, the test generator executes the extended test and checks whether adding the call leads to an exception or to a deadlock. If adding a call leads to a failure, the call is discarded and another call is added instead. For executing tests during test generation, we use the variant of the test that exercises a Sub instance. The reason is that partial tests that fail with Sub can be passed to the crash oracle (Section III-D2), which will check whether the test also fails with $Super$.

4) Examples: Figures 1 and 2 are examples of generic tests for sequentially and concurrently used classes, respectively. The `new A() OR new B()` syntax expresses the two variants of the tests. Depending on which of the two constructors is used, the tests exercise either a superclass instance or a subclass instance.

D. The Superclass Oracle

Given a generic test for two classes $Super$ and Sub, the superclass oracle checks whether the test exposes that Sub is an unsafe substitute for $Super$. The oracle uses $Super$ as an executable specification of the correct behavior of Sub and compares the visible behavior of both classes. The visible behavior of a subclass can diverge in different ways from the visible behavior of a superclass. In the following, we present two variants of the superclass oracle, the output oracle and the crash oracle, which focus on revealing output-diverging substitutes and crashing substitutes, respectively. The input to both oracles is a generic test t. Similar to generic usages, we denote a test working with an OUT of runtime type $Super$ as t_{Super} and likewise for Sub.

1) The Output Oracle: The output oracle checks whether a generic test exposes that Sub is an output-diverging substitute for $Super$. The oracle executes both t_{Super} and t_{Sub} and stores the sequences of return values in the output sets $\mathcal{O}_{Super}$ and $\mathcal{O}_{Sub}$ (Figure 4). Assuming deterministic execution, sequential tests have exactly one output sequence. In contrast, concurrent tests can have multiple output sequences due to scheduling non-determinism. If there exists a sequence of return values that occurs only with Sub, the oracle reports a warning, because Sub is an output-diverging substitute for $Super$. The analysis explores all possible interleavings of concurrent tests to avoid missing an output sequence (Section III-D4).

To compare output sequences of test executions, the oracle transforms the return values into an execution-independent format. For primitive values and Strings, the output sequence contains the actual return value. For reference values, the oracle stores whether the value is `null` or not. For example, the output sequence of executing the sequential test in Figure 1 with an instance of `TreeMap` is the following (values are separated with "—"):

(1) *null — non-null ref. value — int:0*

The first value, *null*, is the return value of `m.put(23,m)`, the second value, a non-null reference value, is the return value of `m.pollLastEntry()`, and the last value, the integer zero, is returned by `m.hashCode()`. Executing the concurrent test in Figure 2 with an instance of `Properties` gives two output sequences:

(2) *null — String:"b"*
(3) *null — null*

Alternatively to this encoding of return values, the output oracle could compare reference values with their `equals()` method, compare the String representations obtained via `toString()`, or compare the hash codes. Unfortunately, with these approaches objects often appear to be different even though they are semantically the same. The reason is that the default implementations of `equals()`, `toString()`, and `hashCode()` refer to object identity. As a result, many of the warnings reported by the output oracle would be spurious, because the difference in output is not a bug.

To avoid false warnings caused by calls to `equals()`, `toString()`, and `hashCode()` in the generated tests, the oracle ignores return values of those methods. With this refinement, the output sequence for Figure 1 with an instance of `TreeMap` is not (1) but the following:

(4) *null — non-null ref. value — ignored*

The output oracle compares the visible behavior of $Super$ and Sub in a strict way. In practice, subclasses often change the return value of methods in ways that are considered to be in line with substitutability (we give examples in Section V-D). Our experiments show that many of the warnings produced by the output oracle are benign differences (Section V-B). In the following, we address this imprecision of the output oracle and present a precise oracle that reports only true bugs.

2) The Crash Oracle: The crash oracle checks whether a test exposes that a subclass is a crashing substitute. Each warning reported by the crash oracle shows how a usage of

Input: Generic test t for classes $Super$ and Sub
Output: Warning about an unsafe substitute, or nothing

1: $\mathcal{F}_{Sub} \leftarrow execute'(t_{Sub})$
2: **if** $\mathcal{F}_{Sub} \neq \emptyset$ **then**
3: $\quad \mathcal{F}_{Super} \leftarrow execute'(t_{Super})$
4: $\quad$ **if** $\mathcal{F}_{Super} = \emptyset$ **then**
5: $\quad\quad reportWarning(t, \mathcal{F}_{Sub})$

Figure 5. Crash oracle. Checks whether Sub is a crashing substitute for $Super$.

the subclass can lead to an uncaught exception or a deadlock in a situation where using the superclass works fine. Since program crashes are certainly undesired behavior, each of the reported warnings points to a severe substitutability problem.

Given a generic test t, the crash oracle first executes it with Sub's constructor (Figure 5). The $execute'()$ function returns a set of failures observed while executing the test. A failure is either an exception or a deadlock. If executing the test with Sub's constructor leads to one or more failures, this does not necessarily mean that Sub is implemented incorrectly. Since tests are generated, they may use the class incorrectly, for example, by passing arguments that cause an `IllegalArgumentException`. To check whether the observed failures are part of the expected behavior of Sub, the oracle executes the test with $Super$'s constructor. If $Super$ also causes a failure, the oracle does not report a warning, because the behavior of the subclass does not diverge from the behavior of the superclass. In contrast, if the set of failures from $Super$ is empty, the oracle reports a warning, because Sub leads to a crash not possible with $Super$.

The crash oracle can be seen as a filter of all generated tests that trigger a failure. With feedback-directed, random test generation, many generated tests crash but only few of them correspond to bugs in the tested code. The crash oracle avoids overwhelming the user with crashing tests by focusing on crashes caused by a substitutability problem.

3) Distinguishing Concurrent from Sequential Problems: We refine the algorithms in Figures 4 and 5 to identify problems that occur only in concurrent tests, and not sequentially. This refinement matters when checking for concurrent substitutability problems without reporting sequential problems.

Before reporting a warning for a concurrent test, the output oracle checks whether the output sequences $\mathcal{O}_{SubOnly}$ are also possible when linearizing the test into a single thread. A linearization of a concurrent test is a sequence of calls that contains all calls from the test while preserving the partial order of calls from the test. Since we generate concurrent tests with two suffixes that each contain a single call, there are exactly two linearizations for each concurrent test: One where the call from thread 1 comes first and one where the call from thread 2 comes first. If all output sequences in $\mathcal{O}_{SubOnly}$ also occur with linearizations of the concurrent test, then the substitutability problem is sequential and not specific to concurrently using Sub and $Super$.

The crash oracle executes the linearizations of a concurrent test before executing t_{Sub}. If the execution of at least one linearization fails, then the oracle does not explore the concurrent execution of the test and instead treats the failing linearization like a sequential test obtained from the test generator.

4) Exploring Executions: The $execute()$ function in Figure 4 and the $execute'()$ function in Figure 5 explore all possible executions of a test. $execute()$ returns the set of output sequences produced by these executions, and $execute'()$ returns the set of failures observed during the executions. For sequential tests, the execution functions simply run the test in a single thread of execution. We assume sequential executions to be deterministic. Sequential tests that do not behave deterministically, for example, because their behavior depends on the current system time, can be made deterministic by controlling the execution environment [14].

For concurrent tests, the analysis faces the challenge that different executions may expose different behavior due to non-deterministic scheduling. If the oracle does not consider all behaviors from executing a test with $Super$'s constructor, the oracle might report false warnings. For example, the oracle may observe that Sub raises an exception but may miss that the exception is also possible with $Super$ because the oracle does not observe the "right" interleaving of $Super$'s test.

To avoid false warnings due to missed interleavings, the execution functions explore all possible executions of a concurrent test. The analysis uses the stateful software model checker Java PathFinder (JPF) [15]. JPF systematically explores all possible interleavings of the concurrent suffixes of the test, that is, all possible behaviors (output sequences or failures) of a concurrent test despite the non-determinism introduced through scheduling. Checking all interleavings of concurrent calls faces the problem of combinatorial explosion. By focusing on two concurrent suffixes that each have a single call, we ensure that the oracle terminates in reasonable time (Section V-G).

5) Examples: Figures 1 and 2 are two examples of problems detected by the crash oracle. In both examples, executing the test with the superclass constructor succeeds, while executing it with the subclass constructor leads to an exception. A client that calls methods in the same way as in the generated tests will be surprised about a program crash, because the superclass does not show this behavior. Section V-C reports examples of warnings from the output oracle.

IV. Implementation

We implemented the approach presented in Section III into a fully automatic testing tool for Java classes. The implementation takes source code or byte code as input and reports unsafe subclasses as output. We implemented a helper tool to find all superclass-subclass pairs in particular packages and to generate constructor mappings as described in Section III-C. Together with this helper tool, the implementation offers a push button technique to analyze entire class libraries.

To implement the superclass oracle we build upon JPF (version r615) for exploring concurrent executions. JPF is a mature and well-tested research tool but has some limitations,

for example, when analyzing native code. Another problem is that testing even only two concurrent threads that each have a single method call can take a significant amount of time. We deal with these problems by filtering the errors raised by JPF and by canceling JPF after a configurable timeout. If JPF crashes or if it cannot analyze a concurrent test within the timeout, we consider the test to be inconclusive. The superclass oracle does not report any warnings for inconclusive tests. That is, we risk missing some unsafe substitutes but avoid false warnings caused by JPF limitations. For the experiments reported in Section V, we set the timeout to ten seconds per test, which is sufficient for 96% of all concurrent tests. The timeout does not affect the detection of deadlocks because deadlocks are reported by JPF immediately when they occur.

The test generator creates tests with a configurable number of calls. For sequential tests, we set the maximum number of OUT calls after instantiating the OUT to five. For concurrent tests, the prefix contains at most five calls to the OUT.

V. Evaluation

We evaluate our approach with well-tested and widely used Java classes. Our main results are:

- Many classes are crashing substitutes (43/145=30%). All these unsafeties correspond to bugs that should be fixed, because they may cause exceptions and deadlocks in clients.
- Even more classes are output-diverging substitutes (61/145=42%). We classify most of them (57/61) as false positives.
- Developers care about crashing substitutes. At least ten bugs found by the analysis have been fixed by now.

For reproducibility, all experimental data are available for download at http://mp.binaervarianz.de/icse2013.

A. Experimental Setup

We apply the analysis to sequentially and concurrently used classes. As sequentially used classes, we consider classes from three popular Java libraries (Table I). We select libraries, instead of closed programs because every usage simulated by the test generator may happen in some client program. Table I shows the number of analyzed superclass-subclass pairs. We consider all pairs of public and concrete classes that are in a direct or indirect subclass relationship, excluding the superclass `Object` and classes without public constructors—in total, 116 class pairs. As auxiliary classes for generating tests, we pass common classes from the Java standard library and all classes from the respective library. The test generator automatically selects those classes that provide required arguments.

As concurrently used classes, we consider subclasses of 30 classes from the Java standard library that are known to be thread-safe. We consider all 323,107 classes included in the Qualitas Corpus [16] and select a class if it has a thread-safe superclass, if it is public and concrete, and if it has a public constructor. This selection yields 29 subclasses to analyze.

We analyze each class pair until a maximum number of tests has been generated: 500 tests for sequentially used classes and 3,000 tests for concurrently used classes. We set a larger limit for the latter, because a single concurrent test checks only two methods against each other. To reach the maximum number of tests, we run the analysis with different random seeds, where the number of tests generated per seed increases with the number of seeds used (similar to [9]). All experiments are done on an eight-core machine with 3GHz Intel Xeon processors and 8GB memory, running 32-bit Ubuntu Linux and the Java Hotspot VM version 1.6.0_27, giving 2GB memory to the VM.

```
ArrayList l = new ArrayList() OR new Chapter();
Set empty = new HashSet();
boolean b = l.addAll(empty);
```

Result:
`b==false` if `l` is `ArrayList`
`b==true` if `l` is `Chapter`

Figure 6. An output-diverging substitute that we classify as a bug.

We manually inspect all warnings reported by the analysis and classify them as bugs or false positives. A warning is a bug if the subclass behavior should be changed to avoid unexpected behavior in client programs, and it is a false positive otherwise. We distinguish between two kinds of false positives: (1) Differences in behavior that we deem to be acceptable, for example, methods where clients expect that the outcome depends on the runtime type of the receiver. (2) Generic tests where the superclass and subclass constructors are not semantically equivalent, that is, the heuristic constructor mapping (Section III-C) is incorrect.

B. Overview of Unsafe Substitutes Found

Table I shows for both the crash oracle and the output oracle how many warnings the analysis finds and how we classify them. For both oracles, a large percentage of the analyzed subclasses are unsafe substitutes: 30% are crashing substitutes and 42% are output-diverging substitutes. For the crash oracle, all reported warnings correspond to bugs, which is not surprising, because the crash oracle focuses on certainly undesired behavior. In contrast, for the output oracle, most reported warnings are false positives. Two of the four bugs detected by the output oracle are also found by the crash oracle, because the diverging subclass behavior manifests both through different output and through a crash.

Which oracle should developers use? We recommend the crash oracle as a default, because it finds 96% of all detected bugs and because it does not report false positives. In contrast, the output oracle has a high false positive rate and finds few additional bugs.

C. Examples of Unsafe Substitutes

Figures 1 and 2 are bugs found by the crash oracle. Figure 6 shows a bug found exclusively by the output oracle. `Chapter`, a subclass of `ArrayList`, modifies the meaning of the return value of `addAll()` in a subtle way: While the superclass returns whether the content of the list has changed, the subclass returns whether the operation was successful. The difference matters when an empty collection is passed to

Table I
SUMMARY OF RESULTS

Subject	Description	Seq./ Conc.	Class pairs	Crash oracle				Output oracle			
				Warnings	Bugs	False pos.		Warnings	Bugs	False pos.	
						Accept. diff	Constr.			Accep. diff	Constr.
Commons Collections 3.2.1	Container classes	Seq.	12	5	5	0	0	3	2	0	1
dom4j 2.0.0-alpha-2	XML processing	Seq.	46	12	12	0	0	26	0	24	2
iText 5.2.0	PDF editing	Seq.	58	21	21	0	0	30	1	9	20
Qualitas 20101126r	Subclasses of thread-safe classes	Conc.	29	5	5	0	0	2	1	1	0
Sum			145	43	43	0	0	61	4	34	23

```
Namespace ns = new DefaultNamespace("a", "b");
boolean b = ns.supportsParent();
```

Result:
`b==false` if `ns` is `Namespace`
`b==true` if `ns` is `DefaultNamespace`

Figure 7. An output-diverging substitute that we classify as a false positive.

`addAll()`, as illustrated in the figure. We reported this bug and the developers fixed it.[5]

Figure 7 is a false positive reported by the output oracle. The subclass `DefaultNamespace` adds a functionality, support for a parent relationship, to its superclass `Namespace`. Both classes provide a method `supportsParent()` to check in a reflection-like manner whether an object supports this functionality. This behavioral difference is documented and should not surprise clients, because the only purpose of the `supportsParent()` method is to find out whether an object supports the functionality, which is known to depend on the runtime type of the object.

D. Root Causes for Unsafe Substitutes

The large percentage of unsafe substitutes raises the question why subclasses extend their superclasses in an unsafe way. For unsafe substitutes that correspond to bugs, the most common root causes are:

- *Stronger precondition for method arguments*. An overriding method imposes a stronger precondition for method arguments than the overridden method. For example, some overriding methods expect only a subset of all subtypes of the declared parameter type [17], and passing another compatible type raises a `ClassCastException`.
- *Stronger precondition for method receiver*. An overriding method has a stronger precondition on the state of the receiver object than the overridden method. For example, some overriding methods access a field that is not used by the overridden method.
- *Removed methods*. A subclass explicitly "invalidates" a method of the superclass by throwing an `UnsupportedOperationException` and thereby breaks the type system's safety guarantee that each method call is understood.
- *Propagated unsafety*. A class extends a class that itself is an unsafe substitute, and the unsafety propagates down the inheritance hierarchy. Given a class A with an unsafe substitute A', a subclass A'' of A' is also an unsafe substitute for A, unless A'' fixes the problem of A'.
  For example, several unsafe substitutes in iText are the result of extending `Properties`, which is an unsafe substitute for `Hashtable`. The problem is that `Properties` assumes to map `String`s to each other but extends `Hashtable<Object,Object>`, from which it inherits methods to put arbitrary objects into the table. Some methods of `Properties`, such as `propertyNames()`, cast entries to `String`, which fails if the table contains entries of other types. Although this problem of `Properties` is documented, it not only may surprise clients of `Hashtable` that refer to a `Properties` instance, but it also affects subclasses of `Properties`.
- *Missing synchronization*. A subclass of a thread-safe class overrides a method without providing the synchronization guarantees that the overridden method provides. For example, some methods override a synchronized method and remove the `synchronized` keyword without ensuring synchronization in another way.
  One subclass, `NonSynchronizedVector` from Eclipse, obviously removes synchronization on purpose. While this may increase performance, it poses a significant risk at clients of `Vector`, because such clients may unconsciously refer to an instance of `NonSynchronizedVector`.

For unsafe substitutes that we classify as false positives, the most common root causes are:

- *Ad hoc reflection*. A method provides hints on the runtime type of an object and on the functionalities supported by this type (for example, see Figure 7). These methods are an ad hoc form of reflection, and clients calling them should be aware that superclass behavior and subclass behavior may differ.
- *String representations*. A method returns a String representation of the receiver object and the String contains type-specific information. The output oracle filters calls to `toString()` to avoid more false positives of this kind, but it does not handle application-specific methods that return String representations.

[5]See issue 3548434 in the iText bug tracker.

Table II
BUGS REPORTED TO DEVELOPERS.

Issue	Status
Castor 2729	Fixed within a day (reported by others)
Commons Collections 394	Fixed by removing the class
Commons Collections 423	Fixed by removing the class
Commons Collections 424	Acknowledged, working on it
dom4j 3547635	Reported
dom4j 3547784	Reported
iText 3547811	Fixed within a day
iText 3547812	Fixed within a day
iText 3548434	Fixed within a day
JBoss Common 126	Fixed within a week
OpenJPA 2243	Fixed within a day

E. Failures Observed by the Crash Oracle

The crash oracle depends on signs of certainly undesired behavior, such as exceptions and deadlocks. This dependence may raise the question to what extent our results depend on a defensive programming style, where illegal state and illegal arguments are made explicit by throwing exceptions. We categorize the failures that expose crashing substitutes as either explicit or implicit. A failure is *explicit* if it is an exception raised in the analyzed code base. In contrast, a failure is *implicit* if it is an exception raised by the JVM or by the Java standard library, or if it is a deadlock. 84% of the failures that expose crashing substitutes in Table I are implicit, that is, independent of a defensive programming style.

F. Feedback from Developers

Do developers really care about substitutability problems? To answer this question, we report a subset of the bugs found by the analysis to the developers (Table II). By the time of this writing, seven of ten reported bugs have been fixed as a reaction to our reports. Another bug has been reported and fixed independently of us. Moreover, at least two other bugs found by the analysis (not listed in Table II) have been fixed in iText 5.3.0 independently of us, but we could not find a corresponding issue in the project's bug tracker. Many of the remaining unreported bugs are in deprecated classes that are likely to be replaced soon. Overall, the feedback from developers suggests that they care about substitutability problems, which confirms our expectation that unsafe substitutes that can surprise clients are not desired and should be fixed.

G. Performance

The runtime performance of our prototype implementation is acceptable for an automatic analysis. To measure performance, we test each class pair until either the maximum number of tests has been generated or until the subclass has been found to be a crashing substitute. For sequentially used classes, the analysis takes on average 41 seconds to find crashing substitutes and 19 seconds to find output-diverging substitutes. For concurrently used classes, it takes on average 21 minutes to find crashing substitutes and 108 minutes to find output-diverging substitutes. Concurrent tests take longer because the analysis explores all interleavings.

VI. LIMITATIONS

There are limitations to be addressed in future work. First, the effectiveness of the approach is bounded by the ability of the test generator to exercise the analyzed classes. For some classes, the current test generator may not be able to trigger an existing substitutability problem, for example, if the class requires mock objects or external services, such as a database connection. Second, the approach relies on heuristic constructor mappings, which account for a large part of the false positives reported by the output oracle (Table I). One could improve the heuristic by a static analysis of `super()` calls in constructors and of how parameters of subclass constructors propagate to the superclass. Third, our results show the approach to work well for library classes. It remains to be studied how to apply the analysis to classes from closed programs, where some of the usages generated for a class may not match the way the class is used within the program.

VII. RELATED WORK

Our work relates to behavioral subtyping [2], [18] and the substitution property [1]. There is a large body of work on verifying that a class is a behavioral subtype of another class [2], [3], [5], [6]. In contrast to our work, these approaches rely on formal specifications of the behavior of subclasses and superclasses, which is not available for most real-world classes. Offutt et al. propose a model for bugs related to inheritance and polymorphism [19]. They describe nine kinds of anomalies, such as a subclass that modifies state defined by the superclass in a way not expected by the superclass. Our analysis automatically detects these kinds of problems.

America proposes an object-oriented language that distinguishes between subclassing and subtyping [4]. Most popular languages, including Java, blend these two concepts into one, giving rise to the problems revealed by our approach. Taivalsaari gives a good overview of the various notions of inheritance and their respective benefits [20].

There are various techniques for generating sequential tests, such as random test generation [13], [21], [22], techniques based on model checking [23], and techniques based on symbolic execution [24]–[26]. Our test generator differs from these approaches by creating both sequential and concurrent tests and by creating generic tests. We use a feedback-directed, random approach [13], but in contrast to [13], our test generator does not call methods in a random order. Instead, the test generator selects methods that may provide arguments for future method calls or that may modify the state of the object under test. The test generator builds upon an approach for generating concurrent tests [9] and extends it to produce generic tests that can compare the behavior of a subclass and a superclass. This work also differs from [9] by considering a different kind of bug.

Tillmann and Schulte propose parameterized unit tests, where all objects involved in a test are parameters to the test [27]. Their approach generates concrete tests from manually written, parameterized tests through symbolic execution.

Generic tests and parameterized tests share the idea of applying the same test to different objects. In contrast to their work, we generate tests fully automatically and use them to check a subclass against its superclass.

McKeeman proposes to test supposedly equivalent programs, such as multiple compilers for the same language, by comparing them with each other [28]. This idea, called differential testing, has also been used to test system programs [26] and refactoring engines [29]. Our approach analyzes software at a finer level of granularity, namely at the class-level instead of the program-level. As a result, our analysis is more widely applicable: Most Java classes extend at least one other class (in addition to `Object`) [30], whereas few programs have a supposedly equivalent program to compare with. A related idea is to compare an old and a new version of a program and to warn developers about regressions [31], [32]. In contrast to these approaches, our analysis reveals problems within a single version of a program.

We use Java PathFinder [15], which unsoundly considers concurrent executions to be sequentially consistent. This limitation may cause the superclass oracle to report a false warning if a subclass crashes with sequentially consistent execution, while its superclass crashes only with a weak memory model. We did not encounter this case during our evaluation. An extension of JPF to address this problem is described in [33].

VIII. Conclusion

We present a practical approach to detect subclasses that are unsafe substitutes for their superclasses. Unsafe substitutes are a severe problem for large-scale object-oriented development because a programmer may not know the runtime type of an object and because a programmer may not be aware of all subclasses of a class. Our approach automatically reports usages of a superclass reference that lead to surprising behavior, such as crashes, when the reference points to a subclass instance. The approach applies to sequentially and concurrently used classes, addressing the need to ensure correctness in both kinds of programs. Our experimental results demonstrate that unsafe substitutes are a prevalent problem in real-world software and that our analysis effectively addresses it.

Acknowledgments

Thanks for Zoltan Majo and the anonymous reviewers for feedback on this paper. This work was partially supported by the Swiss National Science Foundation under grant number 200021-134453.

References

[1] B. Liskov, "Data abstraction and hierarchy," in *OOPSLA*, 1987, pp. 17–34.

[2] B. Liskov and J. Wing, "A behavioral notion of subtyping," *ACM T Progr Lang Sys*, vol. 16, no. 6, pp. 1811–1841, 1994.

[3] G. T. Leavens and W. E. Weihl, "Reasoning about object-oriented programs that use subtypes," in *OOPSLA/ECOOP*, 1990, pp. 212–223.

[4] P. America, "Designing an object-oriented programming language with behavioural subtyping," in *REX Workshop*, vol. 489, 1990, pp. 60–90.

[5] K. K. Dhara and G. T. Leavens, "Forcing behavioral subtyping through specification inheritance," in *ICSE*, 1996, pp. 258–267.

[6] C. Ruby and G. T. Leavens, "Safely creating correct subclasses without seeing superclass code," in *OOPSLA*, 2000, pp. 208–228.

[7] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2002.

[8] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering (2nd edition)*, 2002.

[9] M. Pradel and T. R. Gross, "Fully automatic and precise detection of thread safety violations," in *PLDI*, 2012, pp. 521–530.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java Language Specification, 3rd Edition*, 2005.

[11] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, 2008, pp. 329–339.

[12] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov, "Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code," in *ICSE*, 2012, pp. 727–737.

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

[14] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs," in *CGO*, 2010, pp. 2–11.

[15] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs," *Autom Software Eng*, vol. 10, no. 2, pp. 203–232, 2003.

[16] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas Corpus: A curated collection of Java code for empirical studies," in *APSEC*, 2010.

[17] M. Pradel, S. Heiniger, and T. R. Gross, "Static detection of brittle parameter typing," in *ISSTA*, 2012, pp. 265–275.

[18] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," in *OOPSLA*, 1986, pp. 38–45.

[19] J. Offutt, R. T. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson, "A fault model for subtype inheritance and polymorphism," in *ISSRE*, 2001, pp. 84–95.

[20] A. Taivalsaari, "On the notion of inheritance," *ACM Comput Surv*, vol. 28, no. 3, pp. 438–479, 1996.

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

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

[23] W. Visser, C. S. Pasareanu, and S. Khurshid, "Test input generation with Java PathFinder," in *ISSTA*, 2004, pp. 97–107.

[24] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, 2005, pp. 213–223.

[25] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *TACAS*, 2005, pp. 365–381.

[26] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.

[27] N. Tillmann and W. Schulte, "Parameterized unit tests," in *ESEC/FSE*, 2005, pp. 253–262.

[28] W. M. McKeeman, "Differential testing for software," *Digit Techn J*, vol. 10, no. 1, pp. 100–107, 1998.

[29] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *ESEC/FSE*, 2007, pp. 185–194.

[30] E. D. Tempero, J. Noble, and H. Melton, "How do Java programs use inheritance? An empirical study of inheritance in Java software," in *ECOOP*, 2008, pp. 667–691.

[31] S. McCamant and M. D. Ernst, "Predicting problems caused by component upgrades," in *ESEC/FSE*, 2003, pp. 287–296.

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

[33] K. Kim, T. Yavuz-Kahveci, and B. A. Sanders, "Precise data race detection in a relaxed memory model using heuristic-based model checking," in *ASE*, 2009, pp. 495–499.