Could I Have a Stack Trace to Examine the Dependency Conflict Issue?

Ying Wang*, Ming Wen†, Rongxin Wu†§, Zhenwei Liu*, Shin Hwei Tan‡,
Zhiliang Zhu*, Hai Yu* and Shing-Chi Cheung†§
* Northeastern University, Shenyang, China
Email: {wangying8052, lzwneu}@163.com, {yuhai, zzl}@mail.neu.edu.cn
† The Hong Kong University of Science and Technology, Hong Kong, China
Email: {mwenaa, wurongxin, scc}@cse.ust.hk
‡ Southern University of Science and Technology, Shenzhen, China
Email: tansh3@sustc.edu.cn

***Abstract*—Intensive use of libraries in Java projects brings potential risk of dependency conflicts, which occur when a project directly or indirectly depends on multiple versions of the same library or class. When this happens, JVM loads one version and shadows the others. Runtime exceptions can occur when methods in the shadowed versions are referenced. Although project management tools such as `Maven` are able to give warnings of potential dependency conflicts when a project is built, developers often ask for crashing stack traces before examining these warnings. It motivates us to develop RIDDLE, an automated approach that generates tests and collects crashing stack traces for projects subject to risk of dependency conflicts. RIDDLE, built on top of ASM and EVOSUITE, combines condition mutation, search strategies and condition restoration. We applied RIDDLE on 19 real-world Java projects with duplicate libraries or classes. We reported 20 identified dependency conflicts including their induced crashing stack traces and the details of generated tests. Among them, 15 conflicts were confirmed by developers as real issues, and 10 were readily fixed. The evaluation results demonstrate the effectiveness and usefulness of RIDDLE.**

***Index Terms*—test generation, third-party library, mutation**

I. INTRODUCTION

Java projects are commonly built on top of third-party libraries, which may in turn have reference to other libraries [1]–[3]. According to Java's class loading mechanism, the specified *classpath* determines the locations where to find the referenced classes during runtime [4], [5]. If multiple versions of the same library or class are present on a *classpath*, only one version will be loaded while the others will be shadowed [6]. A DC (Dependency Conflict) issue occurs if these versions are incompatible. Runtime exceptions (e.g., `ClassNotFoundException` and `NoSuchMethodError`) will be thrown when a project references the methods of incompatible shadowed versions at runtime [7].

Most software management tools such as `Maven` [8] support the detection of potential DC issues and give warnings accordingly. However, they only warn developers of the duplicate classes or libraries rather than all instances of duplication that will induce runtime exceptions. Therefore, developers often want to have a stack trace to validate the risk of these warned DC issues. It echoes earlier findings that stack traces provide useful information for debugging [9], [10]. We observe that developers can more effectively locate the problem of those DC issues accompanied by stack traces or test cases than those without. For instance, a DC issue `Issue #57` [11] was reported to project `FasterXML` with details of the concerning library dependencies and versions. Nevertheless, the project developer asked for a stack trace to confirm if the reported issue is a problem.

"I would need full stack trace to make sure exactly where the problem occurs. If that still happens, I need stack trace to investigate further. At this point, this issue will be closed, assuming the problem has been resolved. But please reopen it if the issue is reproducible, and stack trace is available as additional information."

```
1. @Api
2. @SwaggerDefinition(tags = @Tag(name = "test",extensions=
3. @Extension(name="test",properties = @ExtensionProperty(name = "test", value = ""))))
4. public class Test {
5.     public static void main(String[] args) {
6.           BeanConfig beanConfig = FF4jSwaggerConfiguration.getBeanConfig();
7.           beanConfig.setResourcePackage("org.ff4j.web.api");
8.           beanConfig.setScan(true); //Entry method
9.     }
10.}
```
(a) A manual test

```
141. if (reflections.getTypesAnnotatedWith(SwaggerDefinition.class) != null) {
142.    Swagger swagger = readSwaggerConfig(classes);
143. ...
144. }
```
(b) Code snippet of method read(Reader.java)

```
173. AbstractReader configReader;
174. ...
175. configReader.read();
```
(c) Code snippet of method setScan(BeanConfig.java)

```
Exception in thread "main" java.lang.NoSuchMethodError:
org.apache.commons.lang3.StringUtils.prependIfMissing(Ljava/lang/String;Ljava/lang/
CharSequence;[Ljava/lang/CharSequence;)Ljava/lang/String;
1. at io.swagger.util.BaseReaderUtils.parseExtensions(BaseReaderUtils.java:30)
2. ...
3. at io.swagger.jaxrs.Reader.readSwaggerConfig(Reader.java:462)
4. at io.swagger.jaxrs.Reader.read(Reader.java:142)
5. at io.swagger.jaxrs.config.BeanConfig.setScan(BeanConfig.java:172)
```
(d) Stack trace

Fig. 1: An example of a test for `Issue #309`

Similar observations can be made in many other DC issues (e.g., `Issue #8706` [12], `Issue #501` [13], etc.). Developers showed more interests in a failing test case or stack trace to know what the conditions of triggering the shadowed methods are. Consider the following example. In `Issue #309` [14], `Ff4j` directly depends on libraries `commons-lang3 3.0` and `swagger-core 1.5.4`, and transitively references

§Rongxin Wu and Shing-Chi Cheung are the corresponding authors.

the features included in `commons-lang3 3.2.1` via library `swagger-core 1.5.4`. According to `Maven`'s loading mechanism for duplicate JARs, library `commons-lang3 3.2.1` will be shadowed. Figure 2 shows the two paths identified by static analysis from the entry method `setScan()` defined in `Ff4j` to the methods that only defined in the shadowed library. On the two paths, classes `DefaultReader` and `Reader` implement the method `read` defined in the abstract class `AbstractReader`, as shown in Figure 1 (c).

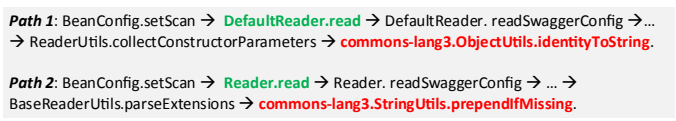

Fig. 2: Two paths identified by static analysis from the entry method to shadowed methods

However, due to dynamic binding, compiler could only resolve at run time whether to use method `DefaultReader.read` or `Reader.read`. This illustrates the shortcoming of static analysis. In this case, with the aid of the test shown in Figure 1 (a), we get a crash with the stack trace shown in Figure 1 (d). Stack trace consists of a sequence of methods that were active on the call stack at the time that an exception occurred due to the DC issue. Each stack frame contains the full-qualified name of method and the exact location of the execution inside the source code together with a file name and line number [15]. It can capture precise runtime information such as dynamic bindings for developers to facilitate the debugging tasks. It also demonstrates the feasibility of reproducing the DC issue. This motivates us to study how to generate tests (including test inputs and test scripts) that lead to failures for DC issues.

For ease of presentation, we refer to the project dependent on libraries as a *host project*; the method in a shadowed class version as a *shadowed method*; and a branch containing a call site of a shadowed method as a *target branch* hereafter. A major challenge of test generation for DC issues lies in the difficulties in reaching the target branches along the path from entry methods defined in a host project to the shadowed methods provided by third party libraries. Especially, the conditions on the path may involve complex object creation, private method invocation or field accesses, etc., making the conditions hard to construct [16]. For instance, we can tell that to reach the target branch, the test (`Lines 1-3` ,and 7) shown in Figure 1 (a), covers a complex condition in `Line 141` of the code snippet shown in Figure 1 (b). However, this branch condition involves the environment dependency and complex objects, which are difficult for test generation tools to construct automatically.

Wang et al. conducted an empirical study on open source Java projects to characterize DC issues. Leveraging the empirical study results, they developed DECCA, a static analysis tool, to detect DC issues with a high precision [7]. However, there is no prior work on the test generation for DC issues. In this paper, we present RIDDLE (tRIgger DepenDency confLict crashEs), the first automated test generation technique for DC issues. RIDDLE works in two phases: condition mutation and condition restoration. In the first phase of condition mutation, it identifies program paths from an entry methods to shadowed methods. It mutates the conditions of each branch in an identified path by making them evaluated to be `TRUE` or `FALSE` so as to force a test execution along the path that invokes a shadowed method. We call the mutation operation forcing a condition evaluating to trigger a desirable branch as *short-circuiting*, and the program after mutation as a *program variant*. Consider the above example shown in Figure 1, after short-circuiting the unsolvable branch condition (`Line 141`) in method `read`, a crash can be triggered by an automatic generated test, as shown in Figure 3.

```
1. @Test(timeout = 4000)
2. public void test0()  throws Throwable  {
3.     BeanConfig beanConfig0 = FF4jSwaggerConfiguration.getBeanConfig();
4.     beanConfig0.setScan(true); //generate stack trace with NoSuchMethodError
5. }
```

Fig. 3: An automatic generated test for `Issue #309` after short-circuiting an unsolvable branch condition

Short-circuiting offers two advantages. Firstly, it significantly increases the chances of generating a stack trace (i.e., runtime information) leading to a DC issue. It helps to collect runtime information required to reach the target branches that often involve environment dependencies and non-linear constraints, which are known to be difficult to solve [16]. Secondly, when a target branch cannot be reached using short-circuiting, it identifies the key constraints to be solved in order to reach a potential DC issue. For example, when a short-circuited condition involves dereferencing a null-initialized object variable, those constraints that compute a proper value for this variable need to be solved. Otherwise, the program execution catches a `NullPointerException` at the short-circuited branch and terminates before reaching the target branch. We will discuss it in Section IV-B.

The first phase generates a set of program variants with short-circuited branch conditions along the path that invokes an identified shadowed method. In the second phase of condition restoration, RIDDLE first uses a guided search-based strategy to generate tests for each short-circuited program variant. Then, it iteratively restores solvable conditions to (1) capture more precise runtime information with more solved constraints and (2) reduce infeasible behaviors caused by short-circuiting. Besides the stack trace obtained after condition restoration, RIDDLE provides the details of remaining branch conditions that cannot be solved after restoration to developers for further checking. From the feedback of the reported issues (see Section IV-C), we observed that developers can easily work out the above unrestored branch conditions based on their domain knowledge of their own projects.

We applied RIDDLE to analyze the latest version of 19 real-world Java projects with duplicate libraries or classes, which include 1.3 million of lines of code, to generate tests to produce their failure-introducing conditions and inputs. Then, we reported 20 identified DC issues with full stack traces and test cases. So far, we have received acknowledgment

from developers on 15 reported issues. Ten of them have been quickly fixed. Most of the confirmed issues are identified in popular projects such as `Apache azure storage` [17], `Google truth` [18], `Webmagic` [19], etc. These results demonstrate that RIDDLE can provide useful test information to facilitate issue diagnosis and fixing. To summarize, we make the following contributions in this paper:

- To the best of our knowledge, we proposed the first automated test generation technique to produce the failure-introducing conditions and inputs of DC issues.
- We provide a publicly available implementation of the technique, RIDDLE[a]. It can be considered as an extension to the search-based test generation tool, EVOSUITE [20].
- The experimental evaluation on real-world projects shows the effectiveness and usefulness of the proposed technique.

II. PRELIMINARIES

A. Problem Formulation

DC issues arise when the loaded classes do not cover the methods referenced by a project [7]. Generally, DC issues are manifested at two granularities:

- **Class level:** Conflicts in classes among libraries, or between host project and libraries. If duplicate classes exist in one project, only one version will be loaded based on the class loading mechanism. In this case, host project will throw `NoSuchMethodError` etc. when referencing the methods that are not defined in loaded classes.
- **Library level:** Conflicts in library versions. If a project directly or transitively depends on multiple versions of the same library, one will shadow the others during the packaging process of build tool. In such scenario, the host project will crash with `ClassNotFoundException` etc. when referencing the methods belonging to the classes only included in the shadowed library.

In this work, we define multiple versions of class C_i or library Lib_i as potential DC issue I_i. Consider a host project $\mathcal{H}$ with a set of potential DC issues $\mathcal{I} = \{I_1, I_2, \cdots, I_n\}$. For each $I_i \in \mathcal{I}$, $\mathcal{S}_i$ is the method set defined in the shadowed version of class C_i or library Lib_i and $\mathcal{RH}_i$ is the method set belonging to duplicate class C_i or library Lib_i and referenced by project $\mathcal{H}$. To formulate our research problem, we introduce the following concepts.

Definition 1. (Risky method set): If $\mathcal{R}_i = \mathcal{RH}_i \bigcap \mathcal{S}_i \neq \emptyset$, we define $\mathcal{R}_i = \{rf_1, rf_2, \cdots, rf_{|\mathcal{R}_i|}\}$ as the risky method set for issue I_i, since the runtime exceptions will be thrown when project $\mathcal{H}$ has references to any risky method $rf_m \in \mathcal{R}_i$.

Definition 2. (Boundary class): Suppose class BC_k belongs to host project $\mathcal{H}$ and directly depends on the methods of third library Lib_i. We then define class BC_k as a boundary class between host project and third party libraries. For each risky method $rf_m \in \mathcal{R}_i$, there is a set of boundary classes in project $\mathcal{H}$ that directly or transitively reference it. We denote such a boundary class set as $\mathcal{CL}_i = \{BC_1, BC_2, \cdots, BC_{|\mathcal{CL}_i|}\}$.

Problem definition: Given a project with DC issue I_i, our research problem is to generate a test t_p (test inputs and test script) to trigger the execution of any risky method $rf_m \in \mathcal{R}_i$, thereby causing runtime exceptions (i.e., `NoSuchMethodError`, `ClassNotFoundException`, etc.).

B. Motivation

Issue reports are vital for any software development. They allow users to inform developers of the problems encountered while using a piece of software. Recent research [21], [22], [23]–[25] shows that providing stack traces or test cases can greatly elevate the quality of issue reports. Stack traces are useful runtime information, which can support developers in debugging tasks [15], [26].

To investigate the importance of a stack trace in validating DC issue reports, we conducted an empirical study to check whether the confirmation rate is different between DC issues with stack traces and those without. We collected the subjects satisfying the following two criteria. First, the Java project have received more than 50 stars or forks (popularity). Second, the bug tracking system of each selected Java project contains two types of DC issues: with and without stack traces. In this manner, we can minimize the impacts of maintenance activity of different projects on the statistics. Note that we used the keywords "library", "dependency", "conflict", etc. to search the DC issues and refined the results by manual checking to remove the noisy ones. For the DC issues without stack traces, we manually checked the results and only collected the ones with clear description of problem manifestation. Eventually, we obtained 22 Java projects with 59 DC issues[b].

Furthermore, we manually investigated the collected 59 issue reports and related discussions to calculate the time (days) for confirmation, which is the duration from the opening time of an issue report to the time when this issue is confirmed. For the unconfirmed issues, we consider their unconfirmed duration (days) from the issue open time to data collection time.

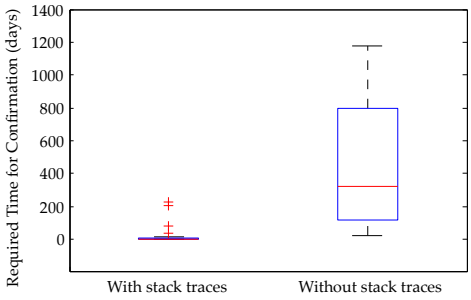

Fig. 4: Required time for confirmation of the two types of DC issues

Figure 4 reports the time for confirmation of two types of DC issues. We observed that, on average, the reported DC issues with stack traces can be confirmed by developers 521 days faster than those without. The result indicates that the stack traces indeed help to speed up validation process for DC issues.

[a]https://skillwind.github.io/RiddleDC/index.html

[b]https://github.com/skillwind/Empirical-Study-Subjects

C. Challenge

Figure 5 shows an example to explain the research challenges of the problem. The host project directly depends on `a-1.0.Jar` and `b-2.0.Jar`, and library `b-1.0.Jar` is transitively introduced by `a-1.0.Jar`. As we can see, the project depends on two versions of library `b`. Suppose the project is built by `Maven`, then `b-1.0.Jar` will be shadowed based on `Maven`'s *nearest wins strategy* and `b-2.0.Jar` is loaded as it appears at the nearest to the root (host project) of the dependency tree. By static analysis, host project references method `D()` which is only included in the shadowed JAR.

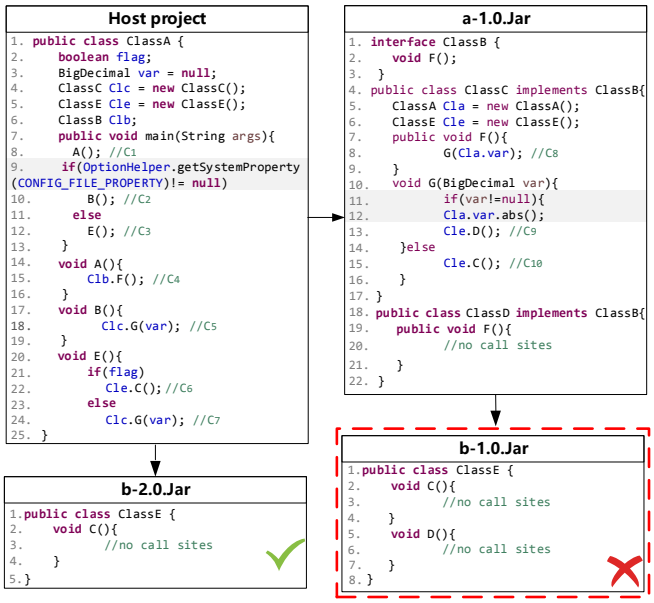

Fig. 5: Source code of motivation example

Using the example, we explain the major research challenge of test generation for a given Java project with DC issues.

Challenge *It is difficult to resolve complex branch conditions to trigger the shadowed methods.* Existing search-based tools like EvoSuite are less ineffective in handling complex branch conditions. They provide little guidance to search the boolean flags and create complex object instances as required by these conditions [16]. Consider `ClassA` defined in the host project as the boundary class under test. To trigger the runtime exception caused by this DC issue, the execution of `main()` under tests (inputs) should cover risky method `D()`. However, the invocation path from `main()` to `D()` passes through a third party library `a-1.0.Jar`, which introduces more complex conditions preventing generated tests from reaching the target branches. For instance, to ensure that a test executes along the invocation path `main()` $\rightarrow$ `B()` $\rightarrow$ `G()` $\rightarrow$ `D()`, it should cover the conditional branches in `Line 9` of `main()` and `Line 11` of `G()`. However, the condition statement in `main()` is implemented to obtain the local path of a log configuration file from system property, which is unlikely to be constructed by test generation tools.

RIDDLE uses a genetic algorithm to evolve sets of candidate tests, which aims to maximize the possibility of triggering the risky method execution with the defined guidance criteria. To address the above challenge, RIDDLE short-circuits all the branch conditions in the path `main()` $\rightarrow$ `B()` $\rightarrow$ `G()` $\rightarrow$ `D()`, to force the execution of generated tests to cover these desirable branches. In this manner, the complex condition in `Line 9` of `main()` can be overcome.

However, infeasible behaviors can be introduced by short-circuiting. Consider the conditional branch in `Line 10` of `G()`. In the case that instance `var` is a null-initialized object variable, the tests will trigger a `NullPointerException` at the statement `Cla.var.abs()`, if we force the above condition to be `TRUE`. RIDDLE solves this key constraint by restoring the branch condition of in `Line 10` of `G()`. After restoring this condition, RIDDLE overcomes this branch by passing a non-null object of type `BigDecimal` to method `G()`. Based on the above steps, risky method `D()` finally can be covered by generated tests. Besides, for the unrestored branch condition (`Line 9` of `main()`), RIDDLE provides its details to developers for further checking.

III. Test Generation

A. Overview

In this work, we propose a test generation approach, RIDDLE. It takes the project with DC issues as input, and output the generated tests that can trigger the risky methods. Figure 6 shows an overview of our approach. It mainly contains the following four steps.

- **Step 1: Static analysis.** RIDDLE identifies risky method set for each DC issue and then extracts the control flow dependencies to construct IMCFG (Inter-method Control Flow Graph).
- **Step 2: Condition mutation.** RIDDLE short-circuits the branches that can reach to an identified risky method, by forcing their condition statements to be evaluated to be `TRUE` or `FALSE`, RIDDLE forces the execution of tests to cover these desirable branches.
- **Step 3: Search-based test strategy.** RIDDLE uses a genetic algorithm to evolve sets of candidate tests, which aims to maximize the possibility of triggering the risky method execution.
- **Step 4: Condition restoration.** To reduce the infeasible behaviors caused by condition mutations, RIDDLE restore the mutated branches in IMCFG depth by depth, and check whether the generated tests can cover risky methods in these cases. If so, it retains the restored branch conditions. Otherwise, it labels the conditions as unrestored conditions. Finally, RIDDLE provides program variants with unrestored branch conditions to developers for further checking their feasibility.

B. Static Analysis

In the static analysis phase, RIDDLE first identifies the risky method set for each DC issue and then constructs IMCFG by control flow analysis and call graph analysis.

Identification for risky method set. We obtain the multiple versions of library Lib_i or class C_i, as well as referenced

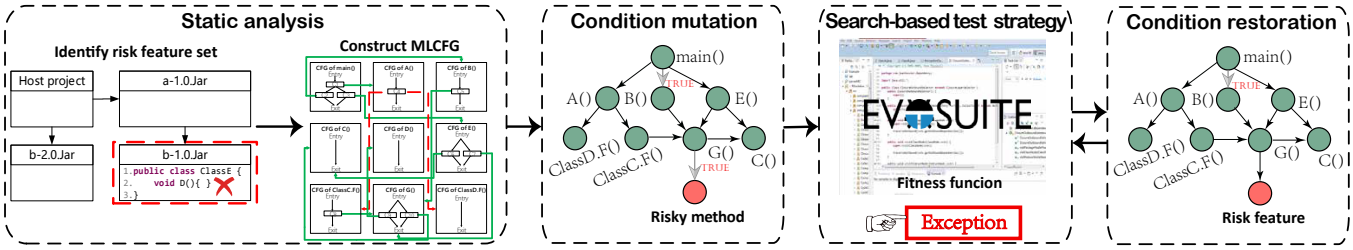

Fig. 6: Logical architecture of RIDDLE

method set $\mathcal{RH}_i$, by analyzing dependency tree and bytecode (JAR or class files). Note that, the dependency tree can be readily extracted from projects' dependency management script (e.g., `pom.xml`, `build.gradle` etc.). Furthermore, the method set S_i defined in the shadowed version of Lib_i or C_i is deduced based on the class loading rules of build tools. In this work, RIDDLE identifies the shadowed method set according to Maven's class loading mechanism, which has been systematically summarized in Wang et al.'s empirical study [7]. Finally, for any DC issue I_i, we extract its corresponding risky method set $\mathcal{R}_i = \mathcal{RH}_i \bigcap \mathcal{S}_i$.

IMCFG construction. Control flow dependencies among method calls help to guide testing to cover the identified risky methods. Inspired by Wu et al.'s approach [9], we construct a reduced version of control flow graph that only preserves the intraprocedural control flow dependencies for call sites within a method. Besides, we apply call graph analysis to build the interprocedural edges between methods. Based on the above steps, we can statically build the IMCFG. Formally, IMCFG is defined as follows.

Definition 3. (Inter-method control flow graph): $G = \{V_\mathrm{f}, V_\mathrm{c}, E_\mathrm{f}, E_\mathrm{c}\}$, where V_f is a set of nodes, each of which represents a method that is passed through by the program paths from any entry method ef_n defined in boundary class BC_k to any risky method $rf_m \in \mathcal{R}_i$; V_c represents a set of call sites pointing to the methods in V_f; $E_\mathrm{f} = \{< f_i, f_j > | f_i, f_j \in V_\mathrm{f}\}$, which denotes a set of invocation relationships between methods, including the dynamic bindings; and $E_\mathrm{c} = \{< c_i, c_j > | c_i, c_j \in V_\mathrm{c}\}$, which is a set of intraprocedural control flow dependencies between call sites.

Definition 4. (Conditional branch): If $\exists < f_i, f_j > \in E_\mathrm{f}$ and there are at least one control flow path from entry to exit that do not pass through the call site pointing to method f_j in the intraprocedural control flow graph of f_i, then we define $< f_i, f_j >$ as a conditional branch. Also, we denote its conditional statement as T_{ij}, which is equivalent to a boolean variable `TRUE` or `FALSE`.

Definition 5. (Polymorphic branch): If $\exists < f_i, f_j > \in E_\mathrm{f}$ and the invocation relationship between f_i and f_j is caused by dynamic binding, then $< f_i, f_j >$ is considered as a polymorphic branch.

Definition 6. (Branch node): If $\exists < f_k, f_t >$ is a conditional or polymorphic branch in IMCFG, we consider f_k is a *branch node*.

We implemented the above static analysis based on the `Soot` [27] framework. RIDDLE leverages `Soot`'s program dependency graph, control flow graph and call graph APIs to construct IMCFG and identify the referenced risky method set based on DECCA. Figure 7 exemplifies the IMCFG construction for the sample code shown in Figure 5. To explain the correspondences between the call sites and methods in Figure 7, we labeled the call site symbols as code comments in Figure 5. With the aid of interprocedural edges, we obtain the three invocation paths from entry method `main()` to risky method `D()`: `main()` → `A()` → `ClassC.F()` → `G()` → `D()`; `main()` → `B()` → `G()` → `D()`; and `main()` → `E()` → `G()` → `D()`. By analyzing the CFG of each method, we found that `main()` → `B()`, `main()` → `E()`, `E()` → `G()`, `E()` → `C()`, `G()` → `D()` and `G()` → `C()` are conditional branches, since their execution is determined by control conditions. Moreover, `A()` → `ClassC.F()` and `A()` → `ClassD.F()` are polymorphic branches, whose execution is determined by dynamic bindings.

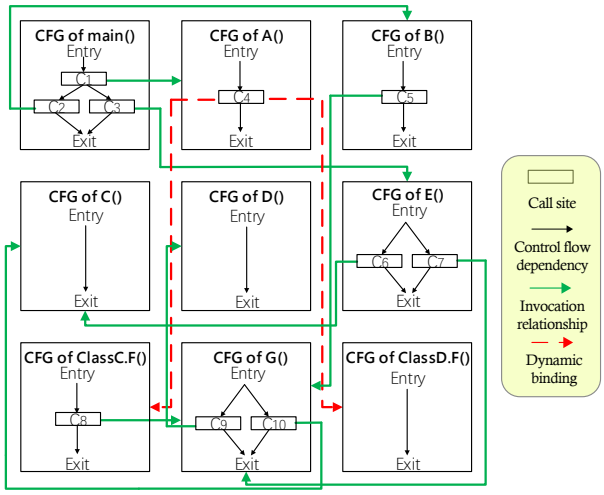

Fig. 7: IMCFG of the example code

C. Search-based Test Strategy

As symbolic execution techniques suffer from path explosion problem and require constraint solving [28], [29], we selected the search-based unit test generation tool, EVOSUITE [20], to build RIDDLE. Our approach improves its coverage criterion to trigger the failures caused by DC issues. For a given host project $\mathcal{H}$ with DC issue I_i, EVOSUITE considers one boundary class $BC_k \in \mathcal{CL}_i$ at a time and uses a genetic algorithm to generate test suites for it.

We briefly describe how EVOSUITE works. EVOSUITE leverages genetic algorithm for generating a JUnit test suite for a given Java class. The steps in EVOSUITE involve:

Initialization. An initial population of individuals is randomly generated [30]. In EVOSUITE, an individual is a whole test suite consists of tests that execute methods in the program and test oracles.

Fitness. The fitness is computed according to the given fitness function, a quantitative measurement of an individual's overall effectiveness [31]. Currently, EVOSUITE supports different coverage criteria (e.g., statement coverage, branch coverage, or mutation coverage), where the default criterion is branch coverage over the Java bytecode.

Crossover. EVOSUITE performs crossover between two test suites by exchanging test cases based on a randomly chosen crossover position.

Mutation. Mutation of test suites in EVOSUITE involves adding new test cases, or mutating individual test. Meanwhile, tests are mutated by inserting, removing, or modifying individual statements and parameters. To enhance the generation of input data, EVOSUITE also leverages focused local searches and dynamic symbolic execution after every predefined number of generations.

Iteration and Termination. The new population is evaluated and evolved until a stopping criterion is reached [32].

Our modifications of EVOSUITE aim to guide the search towards the identified risky method. Specifically, our goal is to maximize the possibility of covering the identified risky method. In general, the more branches on the control flow paths from the entry method to the risky method, the more difficult to generate appropriate input data to trigger the crashes. Hence, to ease the generation of input data, our modified fitness function estimates how close a test suite is to covering the identified risky method. For each method $f_k \in V_f$ in IMCFG, we calculate the number of branches passed through by the paths from f_k to risky method $rf_m \in \mathcal{R}_i$, which is denoted by $BN(f_i)$. In this work, the fitness of individual ts_i is the minimum of $\mathcal{BN}_{ts_i} = \{BN(f_i)|f_i \in CV_{ts_i}\}$, where CV_{ts_i} represents the method set covered by test suite ts_i. The individuals with lower fitness value tend to be selected during the evolution process. Consequently, an individual has fitness 0 if it covers the identified risky method.

RIDDLE calculates the fitness of test suite ts_i in the following four steps:

First, it identifies method set CV_{ts_i} covered by ts_i.

Second, it initializes $\mathcal{BN}_{ts_k} = \{BN(f_k)|f_k \in CV_{ts_i}\}$ to record the number of branches passed through by invocation paths from each method in CV_{ts_i} to risky method $rf_n \in \mathcal{R}_i$.

Third, it let $BN(rf_n) = 0$, where $rf_n \in \mathcal{R}_i$. Then, it removes the methods directly or indirectly invoked by risky method rf_n from IMCFG.

Fourth, RIDDLE traverses the methods in CV_{ts_i} in a bottom-up manner based on invocation relationships. For each $f_k \in CV_{ts_i}$, it performs the following tasks:

(1) If f_k is a leaf node in the call graph, it let $BN(f_k) = \infty$ since the leaf node could not reach the target branch.

(2) If f_k is a non-leaf node in the call graph, it checks whether f_k is a branch node. There are two cases in this scenario: (a) If f_k is a branch node, we assign $BN(f_k)$ to be the sum of $MinChd(f_k)$ and $NBranch(f_k) - 1$, where $NBranch(f_k)$ denotes the number of conditional or polymorphic branches starting from method f_k in IMCFG, $MinChd(f_k)$ represents the minimum of $\mathcal{BN}_{chld} = \{BN(f_t)|f_t \in DV_{f_k}\}$, and DV_{f_k} is the set of methods directly invoked by f_k in the call graph. (b) If f_k is neither a leaf node in the call graph nor a branch node, we assign $MinChd(f_k)$ to $BN(f_k)$.

Finally, the fitness of test suite ts_i is equal to the minimum of $\mathcal{BN}_{ts_i} = \{BN(f_t)|f_t \in CV_{ts_i}\}$.

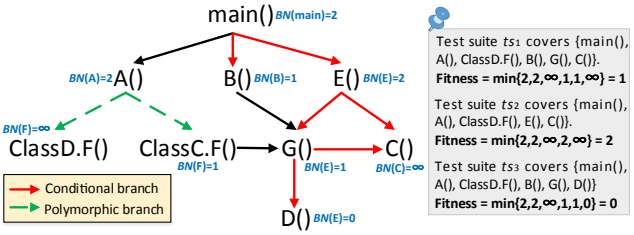

Fig. 8: An illustrative example to explain the fitness of an individual

Figure 8 illustrates how to calculate the fitness of an individual using the sample code shown in Figure 5. Based on invocation relationships, we traverse the methods from the bottom-up and deduce the number of branches from each method to risky method `D()`. As the leaf nodes `ClassD.F()` and `C()` are not reachable to `D()`, we have $BN(ClassD.F) = BN(C) = \infty$. There are two conditional branches starting from `G()`, so $NBranch(G)$ is equal to 2. Then, $BN(G) = min(BN(D), BN(C)) + (2 - 1) = 1$. Similarly, we can obtain $BN(f_k)$ for each method f_k as shown in Figure 8. Consider a test suite ts_1 that covers $CV_{ts_1} = \{$`main()`, `A()`, `ClassD.F()`, `B()`, `G()` and `C()`$\}$. Its fitness is equal to min{2, 2, ∞, 1, 1, ∞} = 1. By contrast, test suite ts_2 covers method `E()` which needs to span more conditional branches to trigger `D()`. In terms of possibility of triggering the risky method, test suite ts_1 is preferable to ts_2, resulting in a higher fitness value. Moreover, test ts_3 has fitness 0, as it successfully covers risky method `D()`.

D. Condition Mutation and Restoration

Condition mutation. $L_t =< ef_m, \cdots, rf_n >$ is an invocation path from an entry method ef_m defined in boundary class $BC_k \in \mathcal{CL}_i$ to risky method $rf_n \in \mathcal{R}_i$. Suppose that L_t passes through a set of conditional branches whose corresponding condition statement set is $\mathcal{CD}_t = \{T_{pq}| < f_p, f_q > \in E_f\}$. To guide test t_k to execute along invocation path L_t, RIDDLE mutates the program P by forcing all the branch conditions in $\mathcal{CD}_t$ to be evaluated to be `TRUE` or `FALSE`, thereby we get a program variant $P'(L_t)$. For each risky method $rf_n \in \mathcal{R}_i$, there is an invocation path set $\mathcal{L}_n = \{L_1, L_2, \cdots, L_{|\mathcal{L}_n|}\}$ that is possible to trigger its execution. RIDDLE generates a program variant set $\mathcal{P} = \{P'(L_k)|L_k \in \mathcal{L}_n\}$ and then adopts a genetic algorithm to generate tests for each $P'(L_k)$.

Short-circuiting the branch conditions provides two benefits: (1) It maximizes the probability of generating a stack trace (i.e., runtime information) leading to a DC issue. It significantly increases the chances of collecting runtime information required to reach the target branches that often involve environment dependencies and non-linear constraints, which are known to be difficult to solve [16]. (2) It identifies the key constraints to be solved in order to reach a potential DC issue, when a target branch cannot be reached based on short-circuiting. As mentioned earlier in Section II-C, short-circuiting may introduce some infeasible behaviors. Therefore, RIDDLE tries to restore the mutated branch conditions to compensate these side effects.

Condition restoration. If the generated test trigger the risky method rf_n under program variant $P'(L_t)$, RIDDLE generates a program variant $P''(L_t)$ by restoring one mutated condition statement that is nearest to entry method ef_m in IMCFG. Furthermore, we check whether the search strategy can generate tests to cover the risky method rf_n in this case. If the tests work, we retain the restored branch condition, otherwise, this branch condition still should be short-circuited. By analogy, RIDDLE restores the branch conditions from the top bottom in IMCFG, and generate tests for their corresponding program variants. Then, we prioritize the test cases that cover the risky method in ascending order based on the number of unrestored branch conditions in program variants. Intuitively, the test that covers risky method with the fewest unrestored branch conditions, is considered as the optimal test.

In summary, RIDDLE handles the condition mutation and restoration process in the following steps:

First, it initializes the test suite $\mathcal{T}_n$ and program variant set $\mathcal{P}$ to empty.

Second, it identifies invocation path set $\mathcal{L}_n$ from ef_m to rf_n. For each path $L_t \in \mathcal{L}_n$, we perform the following tasks:

(1) It identifies the conditional branch set $\mathcal{CD}_t$ passed through by path L_t.

(2) It generates program variant $P'(L_t)$ by short-circuiting the branch conditions in $\mathcal{CD}_t$.

(3) With the aid of search strategy, it generates test suite $\mathcal{T}_t$ for program variant $P'(L_t)$. If $\exists t_k \in \mathcal{T}_t$ can trigger risky method rf_n, then it let $\mathcal{T}_n \leftarrow \mathcal{T}_n \bigcup t_k$.

(4) Then it ranks the conditional branches in $\mathcal{CD}_t$ according to their invocation distances to entry method ef_m in ascending order and obtains sequence Q_B.

(5) For each $T_{ij} \in Q_B$, RIDDLE restores its branch condition and generate program variant $P'(\mathcal{CD}_t \setminus T_{ij})$. Then, it generates test suite $\mathcal{T}_t'$ for $P'(\mathcal{CD}_t \setminus T_{ij})$. If $\exists t_k \in \mathcal{T}_t'$ can trigger risky method rf_n, RIDDLE retains the restored branch condition T_{ij} and let $\mathcal{T}_n \leftarrow \mathcal{T}_n \bigcup t_k$, $\mathcal{CD}_t \leftarrow \mathcal{CD}_t \setminus T_{ij}$ and $\mathcal{P} \leftarrow \mathcal{P} \bigcup P'(\mathcal{CD}_t)$. Otherwise, T_{ij} still should be short-circuited.

After trying to restore the branch conditions in $\mathcal{CD}_t$ depth by depth, RIDDLE finally obtains the test suite $\mathcal{T}_n$ and its corresponding program variant set $\mathcal{P}$.

Since EVOSUITE performs instrumentation on the bytecode of the System Under Test (SUT), RIDDLE mutates branch condition statements with the aid of ASM [33], a Java bytecode manipulation and analysis framework. It can be used to modify existing classes directly in binary form. Restoring some solvable conditions brings two benefits: (1) capture more precise runtime information with more solved constraints and (2) reduce infeasible behaviors caused by short-circuiting. For the unrestored branch conditions, RIDDLE can provide their detailed information to developers for further checking.

TABLE I: Basic information of experimental subjects

ID	Project	Category	Version	KLOC	# Stars
1	Azure storage [17]	SDK	508ec1f	79.6	132
2	Google/truth [18]	Testing	83a2d5d	35.3	1482
3	Ff4j [34]	Framework	83a2d5d	35.3	1482
4	Incubator dubbo [35]	Framework	1c16f78	111.1	20831
5	Jetbrick/template [36]	Framework	ae3896f	10.4	224
6	Webcam/capture [37]	Driver	a837030	16.6	1303
7	Blueflood [38]	Database	4722a34	42.8	564
8	Vertx/swagger [39]	Web server	11437ef	9.1	62
9	Webmagic [37]	Web application	be892b8	12.1	6883
10	Apache Storm [40]	Distributed system	c2931da	267.4	5226
11	Htm.java [41]	Platform	8fc6b59	54.7	255
12	Incubator service [42]	SDK	e1671b4	100.7	549
13	Selendroid [43]	Framework	44118d6	35.5	703
14	HotelsDotCom/styx [44]	Platform	6911de9	54.2	131
15	Wisdom [45]	Framework	a35b643	84.3	76
16	Geowave [46]	Plugin	4de7e02	194.6	272
17	Mayocat shop [47]	Platform	ec84978	27.8	161
18	Vipshop/saturn [48]	Platform	c5ae22e	55.5	951
19	St-js [49]	Plugin	a97a611	34.2	149

IV. EVALUATION

We evaluate the effectiveness and usefulness of RIDDLE using real-world open source projects with the aim to answer the following two research questions:

- **RQ1 (Effectiveness):** What is the overall effectiveness (in terms of condition mutation and restoration)?
- **RQ2 (Usefulness):** Can RIDDLE provide useful information for developers to facilitate diagnosing and fixing DC issues?

A. Experimental Design

To answer the two research questions, we conducted experiments on 19 open source Java projects. These projects were randomly selected from `Github`, satisfying two criteria. First, it has received over 50 stars or forks (popularity). Second, it contains DC issues with risky methods detected by DECCA. We further ensure that the subjects selected for evaluation are different from those used in our empirical study in section II-B, to avoid bias towards developers who prefer bug reports with stack traces. Table I gives the basic information of these projects, which includes: (1) project name, (2) category, (3) the version used in our experiments, (4) lines of code, and (5) number of stars. As we can see from the table, these projects are diverse (covering 10 different categories), non-trivial (with code size ranging from 9.1K $\sim$ 267.4K lines of code), and popular (with the number of stars ranging from 62 to 20,831). The experiments were conducted on a dual-core machine with Intel Core i7 CPU @2.8 GHz and 8GB RAM.

To study RQ1, after identifying the risky method using DECCA, we applied the following three techniques to generate tests for DC issues and then compared their effectiveness on

covering the risky methods. The detailed experimental settings are described below:

- **EVOSUITE:** It uses EVOSUITE's default fitness function to guide the evolutionary search. Although EVOSUITE has many coverage criteria that can be tuned for different tasks. In this paper, we use its default branch coverage criteria, as it has been shown effective in many prior studies [20], [50]–[52].
- **EVOSUITE & Fitness:** It uses the fitness function defined in Section III-C to guide the evolutionary search.
- **RIDDLE:** It leverages search-based strategy with our defined fitness function, condition mutation and restoration strategies to generate tests.

All of the above three techniques target at generating test for the boundary class under test with the EVOSUITE's default options, except for two settings: 1) following earlier works [16], we set the time budget for the search to three minutes per boundary class, and 2) we configured EVOSUITE's parameter `INSTRUMENT_CONTEXT` to be `TRUE`, to allow the three techniques to dive into the methods that are not defined in the boundary class under test based on the invocation context. In addition, due to the usage of randomized algorithms in EVOSUITE, for the above three techniques, we repeated the process of test generation for 10 times for each DC issue with different random seeds. The final results are averaged over the 10 runs in order to avoid the biased results.

To study RQ2, we configured RIDDLE to report: (1) conflicting root causes arising from multiple versions of the same libraries or classes in the project, (2) risky method set, and (3) optimal test, which is the generated test with the fewest unrestored branch conditions, and its corresponding program variant. Then, we reported the above information to developers using their corresponding bug tracking systems and evaluated the usefulness of RIDDLE based on developers' feedback. Based on the reported information, developers can reproduce the failure-introducing conditions and inputs to facilitate issue diagnosis. Specifically, we listed the unrestored branch conditions in the program variant and asked developers to further check whether they could be triggered in reality. For the tests with unrealistic branch condition mutations, we labeled them as false positives. We only labeled the confirmed bug reports as true positives in our evaluation.

B. RQ1: Effectiveness of RIDDLE

Table II shows the effectiveness of RIDDLE in covering the risky methods of 20 DC issues identified from the above 19 subjects, where column "N_{dis}" represents the average invocation path length from entry methods to the risky methods; column "N_{br}" represents the average number of conditional branches on the invocation paths from entry methods to the risky methods; columns "EVOSUITE", "EVOSUITE & Fitness" and "RIDDLE" represent the number of tests generated by the above three techniques, which can trigger the risky methods; N_{lg} represents the number of program variants with infeasible behaviors introduced by short-circuiting which can be solved by condition restoration operations.

TABLE II: Effectiveness of RIDDLE in covering risky methods of 20 DC issues identified from 19 subjects (averaged over 10 runs)

ID	Project	N_{dis}	N_{br}	Evosuite	Evosuite& Fitness	RIDDLE	N_{lg}
1	Azure storage	19.4	16.3	0	0	72.4	5.5
2	Google/truth	15.3	13.9	0	0	15.2	2.2
3	Ff4j	6.1	7.5	0	2.5	10.3	0.0
4		18.2	14.2	0	0	13.7	1.0
5	Incubator dubbo	23.6	15.7	0	0	12.5	3.5
6	Jetbrick/template	14.1	11.3	0	0	15.4	4.3
7	Webcam/capture	15.5	12.8	0	0	1.0	0
8	Blueflood	17.4	14.7	0	0	1.0	0
9	Vertx/swagger	19.2	10.2	0	0	20.0	0
10	Webmagic	26.9	13.5	0	0	8.5	3.0
11	Apache Storm	13.9	9.7	0	6.5	17.0	1.0
12	Htm.java	12.2	8.2	0	0	19.3	0
13	Incubator servicecomb	16.5	11.5	0	0	16.5	5.0
14	Selendroid	17.3	13.9	0	0	14.3	1.0
15	HotelsDotCom/styx	14.1	10.2	0	0	2.4	0
16	Wisdom	13.7	7.3	0	0	12.4	0
17	Geowave	17.3	8.4	0	0	13.2	0
18	Mayocat shop	16.2	10.2	0	0	18.4	3.5
19	Vipshop/saturn	15.5	10.5	0	0	4.5	0
20	St-js	12.5	16.4	0	0	5.5	1.0

As shown in Table II, the N_{dis} of the above subjects varies from 6.12 to 26.90, and N_{br} varies from 7.34 to 16.34. These two values indicate the challenges of triggering the risky methods. Despite these challenges, RIDDLE can successfully generate tests reaching the target branches for DC issues in all the subjects. However, EVOSUITE could not cover the risky methods because its branch coverage criteria fail to guide the search to explore the methods that lead to the risky ones. In contrast, EVOSUITE & Fitness outperforms EVOSUITE for the `Ff4j` and `Apache Storm` projects, which have shorter invocation paths (smaller N_{dis}) with fewer complex conditional branches (smaller N_{br}). Using the fitness function defined in our approach, EVOSUITE & Fitness avoids exploring some complex conditional branches. Compared with RIDDLE, it still fails to reach the branches guarded by unsolvable conditions without short-circuiting these conditions.

RIDDLE indeed provides an effective strategy to collect runtime information for DC issues, by forcing the generated tests to reach the target branches. However, the challenges inhibiting test generation tools from achieving high code coverage, (e.g., creation of complex objects), may appear in non-branch condition statements. In these cases, the tests are unlikely generated by RIDDLE, if they cannot be avoided by condition mutation or the guidance of fitness function during evolution process. Taking a DC issue in project `Htm.java` as an example, there are 63 invocation paths from boundary classes to risky methods. Among them, 43 have to pass through the statements shown in Figure 9, which require a browser's local installation path. Nevertheless, such environment dependencies and complex object are difficult to construct automatically. Consequently, on average, RIDDLE only generates 19.3 tests triggering the crashes, over 10 runs.

In some cases, short-circuiting branch conditions can cause infeasible behaviors. As shown in Table II, after condition mutation, such side effects are introduced in five program variants of project `Azure storage`. For instance, if RIDDLE short-

```
1. File connectFile = prepareConnectFile(server.getUrl());
2. BrowserInstallation installation = browserLocator.findBrowserLocationOrFail();
3. commandLine = new CommandLine(installation.launcherFilePath(),
                    connectFile.getAbsolutePath());
4. commandLine.executeAsync(); ...
```

Fig. 9: Code snippet in project `Htm.java`

```
1. int bodyPos = 0;
2. Buffer wireBuffer = null;
      ...
3. void decodeFromWire(int bodyPos, Buffer wireBuffer){
4.           if(wireBuffer!=null && bodyPos!=0){
5.                 int length = wireBuffer.getInt(bodyPos);
6.           }
7.     }   ...
```

Fig. 10: Code snippet in project `Azure storage`

circuits the branch condition in `Line 4` of the code snippet shown in Figure 10, we trigger a `NullPointerException` when executing the statement in `Line 5`, in the case that variable `wireBuffer` is a null-initialized object variable. To address this problem, RIDDLE restores the branch condition in `Line 4`, and then reaches the target branch (`Line 5`) by passing a non-null object of type `Buffer` to method `decodeFromWire`.

C. RQ2: Usefulness of RIDDLE

Table III presents the results of DC issues reported by RIDDLE, where column "$|\mathcal{R}_i|$" represents the number of risky methods in each reported DC issue; column "$|\mathcal{CL}_i|$" represents the number of boundary classes that can reach to risky methods; column "$N_{\text{ur}}/N_{\text{total}}$" represents the ratio of unrestored branch conditions to the total number of branches on the invocation path covered by reported tests. By communicating with developers, we collected developers' feedback on the reported DC issues and summarized them into two findings as follows.

First, among the 20 reported issues, 15 (75%) were confirmed by developers as real issues within a few days, thereby we labeled them as true positives. Ten out of the 15 confirmed issues (67%) were quickly fixed, and five out of the 15 confirmed issues (33%) were in the process of being fixed.

TABLE III: Results of 20 DC issues reported by RIDDLE

ID	Project	Issue ID	$\lvert\mathcal{R}_i\rvert$	$\lvert\mathcal{CL}_i\rvert$	$N_{\text{ur}}/N_{\text{total}}$
1	Azure storage	Issue #345 [53]♢	2	4	1/13
2	Google/truth	Issue #473 [54]♢	1	3	2/11
3	Ff4j	Issue #309 [14]♢	1	3	0/5
4		Issue #315 [55]♢	3	5	2/12
5	Incubator dubbo	Issue #2134 [56]♢	3	6	3/13
6	Jetbrick/template	Issue #39 [57]♢	2	3	2/9
7	Webcam/capture	Issue #653 [58]♠	3	6	4/10
8	Blueflood	Issue #829 [59]	2	5	4/12
9	Vertx/swagger	Issue #102 [60]	1	2	2/9
10	Webmagic	Issue #816 [61]	3	4	3/15
11	Apache Storm	STORM-3171 [62]♢	2	3	0/8
12	Htm.java	Issue #540 [63]♠	1	2	2/10
13	Incubator service	Issue #858 [64]♣	2	3	2/8
14	Selendroid	Issue #1169 [65]	1	2	2/11
15	HotelsDotCom/styx	Issue #227 [66]♢	1	3	2/9
16	Wisdom	Issue #573 [67]♠	1	2	1/6
17	Geowave	Issue #1371 [68]♠	2	3	2/7
18	Mayocat shop	Issue #272 [69]♢	2	2	3/9
19	Vipshop/saturn	Issue #477 [70]♢	1	2	3/11
20	St-js	Issue #146 [71]♠	2	3	2/10

♢: The issues have already been fixed.
♠: The issues were confirmed and in the process of being fixed.
♣: False positive results.

Moreover, among the five unconfirmed issues, only one was labeled as false positive and the others are not confirmed mainly due to the inactive maintenance of the corresponding release versions. For the ten fixed DC issues, developers agreed that the shadowed methods indeed can cause runtime exceptions with the reported stack traces, so we receive quick feedback from these developers. Encouragingly, in `Issue #227` [66], one developer complimented on the usefulness of our issue reports:

"Thanks for the bug report! The provided stack trace was very helpful for us to further investigate this bug. I have opened a PR for this. Please feel free to have a look. "

We also got a positive feedback, in `Issue #345` [53]:

"The stack trace is useful information because this is how we can decide the feasibility of reproducing the DC issue."

Meanwhile, for `Issue #858` [64], developers acknowledged the risks associated with the dependency conflicts reported by RIDDLE. However, they mark the bug report as a false positive after checking the mutated branch conditions. The developers noted that the runtime exceptions might never happen because the statements in this branch are dead code and the condition is not satisfiable in reality.

Second, developers agreed with the condition mutation strategy for triggering the possible crashes caused by DC issues. In particular, one experienced developer [72] from `Microsoft` responded in `Issue #345` [53] that:

"Although the way you arrived at the parse error is artificial, I believe that a similar parse error in the code is achievable in actual usage, presumably leading to the runtime exceptions due to dependency conflicts (if indeed such a conflict exists in the application code). The tests did bring to our attention to update the conflicting library to the latest version, so I would not call it a false positive."

The high confirmation rate of the reported DC issues indicates that developers accepted the reported runtime information with some unrestored conditions and were likely able to use their domain knowledge to figure out them.

Moreover, the low values for the column "$N_{\text{ur}}/N_{\text{total}}$" in Table III indicates that in our reported issues, most of the branch conditions on execution traces can be restored by RIDDLE, which are very close to the actual runtime information.

Finally, developers have expressed great interest in including RIDDLE as part of their projects. For instance, we received a feedback in Issue #473 [54]:

"If you have a test generation tool that can run continually on `Truth` and other `Google` projects, or if you have other information to share, I'd be happy to hear about it."

In addition, after resolving Issue #540 [63], we accepted an inspiring invitation from the developers of `Htm.java` to continuously maintain the dependencies of third party libraries in their project, with the aid of RIDDLE.

V. THREATS TO VALIDITY

Validity of bug reports. We rely on developers' feedback to validate our submitted bug reports. In general, there may be disagreement between the developers on the validity of the

bug reports. However, we did not encounter such disagreement for all the evaluated subjects. Therefore, we believe that the positive feedback that we received are strong indications of the usefulness of our approach.

Condition restoration strategy. After short-circuiting branch conditions, RIDDLE restores them step by step in MLCFG following a top-down strategy. In our evaluation, the effectiveness of the proposed condition restoration strategy has been confirmed by developers. However, the proposed restoration strategy may generate invalid tests if path conditions interact with each other.

Limitation for detecting diverse types of DC issues. The proposed test generation technique focuses on crash-related DC issues due to the reference of shadowed features or classes. However, in some cases, conflicts could be caused by the changes in semantics, performance or other attributes of the duplicated libraries presented on a project's classpath. RIDDLE does not analyze the above manifestations of DC issues, and may miss some DC issues.

VI. Related work

Dependency Conflict. Patra et al. [73] studied the library conflicts problem specifically for JavaScript language, and proposed a tool, CONFLICTJS, to automatically detect conflicts. Wang et al. [7] conducted the first empirical study to investigate the manifestation and the fixing patterns of DC issues in Java projects. Based on their findings, they proposed a tool, DECCA, to assess severity of potential DC issues using static analysis. RIDDLE differs from DECCA in several aspects: (1) DECCA detects the existence of DC issues using static analysis, while RIDDLE leverages DECCA to identify risky methods causing DC issues, and combines dynamic analysis (i.e., automatic test generation), condition mutation, search strategies and condition restoration to obtain stack traces reaching these methods; and (2) DECCA only categorizes the severity levels of DC issues, while RIDDLE provides stack trace information and the failure-introducing conditions to help developers reproduce and debug DC issues. So, RIDDLE and DECCA complement each other.

Stack trace analysis and fault localization. Prior studies provided empirical evidence that stack traces are useful for developers to fix bugs [9], [10], [15], [21], [22], [26], [74], [75] Meanwhile, Coelho et al. [76] conducted a detailed empirical study of 6,000 Java exception stack traces and explored the common causes for system crashes. Inspired by these studies, RIDDLE generates stack traces to explain DC issues. The positive feedback from developers, who get the stack trace information in our evaluation, further confirm the findings of prior studies. The concept of *predicate switching* (i.e., flipping branches outcome during execution for locating faults) [77] is similar to the condition mutations in RIDDLE. Moreover, predicate switching leverages dynamic instrumentation for controlling the program execution, while RIDDLE mutates branch conditions and generate tests based on the mutated program variants.

Automated Test Generation. Many test generation techniques have been proposed to detect software bugs [78]–[104]. Zhang et al. [29] implemented an isomorphic regression testing approach named ISON, which forced the existing tests to execute the originally uncovered code by modifying the isomorphic code of two program versions. The behaviors (i.e., test outputs) of the modified programs were then compared to check if abnormal behaviors were induced in the new version. Soltani et al. [105] presented EVOCRASH, a post-failure approach which used data from crash stack traces as input and combined with a guided genetic algorithm to search for a test case that could trigger the target crash. Different from prior work, RIDDLE is the first approach that generate tests with buggy frame in the stack traces to examine DC issues without known crash information. It short-circuits the branch conditions to trigger the risky methods caused by DC issues and then addresses the side effects by condition restoration.

VII. Conclusion and Future work

In this paper, we developed RIDDLE, the first automatic approach that generates tests and collects crashing stack traces for the projects with DC issues. To overcome the complex branch conditions, RIDDLE combines condition mutation and search strategies to generate tests reaching the risky methods that cause DC issues. Furthermore, it restores solvable branch conditions to capture more precise runtime information. We applied RIDDLE on real-world projects. Positive feedback were obtained from project developers on RIDDLE's results. Most of the submitted bug reports were confirmed by developers with a majority of them readily fixed. Also, developers showed great interests in RIDDLE, and acknowledged that stack traces are very helpful for their debugging tasks. These feedback from developers demonstrate the practical usefulness of RIDDLE.

An alternative mutation strategy is to perform selective mutation of branch conditions that cannot be solved by EVOSUITE to reduce the cost of condition restoration. As identifying the unsolvable branch conditions reaching to target branches may incur more overhead, we choose to mutate all branch conditions to drive the program executions towards the target branches in one step. We leave the investigation of the efficiency of this alternative as future work.

Acknowledgment

The authors express thanks to the anonymous reviewers for their constructive comments. Part of the work was conducted during the first author's internship at HKUST in 2018. The work is supported by the National Natural Science Foundation of China (Grant Nos. 61374178, 61603082 and 61402092), the Fundamental Research Funds for the Central Universities (Grant No. N171704004), the Hong Kong RGC/GRF grant 16202917, MSRA grant, and the Science and Technology Innovation Committee Foundation of Shenzhen (Grant No. JCYJ20170817110848086). Dr. Rongxin Wu is the first corresponding author of this paper.

REFERENCES

[1] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 356–367.

[2] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: an evolutionary study," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1275–1317, 2015.

[3] W. Ming, L. Yepang, W. Rongxin, X. Xuan, C. Shing-Chi, and S. Zhendong, "Exposing Library API Misuses via Mutation Analysis," in *Proceedings of the 41th International Conference on Software Engineering*, ser. ICSE 2019, 2019.

[4] J. Gosling, B. Joy, and G. Steele, *The Java language specification*. Addison-Wesley Professional, 2000.

[5] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine, "Glacier: transitive class immutability for java," in *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*. IEEE, 2017, pp. 496–506.

[6] S. Liang and G. Bracha, "Dynamic class loading in the java virtual machine," *Acm sigplan notices*, vol. 33, no. 10, pp. 36–44, 1998.

[7] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the dependency conflicts in my project matter?" in *Proceedings of the 2018 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, 2018, pp. 1–12.

[8] "Maven," http://maven.apache.org/, 2018, accessed: 2018-08-18.

[9] R. Wu, M. Wen, S.-C. Cheung, and H. Zhang, "Changelocator: locate crash-inducing changes based on crash reports," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2866–2900, 2018.

[10] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: locating crashing faults based on crash stacks," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 204–214.

[11] "Issue #57," https://github.com/FasterXML/jackson-module-jaxb-annotations/issues/57, 2018, accessed: 2018-08-18.

[12] "Issue #8706," https://github.com/spring-projects/spring-boot/issues/8706, 2018, accessed: 2018-08-18.

[13] "Issue #501," https://github.com/apollographql/apollo-android/issues/501, 2018, accessed: 2018-08-18.

[14] "Issue #309," https://github.com/ff4j/ff4j/issues/309, 2018, accessed: 2018-08-18.

[15] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 118–121.

[16] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.

[17] "Azure storage," https://docs.microsoft.com/en-us/java/api/overview/azure/storage?view=azure-java-stable, 2018, accessed: 2018-08-18.

[18] "Google/truth," http://google.github.io/truth/, 2018, accessed: 2018-08-18.

[19] "Webmagic," http://webmagic.io/, 2018, accessed: 2018-08-18.

[20] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 416–419.

[21] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 308–318.

[22] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What makes a good bug report?" *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 618–643, 2010.

[23] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: a case study of bug reports," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1. IEEE, 2010, pp. 505–514.

[24] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 117–128.

[25] L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline, and G. Venolia, "Debugging revisited: Toward understanding the debugging needs of contemporary software developers," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 2013, pp. 383–392.

[26] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 181–190.

[27] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: a retrospective," in *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, Oct. 2011. [Online]. Available: http://www.bodden.de/pubs/lblh11soot.pdf

[28] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2005, pp. 365–381.

[29] J. Zhang, Y. Lou, L. Zhang, D. Hao, L. Zhang, and H. Mei, "Isomorphic regression testing: executing uncovered branches without test augmentation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 883–894.

[30] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon, "From genetic to bacteriological algorithms for mutation-based testing," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 73–96, 2005.

[31] A. Aleti, I. Moser, and L. Grunske, "Analysing the fitness landscape of search-based software testing problems," *Automated Software Engineering*, vol. 24, no. 3, pp. 603–621, 2017.

[32] M. Harman, R. Hierons, and M. Proctor, "A new representation and crossover operator for search-based optimization of software modularization," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc., 2002, pp. 1351–1358.

[33] "Asm," https://asm.ow2.io/, 2018, accessed: 2018-08-18.

[34] "Ff4j," http://ff4j.org/, 2018, accessed: 2018-08-18.

[35] "Incubator dubbo," http://dubbo.incubator.apache.org/en-us/, 2018, accessed: 2018-08-18.

[36] "Jetbrick/template," http://subchen.github.io/jetbrick-template/, 2018, accessed: 2018-08-18.

[37] "Webcam/capture," http://webcam-capture.sarxos.pl/, 2018, accessed: 2018-08-18.

[38] "Blueflood," http://www.blueflood.io, 2018, accessed: 2018-08-18.

[39] "Vertx/swagger," https://vertx.io/blog/presentation-of-the-vert-x-swagger-project/, 2018, accessed: 2018-08-18.

[40] "Apache storm," http://storm.apache.org/, 2018, accessed: 2018-08-18.

[41] "Htm.java," https://numenta.org/, 2018, accessed: 2018-08-18.

[42] "Incubator service," http://servicecomb.incubator.apache.org/, 2018, accessed: 2018-08-18.

[43] "Selendroid," http://selendroid.io, 2018, accessed: 2018-08-18.

[44] "Hotelsdotcom/styx," https://github.com/Netflix/Hystrix/wiki, 2018, accessed: 2018-08-18.

[45] "Wisdom," http://wisdom-framework.org, 2018, accessed: 2018-08-18.

[46] "Geowave," http://locationtech.github.io/geowave/, 2018, accessed: 2018-08-18.

[47] "Mayocat shop," https://github.com/jvelo/mayocat-shop/, 2018, accessed: 2018-08-18.

[48] "Vipshop/saturn," https://vipshop.github.io/Saturn/#/, 2018, accessed: 2018-08-18.

[49] "St-js," http://st-js.org/, 2018, accessed: 2018-08-18.

[50] G. Fraser, J. M. Rojas, J. Campos, and A. Arcuri, "Evosuite at the sbst 2017 tool competition," in *Ieee/acm International Workshop on Search-Based Software Testing*, 2017, pp. 39–42.

[51] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[52] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *International Symposium on Search Based Software Engineering*. Springer, 2011, pp. 33–47.

[53] "Issue #345," https://github.com/Azure/azure-storage-java/issues/345, 2018, accessed: 2018-08-18.

[54] "Issue #473," https://github.com/google/truth/issues/473, 2018, accessed: 2018-08-18.

[55] "Issue #315," https://github.com/ff4j/ff4j/issues/315, 2018, accessed: 2018-08-18.

[56] "Issue #2134," https://github.com/apache/incubator-dubbo/issues/2134, 2018, accessed: 2018-08-18.

[57] "Issue #39," https://github.com/subchen/jetbrick-template-2x/issues/39, 2018, accessed: 2018-08-18.

[58] "Issue #653," https://github.com/sarxos/webcam-capture/issues/653, 2018, accessed: 2018-08-18.

[59] "Issue #829," https://github.com/rackerlabs/blueflood/issues/829, 2018, accessed: 2018-08-18.

[60] "Issue #102," https://github.com/phiz71/vertx-swagger/issues/102, 2018, accessed: 2018-08-18.

[61] "Issue #816," https://github.com/code4craft/webmagic/issues/816, 2018, accessed: 2018-08-18.

[62] "Storm-3171," https://issues.apache.org/jira/browse/STORM-3171, 2018, accessed: 2018-08-18.

[63] "Issue #540," https://github.com/numenta/htm.java/issues/540, 2018, accessed: 2018-08-18.

[64] "Issue #858," https://github.com/apache/incubator-servicecomb-java-chassis/issues/858, 2018, accessed: 2018-08-18.

[65] "Issue #1169," https://github.com/selendroid/selendroid/issues/1169, 2018, accessed: 2018-08-18.

[66] "Issue #227," https://github.com/HotelsDotCom/styx/issues/227, 2018, accessed: 2018-08-18.

[67] "Issue #573," https://github.com/wisdom-framework/wisdom/issues/573, 2018, accessed: 2018-08-18.

[68] "Issue #1371," https://github.com/locationtech/geowave/issues/1371, 2018, accessed: 2018-08-18.

[69] "Issue #272," https://github.com/jvelo/mayocat-shop/issues/272, 2018, accessed: 2018-08-18.

[70] "Issue #477," https://github.com/vipshop/Saturn/issues/477, 2018, accessed: 2018-08-18.

[71] "Issue #146," https://github.com/st-js/st-js/issues/146, 2018, accessed: 2018-08-18.

[72] "Microsoft developer," https://github.com/mirobers, 2018, accessed: 2018-08-18.

[73] J. Patra, P. N. Dixit, and M. Pradel, "Conflictjs: finding and understanding conflicts between javascript libraries," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 741–751.

[74] M. Wen, R. Wu, and S. Cheung, "Locus: Locating bugs from software changes," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, 2016.

[75] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE 2018. IEEE, 2018, pp. 187–198.

[76] R. Coelho, L. Almeida, G. Gousios, A. Van Deursen, and C. Treude, "Exception handling bug hazards in android," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1264–1304, 2017.

[77] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 272–281.

[78] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[79] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with java pathfinder," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 97–107, 2004.

[80] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 2007, pp. 815–816.

[81] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "Mujava: a mutation system for java," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 827–830.

[82] C. Csallner and Y. Smaragdakis, "Jcrasher: an automatic robustness tester for java," *Software: Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[83] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 213–223.

[84] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[85] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon, "Automatic and scalable t-wise test case generation strategies for software product lines," in *International Conference on Software Testing*. Springer Lecture Notes in Computer Science (LNCS), 2010.

[86] M. Christakis and P. Godefroid, "Ic-cut: A compositional search strategy for dynamic test generation," in *Model Checking Software*. Springer, 2015, pp. 300–318.

[87] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in web applications using dynamic test generation and explicit-state model checking," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, p. 474, 2010.

[88] E. P. Enoiu, D. Sundmark, A. Čaušević, R. Feldt, and P. Pettersson, "Mutation-based test generation for plc embedded software using model checking," in *IFIP International Conference on Testing Software and Systems*. Springer, 2016, pp. 155–171.

[89] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *International Conference on Software Engineering*, 2010, pp. 225–234.

[90] G. Denaro, A. Gorla, and M. Pezze, "Datec: Contextual data flow testing of java classes," in *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. IEEE, 2009, pp. 421–422.

[91] S. Hallé, E. La Chance, and S. Gaboury, "Graph methods for generating test cases with universal and existential constraints," in *IFIP International Conference on Testing Software and Systems*. Springer, 2015, pp. 55–70.

[92] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," in *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4. ACM, 2002, pp. 123–133.

[93] Y. Kim, S. Hong, B. Ko, D. L. Phan, and M. Kim, "Invasive software testing: Mutating target programs to diversify test exploration for high test coverage," in *Software Testing, Verification and Validation (ICST), 2018 IEEE 11th International Conference on*. IEEE, 2018, pp. 239–249.

[94] F. Zaraket, W. Masri, M. Adam, D. Hammoud, R. Hamzeh, R. Farhat, E. Khamissi, and J. Noujaim, "Guicop: Specification-based gui testing," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 747–751.

[95] A. Arcuri, "Test suite generation with the many independent objective (mio) algorithm," *Information and Software Technology*, 2018.

[96] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 49–60.

[97] M. Fazzini, M. Prammer, M. d'Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 141–152.

[98] A. Bertolino, J. Gao, E. Marchetti, and A. Polini, "Automatic test data generation for xml schema-based partition testing," in *Proceedings of the second international workshop on automation of software test*. IEEE Computer Society, 2007, p. 4.

[99] P. D. Marinescu and C. Cadar, "Katch: high-coverage testing of software patches," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 235–245.

[100] D. Giannakopoulou, N. Rungta, and M. Feary, "Automated test case generation for an autopilot requirement prototype," in *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*. IEEE, 2011, pp. 1825–1830.

[101] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in udita," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 225–234.

[102] D. H. Bushnell, C. Pasareanu, and R. M. Mackey, "Automatic testcase generation for flight software," 2008.

[103] J. Burnim, K. Sen, and C. Stergiou, "Testing concurrent programs on relaxed memory models," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 122–132.

[104] S. Kundu, M. K. Ganai, and C. Wang, "Contessa: Concurrency testing augmented with symbolic analysis," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 127–131.

[105] M. Soltani, A. Panichella, and A. van Deursen, "A guided genetic algorithm for automated crash reproduction," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 209–220.