Will Dependency Conflicts Affect My Program's Semantics?

Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung *Senior Member, IEEE*, Hai Yu[†], Chang Xu *Senior Member, IEEE*, Zhiliang Zhu *Member, IEEE*

Abstract—Java projects are often built on top of various third-party libraries. If multiple versions of a library exist on the classpath, JVM will only load one version and shadow the others, which we refer to as *dependency conflicts*. This would give rise to *semantic conflict* (SC) issues, if the library APIs referenced by a project have identical method signatures but inconsistent semantics across the loaded and shadowed versions of libraries. SC issues are difficult for developers to diagnose in practice, since understanding them typically requires domain knowledge. Although adapting the existing test generation technique for dependency conflict issues, RIDDLE, to detect SC issues is feasible, its effectiveness is greatly compromised. This is mainly because RIDDLE randomly generates test inputs, while the SC issues typically require specific arguments in the tests to be exposed. To address that, we conducted an empirical study of 316 real SC issues to understand the characteristics of such specific arguments in the test cases that can capture the SC issues. Inspired by our empirical findings, we propose an automated testing technique SENSOR, which synthesizes test cases using ingredients from the project under test to trigger inconsistent behaviors of the APIs with the same signatures in conflicting library versions. Our evaluation results show that SENSOR is effective and useful: it achieved a $Precision$ of 0.898 and a $Recall$ of 0.725 on open-source projects and a $Precision$ of 0.821 on industrial projects; it detected 306 semantic conflict issues in 50 projects, 70.4% of which had been confirmed as real bugs, and 84.2% of the confirmed issues have been fixed quickly.

Index Terms—Third-party Libraries, Test Generation, Empirical Study.

1 INTRODUCTION

BUILDING software projects on top of third-party libraries is a common practice to save development cost and improve software quality [1], [2], [3], [4]. However, the heavy dependencies on third-party libraries often induce dependency conflict issues [5]. When multiple versions of the same library class are present on the `classpath`, the Java class loader will load only one version and shadow the others [6]. If the loaded version has inconsistent implementations with the intended but shadowed versions, dependency conflict issues will occur, inducing risks of runtime exceptions or unexpected program behaviors.

The state-of-the-art techniques [5], [7] for detecting dependency conflict issues mainly focus on specific categories of the issues, such as `ClassNotFoundException` and `NoSuchMethodError`, which happen when the loaded library versions do not cover all the APIs referenced by the client projects. One limitation of these techniques is that they cannot identify the dependency conflict issues that arise from referencing those APIs with identical method signatures but inconsistent behaviors across multiple library versions [8], [7]. We refer to such issues as Semantic Conflict issues (SC issues for short). In the strict sense, the projects suffer from SC issues can pass the compilation and build process, and such issues are only exposed as inconsistent behaviors at runtime. Figure 1 gives a real example of SC issues. On the `classpath` of the project `Openstack-java-sdk 3.2.5`, there are two versions of the library `Jackson-core-asl`, namely, *Version 1.9.4* and *Version 1.9.13*. In the example, Java class loader loads *Version 1.9.13* but shadows *Version 1.9.4*. As shown in the code snippet, the method `createClientExecutor()` in the project will transitively invoke `validate(ClientResponse)` of the library `Jackson-core-asl`. However, the implementations of `validate(ClientResponse)` are semantically inconsistent between the two versions. The project was originally designed to use *Version 1.9.4* of `Jackson-core-asl`, which is unfortunately shadowed. Although there will be no runtime exceptions in such cases, the semantic inconsistency of library method implementations will inappropriately affect the variable states of the client project via the invocation of the concerned methods, leading to unexpected program behaviors.

SC issues arise from code changes of API implementations, which are common in popular libraries. Many of these changes are too subtle for developers to understand their effects on program semantics [2]. Existing test suites may not help effectively expose such differences neither. As such,

Ying Wang, Chao Wang, Hai Yu, and Zhiliang Zhu are with the Software College, Northeasthern University, China. E-mail: wangying@swc.neu.edu.cn, wangc_neu@163.com, {yuhai, zzl}@mail.neu.edu.cn.
Rongxin Wu is with Department of Cyber Space Security, Xiamen University, China. E-mail: wurongxin@xmu.edu.cn.
Ming Wen is with School of Cyber Science and Engineering, Huazhong University of Science and Technology, China. E-mail: mwenaa@hust.edu.cn.
Yepang Liu is with Department of Computer Science and Engineering, and Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation, Southern University of Science and Technology, Shenzhen, China. E-mail: liuyp1@sustech.edu.cn.
Shing-Chi Cheung is with Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, China. E-mail: scc@cse.ust.hk.
Chang Xu is with State Key Lab for Novel Software Technology, and Department of Computer Science and Technology, Nanjing University, China. E-mail: changxu@nju.edu.cn.
[†]*Hai Yu is the corresponding author.*
Manuscript received May 9, 2020; revised August 26, 2020.

```
// Method in shadowed library Jackson-core-asl.jar - 1.9.4
102: public boolean validate(ClientResponse<T>  response){
103:     return true;
104: }
// Method in loaded library Jackson-core-asl.jar - 1.9.13
102: public boolean validate(ClientResponse<T>  response){
103:     return response.getStatus();
104: }
// Entry method in class connector.RESTEasyConnector of the client project
134: public ClientExecutor createClientExecutor(){
135:     if (request.status())
136:         return request.getDefaultExecutor();
137:     else
138:         return null;
139: }
// Method referenced by entry method (defined in jackson-jaxrs.jar - 1.9.4)
56: public boolean status(){
57:     return (Paster.validate(response) && getAccesss())
58: }
```

Fig. 1: `Issue #214` [9] in `Openstack-java-sdk 3.2.5`

SC issues are difficult to diagnose. For example, a developer left the following comment in the pull request [10] of the aforementioned issue:

> *"I have encountered these types of semantic inconsistency issues lots of times when dealing with dependency conflicts. When such issues happen, signature changes can be detected by static analyzers. However, semantic changes would be more difficult to detect. Empirically, developers diagnose them by reading the git history of the library or dynamic testing."*

Detecting SC issues typically requires rich domain knowledge to discern the subtle differences in API implementations, which is a non-trivial task. Therefore, an automated technique to detect SC issues is highly desirable. We note that the most relevant and recent technique is RIDDLE [7]. It was designed to verify dependency conflict issues caused by the missing of classes or methods. RIDDLE can generate tests to drive the execution of a client project towards the target call sites that could induce dependency conflict issues. While it seems possible to adapt RIDDLE using the idea of differential testing (i.e., comparing the runtime behavior of a target API across multiple library versions) to detect SC issues, our trial on 70 open-source projects shows that this technique is not that effective as anticipated (see Section 2.3). It is mainly because merely reaching the call sites of a target API and invoking it with random arguments can hardly trigger the inconsistent behaviors of the API across different versions. As such, many SC issues, whose manifestation requires specific arguments (referred to as *divergence arguments* in this paper), cannot be effectively exposed. This motivates us to design a more effective testing technique to detect SC issues.

As discussed above, an obvious challenge in detecting SC issues via testing is to generate divergence arguments to trigger inconsistent API behaviors across different library versions. To address this challenge, we performed an in-depth study of 316 real SC issues collected from open-source Java projects to understand the characteristics of divergence arguments in the test cases that could expose these issues. The study revealed several interesting findings. First, to generate desirable test inputs for detecting SC issues, almost all (99.9%) the API calls and (95.5%) object constructors take at least one argument, and, for 98.5% of these APIs and constructors, at least one of their arguments has specific values that can hardly be generated by random techniques such as RIDDLE. Second, we observed three common patterns to produce divergence arguments for API calls and object constructors in the test cases. Third, we found that, for 61.5% of our analyzed API calls and object constructors, their divergence arguments can be directly obtained from the source code of the client project. Fourth, for the APIs and constructors in the test cases whose arguments cannot be found in the source code of the client project, we replaced the arguments with other compatible values that can be found in the client project's source code and discovered that 119 out of 193 (61.7%) such revised test cases could still capture SC issues.

Inspired by our empirical findings, our idea of seeding divergence arguments for triggering SC issues is to synthesize concerned object constructors and API calls with these arguments from the source code of client projects. Specifically, we synthesize an object constructor/API call with divergence arguments by distilling the set of legitimate API usages and the values of its arguments from the source code. We refer to the set as the constructor's/API's *invocation context*. We implemented our idea into an automated testing technique, SENSOR. Given a client project to analyze, SENSOR first extracts the invocation contexts of each object constructor and API call from the source code and leverages them to construct a class instance pool and an API argument pool. Combining a seeding strategy of class instances/API calls with EVOSUITE, it then generates tests to trigger the concerned library APIs and checks whether they behave consistently across different versions. In our approach, SENSOR does not simply report all detected behavioral inconsistencies as bugs. Instead, it pinpoints the differences in variable states of a project under analysis and provides such fine-grained information to help developers further diagnose SC issues.

We evaluated SENSOR using 140 open-source projects on `GitHub` and 10 industrial projects from `Neusoft` Co. Ltd (SSE: 600718) [11]. SENSOR achieved a *Precision* of 0.898 and a *Recall* of 0.725 on open-source projects, and a precision of 0.821 on industrial projects. SENSOR detected 306 real SC issues from 50 open-source projects. We reported these issues to the developers of the corresponding projects and detailed the issues' impact on program behaviors. So far, 70.4% of our reported issues have been confirmed by the developers as real bugs, and 84.2% of the confirmed issues have been fixed quickly. Most of the confirmed issues are from popular projects such as `Rest-assured` [12] and `Java-design-patterns` [13]. From the feedback on our reported issues (see Section 5.3), we observed that developers acknowledged the pervasiveness of SC issues and the necessity of a testing technique to diagnose such issues. They also expressed great interests in using SENSOR. These results demonstrate the effectiveness and usefulness of SENSOR. In summary, we make four major contributions in this paper:

- An empirical study of 316 real SC issues for exploring the characteristics of test cases that can expose SC issues.
- A fully automated technique, SENSOR, for detecting SC

```java
// Project: Hmily 2.0.0
// Loaded version in library org.jboss.netty 3.2.5
public String getPrefixFromTerm(String term, TimerContext context) {
    if ( StringUtils.isNotBlank(term) ) {
        String[] parts = term.split(":");
        return parts[0];
    }
    if ( enableTimers && context != null ) {
        context.stop();
    }
    return null;
}
```

```java
// Shadowed version in library io.netty 3.10.5
public String getPrefixFromTerm(String term, TimerContext context) {
    if ( StringUtils.isNotBlank(term) ) {
        String[] parts = term.split(":");
        return parts[0];
    }
    stopTimer(context);
    return null;
}
private void stopTimer(TimerContext context) {
    if ( enableTimers && context != null ) {
    context.stop();
    }
}
```

Fig. 2: A motivating example

issues.

- A benchmark dataset for assessing SENSOR and similar approaches for detecting the issues induced by semantic inconsistencies of library APIs across different versions.
- A systematic analysis and discussions of SC issues' impacts on program behaviors.

Our tool and dataset are available at: https://sensordc.github.io/.

2 PRELIMINARIES

2.1 Motivation

To roughly estimate the scale of SC issues, we statically detected the semantic inconsistency of the conflicting API pairs by comparing their *code structures* in terms of call graphs and control flow graphs. We first collected 1,654 Java projects from `GitHub` based on two criteria: (1) it has achieved over 50 stars or forks (popularity); and (2) it is built on the `Maven` platform. Then, we compared the code structures of the conflicting API pairs in these projects and labeled them as potential SC issues if their code structures are different. The results showed that 73.1% of the projects contain at least one potential SC issue. Each of them contains on average 20 conflicting library API pairs that potentially cause SC issues.

The static analysis of SC issues based on different code structures can be highly imprecise for some projects. Figure 2 illustrates a false positive SC issue found by the static approach. There are two versions of the class `Netty.bootstrap.ServerBootstrap` on the `classpath` of the project `Hmily-2.0.0` [14], which are included by the libraries `org.jboss.netty 3.2.5` and `io.netty 3.10.5`, respectively. Due to `Maven`'s *first declaration wins* strategy, only the method `getPrefixFromTerm()` declared in the library `org.jboss.netty 3.2.5` is loaded and invoked by the client project. Although the call graphs of these two versions differ, the method in `io.netty 3.10.5` was simply a code refactoring in `org.jboss.netty 3.2.5` that did not affect the program semantics.

Validation of SC issues is non-trivial. It requires domain knowledge to understand the implementations of the client project and its libraries. This motivates us to validate SC issues using automatically generated tests.

2.2 Problem formulation

To formulate our research problem, we introduce the following concepts. In particular, we let C_i' be a shadowed class version and C_i be the actually-loaded class version, and use $C :: m$ to denote an API m of class C.

Definition 1. (Conflicting API pair): Let $C_i' :: m_k$ be an API included in the shadowed class version and referenced by the client project $\mathcal{H}$, and $C_i :: m_k$ be an API belonging to the actually-loaded class version, where m_k represents the method signature. If $C_i' :: m_k$ and $C_i :: m_k$ share the same signature, we consider $C_i' :: m_k$ and $C_i :: m_k$ as a pair of conflicting APIs, which is denoted as $CAP\langle C_i' :: m_k, C_i :: m_k\rangle$. Conflicting class versions C_i and C_i' caused by a dependency conflict issue, may introduce a set of conflicting API pairs. We denote the set of conflicting API pairs as $CA\langle C_i, C_i'\rangle$.

Definition 2. (Isomerous conflicting API pair): Suppose that $CAP\langle C_i' :: m_k, C_i :: m_k\rangle$ is a conflicting API pair. If there are implementation differences between $C_i' :: m_k$ and $C_i :: m_k$, we consider $CAP\langle C_i' :: m_k, C_i :: m_k\rangle$ as an isomerous conflicting API pair.

Definition 3. (Original dependency path): For each API $C_i' :: m_k$ included in a shadowed class version and directly or indirectly referenced by a class C_1 in the client project, we define any path $Op = \langle C_1 :: m_1, \cdots, C_{i-1} :: m_{k-1}, C_i' :: m_k\rangle$ as its original dependency path, where $C_1 :: m_1$ represents an entry method in the class C_1 of the client project indirectly referencing the method $C_i' :: m_k$ along Op.

Definition 4. (Actual dependency path): Suppose that $CAP\langle C_i' :: m_k, C_i :: m_k\rangle$ is a conflicting API pair. For each original dependency path Op with respect to API $C_i' :: m_k$, we define $Ap = \langle C_1 :: m_1, \cdots, C_{i-1} :: m_{k-1}, C_i :: m_k\rangle$ as the corresponding actual dependency path, as the build environment enforces the interactions between entry method $C_1 :: m_1$ in the class C_1 of the client project and API $C_i :: m_k$ included in the actually-loaded class C_i along Ap. Note that, Op and Ap share the subpath from entry method $C_1 :: m_1$ to $C_{i-1} :: m_{k-1}$.

Problem: Given a project with a set of conflicting API pairs $CA\langle C_i, C_i'\rangle$, our research problem is how to automatically generate tests to trigger the execution of each isomerous conflicting API pair $CAP\langle C_i' :: m_k, C_i :: m_k\rangle \in CA\langle C_i, C_i'\rangle$ along the original and actual dependency paths, and expose the inconsistent behaviors of client project.

2.3 Challenges

RIDDLE is the state-of-the-art technique that generates tests to detect dependency conflict issues in projects where the loaded library versions fail to cover all the referenced APIs based on their method signatures [7]. However, this technique is not applicable to detecting SC issues, since SC

```
// API in shadowed library apache.httpcomponents:httpcore:4.4.6
public boolean keepAlive(HttpContext context, String param) {
    if (context.getContext().equals(HTTP.CONN_CLOSE)) {
            return false;
    } // Field HTTP.CONN_CLOSE == "Close"
    return true;
}
// API in loaded library apache.httpcomponents:httpcore:4.4.5
public boolean keepAlive(HttpContext context , String param) {
    return true;
}
// Entry method in class HttpHeartbeatSender of the client project
public boolean operate(String parameter) throws Exception {
    DefaultRequestDirector client = new DefaultRequestDirector();
    return client.execute(this.context, parameter);
}
// Method referenced by entry method (defined in apache.httpcomponents:
   httpclient.jar:4.5.3)
public boolean execute(HttpContext context, String param){
    DefaultConnectionReuseStrategy connStrategy
                             = new DefaultConnectionReuseStrategy();
    if (param.equals(HttpHeaders.CONNECTION)){
            return connStrategy.keepAlive(context, param);
    } // Field HttpHeaders.CONNECTION == "Connection"
    return false;
}
```

(a) Issue #1730 in project `Sentinel` v1.8.0

```
// heartbeat.HttpHeartbeatSender.java
public HttpHeartbeatSender(String attribute){
      this.context = new BasicHttpContext(attribute);
}
```

(b) Constructor of the class *heartbeat.HttpHeartbeatSender*

```
// heartbeat.HttpHeartbeatSender.java
public void scheduleHeartbeatTask(){
    HttpHeartbeatSender httpHeartbeatSender
                        = new HttpHeartbeatSender("Close");
    if (httpHeartbeatSender.operate("Connection")) {
       conn.close();
    }
}
```

(c) A caller method of class *heartbeat.HttpHeartbeatSender*'s constructor and API *HttpHeartbeatSender.operate(String)*

```
//@Test
public void test01() throws Throwable {
  HttpHeartbeatSender httpHeartbeatSender0
                              = new HttpHeartbeatSender("#a");
  boolean boolean0 = httpHeartbeatSender0.operate("./1");
     assertTrue(boolean0);
}
boolean0 == true  //On apache.httpcomponents:httpcore:4.4.5
boolean0 == true  //On apache.httpcomponents:httpcore:4.4.6
```

(d) A test case generated by `Riddle` its corresponding results

Fig. 3: An example for illustrating RIDDLE's challenges in detecting SC issues

issues arise from referencing the APIs with identical method signatures but inconsistent behaviors across multiple library versions.

RIDDLE uses genetic algorithm to evolve sets of candidate tests, aiming to maximize the possibility of triggering the conflicting API execution with the defined guidance criteria (see more detailed description of fitness function in Section 4.3). However, a major challenge of test generation for dependency conflict issues lies in the difficulties in reaching the target branches along the path from entry methods defined in a client project to the shadowed APIs provided by third-party libraries. Especially, the conditions on the path may involve complex object creation, field accesses, etc., making the conditions hard to construct. To address this problem, RIDDLE mutates the conditions of each branch in an identified path by making them be evaluated to be `True` or `False` so as to force a test execution along the path that invokes a shadowed API (this mutation operation is called *short-circuiting*). As RIDDLE is designed to trigger the referenced but shadowed APIs due to dependency conflicts, after short-circuiting the unsolvable branch conditions, a crash can be captured by its generated tests.

We may adapt the mechanism to detect SC issues. For example, after identifying the conflicting library APIs for SC issues, we can use RIDDLE to generate tests to drive the program to execute along the path from an entry method to the isomerous conflicting library APIs. Then, we can execute the generated test using the shadowed version and the loaded version, respectively, and compare their test outcomes to check the semantic inconsistency. However, based on our trials on 70 Java projects with RIDDLE, we observed that this approach cannot effectively detect SC issues, due to its randomly generated arguments and short-circuiting operations.

Consider an SC issue #1730 [15] of project `Sentinel` v1.8.0, which is caused by directly and indirectly referencing multiple versions of library `httpcomponents.httpcore` (version v4.4.5 was loaded by build tool, while version v4.4.6 was shadowed). As shown in Figure 3(a), an entry method `HttpHeartbeatSender.operate(String)` of the client project depends on method `DefaultRequestDirector.execute(HttpContext, String)` in library `httpcomponents.httpclient` v4.5.3. Specifically, the above method is originally designed to reference API `ConnectionStrategy.keepAlive(HttpContext, String)` defined in library `httpcomponents.httpcore` v4.4.6. Since version v4.4.6 was shadowed, such entry method of the client project actually transitively invoke the API `ConnectionStrategy.keepAlive(HttpContext, String)` in version v4.4.5, whose implementations were semantically inconsistent with the intended version.

To trigger the conflicting library API `Connection-Strategy.keepAlive(HttpContext)` in the intended version `httpcomponents.httpcore` v4.4.6 or the loaded version `httpcomponents.httpcore` v4.4.5, a test must instantiate an object of the class `HttpHeartbeatSender` with the aid of its constructor as shown in Figure 3(b), and then call the entry method `HttpHeartbeatSender.operate(String)`

TABLE 1: The statistics of the subjects collected in our study (k = 1,000)

# Project	# Star			Size (# kLOC)			# Test case			# Library			Lib_v			Path length		
128	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
	61	22.4k	931	0.7	509.1	91.4	30	4,824	379	20	129	41	6	116	53	3	13	7

provided by the object. Figure 3(d) shows a test generated by RIDDLE. Although the randomly generated argument `"./1"` (for the entry method) does not meet the condition at `Line 88` of method `DefaultRequestDirector.execute(HttpContext,String)`, RIDDLE can force the test execution along the path that invokes the target API `ConnectionStrategy.keepAlive(HttpContext)`, by mutating the condition of this branch to be evaluated to be `True`. However, this mutation violates the originally designed semantic constraints and cannot be considered as a convincing evidence for revealing inconsistent behaviors. In this case, even if RIDDLE triggered the target API, the randomly generated argument `"#a"` (for class `HttpHeartbeatSender`'s constructor) cannot satisfy the semantic constraint of condition at `Line 68` of method `ConnectionStrategy.keepAlive(HttpContext)` in `httpcomponents.httpcore` v4.4.6. As such, the test could not capture the semantic inconsistency between these two conflicting library versions (both returning `True` on versions v4.4.6 and v4.4.5).

As observed from the above example, the SC issue requires a specific argument to be triggered, i.e., valid parameters `"Close"` and `"Connection"`, which can be found in a caller method of `HttpHeartbeatSender`'s constructor and API `HttpHeartbeatSender.operate(String)` (as shown in Figure 3(c)). RIDDLE, which generates test inputs randomly and mutates branch conditions, is ineffective in exposing such issues. Even worse, using the invalid arguments to instantiate class objects can easily trigger runtime exceptions (i.e., `NullPointerException`) before reaching the call sites of target APIs. The example motivates us to conduct an empirical study to understand the characteristics of specific arguments that expose SC issues.

3 EMPIRICAL INVESTIGATION

As discussed above, detecting SC issues typically requires rich domain knowledge about how to generate divergence arguments to capture the subtle differences in API implementations across conflicting library versions. Hence, understanding the characteristics of arguments on the test cases that can effectively expose SC issues, helps design an effective SC issue detection technique. To achieve this goal, we conducted an empirical study on a collection of test cases that have captured real SC issues. The study aims to answer the following two research questions.

RQ1: *Are randomly generated arguments in test cases likely to capture inconsistent program behaviors of SC issues? What are the characteristics of divergence arguments?*

Generating the desirable test inputs with specific semantic meanings or in specific formats is a significant challenge for automated test generation techniques [16]. Specifically, to increase the likelihood of triggering the conflicting API pairs and revealing SC issues, we should be able to generate divergence arguments for the concerned API calls and related object constructors. To ease our presentation, we refer to an API call/object constructor taking no arguments as a *no-args API/constructor*, and an API/object constructor that takes arguments as a *parameterized API/constructor*. In this paper, we focus on the characteristics of the divergence arguments required by parameterized APIs and constructors in the test cases, i.e., the concrete values held by arguments, including strings, primitive types, and object references.

RQ2: *Can divergence arguments in test cases be found in the source code of the client project?*

The above investigation can provide empirical evidence and guidance to help construct divergence arguments for parameterized APIs and constructors in test generation.

3.1 Collection for benchmark dataset

Identifying existing tests written by developers or generated by tools that can detect SC issues is difficult. To achieve such a goal, we first simulate a series of dependency conflicts for a given project by altering the actually-loaded versions of its referenced libraries. We then execute the project's associated tests to see if it can capture the inconsistent behaviors introduced by the version substitution. The steps and criteria for constructing such a dataset are described as follows in detail:

Step 1: Selecting subjects. We randomly selected Java projects from `GitHub` satisfying three conditions: (1) including more than 30 test class files designed by the original developers with their domain knowledge; (2) passing all the associated test cases without errors (ensuring no SC issues in the selected version); (3) depending on more than 20 libraries (having more upgraded/downgraded candidate libraries). As such, we obtained 1221 open-source projects.

Step 2: Altering the actually-loaded library version. For each library on the dependency tree of a subject, we first collected a set of its version numbers released on the `Maven` central repository, which is denoted as $V = \{v_1, v_2, \cdots, v_n\}$. We iteratively used each library version $v_i \in V$ to replace its original version on the dependency tree. Then, we checked whether the associated tests thrown `AssertionErrors` when running on the subject after replacements. The rationale is that the `AssertionError` in `JUnit` tests is used to indicate whether the actual variable values are equal to their expected values [17]. If a test passes for the selected version of a subject and fails for the revised version with an `AssertionError`, we consider that the failing test captures an SC issue caused by the substitution of library version v_i. By manually debugging two versions of the program triggered by this failing test, on the execution traces, we can identify a pair of isomerous conflicting APIs defined in both the original and altered library versions.

Eventually, from 128 Java projects, we obtained 316 SC-revealing test cases, which correspond to 316 conflicting API pairs. Table 1 shows the statistics of the subjects. They

```
// Project: struts-master/core  2.5.22                              (a)
// Test case: DateInterceptorTest.testDateField
DateFormat date = new SimpleDateFormat("yyyy-mm-dd");
```

```
// Project: index-providers 5.4.1                                   (b)
// Test case: EsIndexTest.setUpClass
EsClient client = new EsClient("localhost", 9200);
EsIndex index = new EsIndex(client);
```

```
// Project: kitodo-production 3.0.0                                 (c)
// Test case: MockEntity.prepareNodeSettings
ConfigMain cm = new ConfigMain("NodeSettings");
String host = cm.getParameter("elasticsearch.host", "9205");
HttpHost httpHost = new HttpHost(host);
```

Fig. 4: The illustrative examples for explaining the constructors' arguments (denoted by arguments) in test cases

are large (up to 509.1 kLOC), popular (up to 22.4k stars) and well-maintained (up to 4,824 associated test cases). Moreover, they have large-scale dependency trees (up to 129 referenced libraries), and on average, each library has 53 versions released on the `Maven` central repository (denoted as Lib_v). And more notably, the average length of dependency path from entry methods to the conflicting APIs triggered by the SC-revealing test cases is 7. The statistics indicate that the collected test cases are representative.

3.2 Empirical findings of RQ1

To answer RQ1, we manually checked 1,207 API calls and 2,356 class instances used in 316 collected SC-revealing test cases, to analyze the characteristics of the divergence arguments required by their corresponding constructors. By investigation, we found that in the above test cases, 1,206 out of 1,207 API calls (99.9%) are parameterized ones, and 2,249 out of 2,356 class instances (95.5%) need to be created using parameterized constructors. Among 1,206 parameterized APIs and 2,249 parameterized constructors in the test cases, the number of their required arguments ranges from 1 to 7 (i.e., 3±0.36). For 10,711 divergence arguments needed by the parameterized APIs and constructors, we investigated the corresponding source files to understand their characteristics of assignments. Based on our observations, we divide the arguments into the following three types:

Type 1. *The arguments are strings or primitive type values with specific semantic meanings or in specific formats (30.6%).* 3,276 out of 10,711 arguments are strings or primitive type values (i.e., numeric and enumeration variables) in the test cases. By manually checking their values, we found that 2,547 out of 2,632 string arguments (96.8%) are constrained in specific formats or have specific semantic meanings that reflect developers' domain knowledge, such as, the protocol or date related strings. For instance, in the test file as shown in Figure 4(a), a string `"yyyy-mm-dd"` is assigned to the parameter of constructor `SimpleDateFormat(String)`, which can rarely be generated randomly.

Besides, most of the primitive type arguments are used to specify boundary or specific values, e.g., -1, 9200, etc. For 644 primitive type arguments, we used a random value to replace each argument and then run the corresponding revised test cases on both the original and the altered library versions. The above process is repeated ten times for each argument. Over the ten runs, if the revised test cases can capture SC issues once a time, we consider the corresponding argument can be replaced with random values. Unfortunately, 623 out of 644 arguments (96.7%) failed to trigger the inconsistent behaviors after the random replacements.

Type 2. *The arguments are the instances of other classes created by the constructors with specific inputs (18.5%).* 1,978 out of 10,711 arguments are the instances of other classes. 1,924 out of the above 1,978 object constructors (97.3%) require arguments. In such cases, developers should recursively construct class instances, and the combination of the involved arguments determines the outermost instance's state. Therefore, it requires rich domain knowledge to create such combination of specific arguments. For example, as shown in Figure 4(b), the argument of constructor `EsIndex(EsClient)` in the test case is created by constructor `EsClient(String, int)` with specific arguments. The arguments `"localhost"` and `9200` determine the state of constructed object `client`.

Type 3. *The arguments are returned by the other API calls with specific inputs (5,457/10,711 = 50.9%).* In such cases, the states of arguments are determined by the API calls with valid arguments and the states of class instances providing the above API calls. Figure 4(c) shows an example of this type of assignment in test file `MockEntity.prepareNodeSettings`, the argument of constructor `HttpHost(String)` is returned by API `ConfigMain.getParameter(String, String)` with valid inputs `"elasticsearch.host"` and `"9205"`. In such scenario, developers should also instantiate the classes that provide the required API calls, using specific argument `"NodeSettings"`. Similar to **Type 2** cases, the combination of the involved arguments required by API calls and the recursively constructed class instances, makes the parameter assignments more complicated.

For most of the parameterized APIs and constructors, they require mixed types of arguments. In many cases, when instantiating a class instance or calling an API, to construct one type of arguments, we need to recursively create the other types of arguments. An effective technique for the generation of valid test inputs is essential to capture real program behaviors.

Finding 1: *In the collected SC-revealing test cases, 1,206 out of 1,207 API calls (99.9%) and 2,249 out of 2,356 class instances (95.5%) require at least one arguments to capture the inconsistent behaviors.*

Finding 2: *10,551 out of 10,711 divergence arguments (98.5%) required by the parameterized APIs and constructors are subject to semantic constraints, which can hardly be replaced by random values.*

3.3 Empirical findings of RQ2

To answer RQ2, for 1,206 parameterized APIs and 2,249 parameterized constructors in the collected test cases, we first located their corresponding caller methods in the client projects' source code. Furthermore, we manually checked whether the valid arguments of API calls and constructors could be found in the code snippets of their caller methods. The rational is that the caller methods mostly contain the invocation contexts of a parameterized API/constructor. Note that a parameterized API/constructor may have more than one caller methods in the source code.

For each collected parameterized API/constructor, suppose $Argu_t$ is the total number of the required arguments, and $Argu_s$ is the number of arguments that can be identified in the source code of client project. We found that 2,124 out of 3,455 parameterized APIs and constructors (61.5%) whose corresponding $Argu_s/Argu_t$ values are greater than 0. This means that for 2,124 parameterized APIs and constructors, at least one of their required arguments can be found in the source code. Specially, among the above 2,124 parameterized APIs and constructors, 1,423 of them (67.0%) whose corresponding $Argu_s/Argu_t$ values are equal to 1. For the rest 701 cases whose corresponding $Argu_s/Argu_t$ values are between 0 and 1, we observed that 61.9% of their arguments are passed by the input parameters of the constructors' caller methods. However, the above caller methods are the APIs provided for invocation by the third-party projects. As a result, their valid arguments could not be found in the source code of client projects.

For the remaining 1,331 parameterized APIs and constructors (38.5%) whose arguments cannot be found in the source code (i.e., $Argu_s/Argu_t$ values are equal to 0), we performed the following tasks: (1) We manually extracted the valid arguments from the code snippets of their caller methods. If an argument could not be found in the source code, we randomly assigned a value to it. (2) Let $Argu'_s$ be the number of arguments that are manually extracted from source code. For each API call/class instance created by the original developers in the test cases, we replaced it with our constructed ones whose corresponding $Argu'_s/Argu_t$ values are greater than 0 (i.e., at least one argument could be found in the source code). (3) After the replacement, we executed the revised test cases and checked whether they could still capture the inconsistent behaviors with `AssertionErrors`. Finally, 953 out of 1331 parameterized APIs/constructors (71.6%) were replaced in 193 test cases. For the above 193 revised test sripts, 119 of them (61.7%) successfully detected the SC issues when running on the project versions with upgraded/downgraded libraries. The average value of $Argu'_s/Argu_t$ of the 601 replaced constructors in the 119 test cases that can capture SC issues is 0.25 higher than that of the 352 replaced constructors in the 74 test cases that fail to detect SC issues (0.61 *v.s.* 0.36).

From the above results, we can draw the conclusion that combining the API calls and object constructors with their valid arguments extracted from the source code to generate tests can help to expose the inconsistent behaviors. The more valid arguments injected to the constructors, the higher success rate of capturing SC issues.

Finding 3: *In the collected test cases that can capture the SC issues, 2,124 out of the 3,455 parameterized APIs and constructors (61.5%) of which parts of their arguments can be found in the source code.*

Finding 4: *When we substituted our injected arguments for the constructor arguments that cannot be found in the source code, 119 out of 193 test cases (61.7%) captured SC issues.*

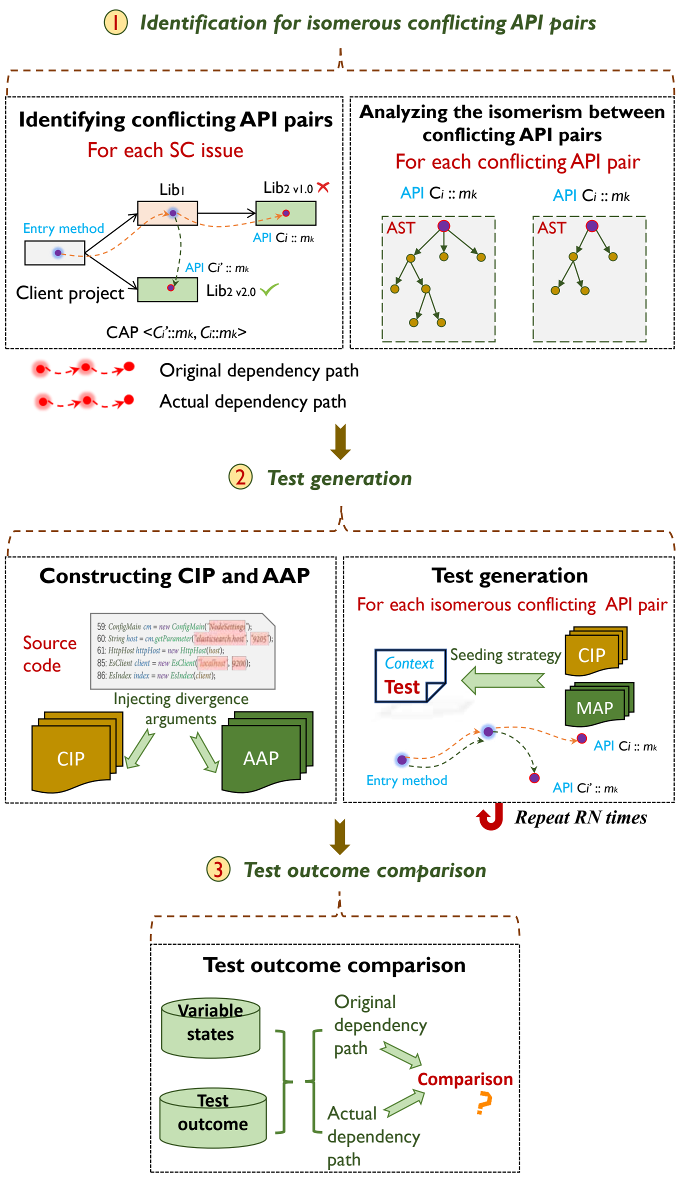


Fig. 5: The overall architecture of SENSOR

The empirical findings of RQ1 and RQ2 shed lights on understanding the characteristics of divergence arguments and provide valuable guidance to design an automated test generation technique for detecting SC issues.

4 THE SENSOR APPROACH

4.1 Overview

Figure 5 shows an overview of our approach, which involves three steps: *identification for isomerous conflicting API pairs*, *test generation* and *test outcome comparison*. First, SENSOR finds a set of conflicting API pairs introduced by a dependency conflict and identifies which of these conflicting API pairs are isomerous. Second, it generates tests to capture the inconsistent variable states of a given client project affected by isomerous conflicting API pairs. Third, by comparing their test outcomes obtained on two conflicting class versions, SENSOR identifies the SC issues and points out their impacts on the client project's program behaviors at a fine-grained level to help diagnose SC issues.

4.2 Identifying isomerous conflicting API pairs

SENSOR identifies the isomerous conflicting API pairs at a fine-grained level based on code differences detected iteratively using GUMTREE [18]. It considers that those conflicting API pairs with different implementations will potentially cause semantic conflicts.

Identifying conflicting API pairs. By analyzing the dependency tree of a client project, SENSOR identifies multiple versions of a class C_i or a library Lib_i. For each API $C'_i :: m_k$ defined in the shadowed class version C'_i and referenced by the client project, SENSOR considers the API $C_i :: m_k$ that satisfies one of the following two conditions as its replaceable method:

- API $C_i :: m_k$ has the same signature (i.e, method name, parameter types and return types) as $C'_i :: m_k$ and is defined in the actually-loaded class version.
- Suppose class CS_i is the superclass of C_i. If the actually-loaded class C_i does not include the API with the same signature as $C'_i :: m_k$, then SENSOR regards the API defined in its superclass $CS_i :: m_k$ that can be overridden by $C'_i :: m_k$ as its replaceable method. In this case, the API compatibility will not be broken due to dynamic binding mechanisms.

Finally, $C'_i :: m_k$ and $C_i :: m_k$ are identified as a pair of conflicting APIs, which is denoted as $CAP\langle C'_i :: m_k, C_i :: m_k\rangle$.

Analyzing isomerous conflicting API pairs. We adopt GUMTREE [18] to check if implementation differences exist in a conflicting API pair. GUMTREE detects code differences based on abstract syntax trees (ASTs). When applying GUMTREE, SENSOR needs to consider the cases where a pair of conflicting APIs exhibit no difference in terms of their ASTs but are semantically different due to the changes in their depended methods. Typically, an API could invoke a series of methods which constitute a call graph. Any changes in the methods invoked by an API could possibly affect the states of its referenced variables, thereby changing the API's semantics. To perform a comprehensive analysis for a conflicting API pair, we construct the corresponding call graphs of the two APIs, and then iteratively compare each method pair with the same signature on the call graphs in a top-down manner. In the process of iterative analysis, we consider the above conflicting API pair as an *isomerous conflicting API pair*, if there are AST differences identified by GUMTREE between one comparable method pair on the call graphs.

Although our iterative analysis can capture all code differences between a pair of method invocation paths on the call graphs, not all the differences are useful in practice. According to an empirical study conducted by Schröter et. al [19], nearly 90% of the issues are fixed within the **top-10** methods along the invocation paths. Other deeper methods barely affect the program semantics of the client project. To reduce such false positives, SENSOR only analyzes the methods whose call depth is less than **ten**, along the original and actual dependency paths of a conflicting API pair.

Note that, SENSOR performs static analysis on Java projects based on SOOT framework [20]. It is well-known that statically constructing sound and complete call graphs and program dependency graphs for Java language is challenging due to the language features such as dynamic binding and reflections [21], [22]. In our approach, SENSOR does not consider reflection-related dependencies, due to static analysis framework's limitations. While it takes dynamic binding-related dependencies into account, such as the methods overridden by the ones in other classes, etc. Especially, SENSOR leverages SOOT's program dependency graph and call graph APIs to identify the isomerous conflicting API pairs with configuration `cg.spark geom-pta:true` [23], which is proposed by Xiao et al. [24] using context sensitive points-to analysis to obtain more precise call graphs.

4.3 Test generation

SENSOR is built on top of EVOSUITE, which adopts a genetic algorithm (GA) to derive a test suite for a given target class. SENSOR considers a class that contains an entry method directly or indirectly referencing the identified isomerous conflicting APIs, as a target class. However, each isomerous conflicting API pair may correspond to a set of target classes defining the above entry methods. In such cases, SENSOR iteratively generates tests for each identified target class.

To precisely capture the program behaviors, SENSOR adopts a seeding strategy of class instances and API calls with divergence arguments, inspired by the our empirical findings summarized in Section 3. SENSOR injects the invocation context information extracted from the source code into class instances and APIs with the aim of generating divergence arguments. To be more specific, for each class and API defined in a client project, SENSOR constructs a pool of class instances (denoted as CIP), and a pool of API calls (denoted as AAP), with the injected invocation contexts. When EVOSUITE needs to instantiate a class or insert an API call into a test, SENSOR tries to select an instance/API from CIP/AAP and provide it to EVOSUITE.

Fitness function. The GA works by iteratively selecting individuals from the population based on their fitness function with respect to the search objective. Note that the individuals of the GA's population are test suites. SENSOR adopts the fitness function defined by RIDDLE, which aims to maximize the possibility of covering the identified target APIs [7]. In general, it is more difficult to generate appropriate input data to trigger the target API's execution when there are more branches along the control flow paths from an entry method to the target API. Hence, to ease the generation of input data, RIDDLE's fitness function estimates the number of branches that pass along the path from entry method to the target API. Then, an individual (i.e., a test suite) with a lower fitness value, is more likely to cover the identified conflicting API. Let CM_t be a set of methods covered by test suite ts_t, and $BN(m_p)$ is the number of branches passed through by the path from method $m_p \in CM_t$ to conflicting API m_k. The fitness of individual ts_t is $\min\{BN(m_p)|m_p \in CM_t\}$. The individuals with lower fitness values tend to be selected during the evolution process. Consequently, an individual has fitness 0 if it covers the identified conflicting API.

Constructing CIP and AAP. SENSOR constructs a class instance involving two steps: identifying its possible object constructors and extracting the constructors' invocation contexts from the source code of the client project. Let

$MOI_i = \{mo_1, mo_2, \cdots, mo_n\}$ be a set of possible object constructors of a given class C_i collected by the static analysis approach [25], where $mo_k \in MOI_i$ represents an object constructor of this class. For most of the cases, a constructor requires parameters for substantiation. To inject valid arguments into $\forall mo_k \in MOI_i$, SENSOR performs the following tasks: (1) it identifies a set of caller methods $M_k = \{m_1, m_2, \cdots, m_n\}$ in the client project, which reference the constructor $mo_k \in MOI_k$; and (2) for each caller method $m_t \in M_k$, SENSOR locates the source file where it is defined, and considers the code snippets within the source file as a search scope of the invocation contexts of mo_k. Specially, SENSOR takes into account the following four cases to extract the invocation contexts of $mo_k \in MOI_i$, based on three types of constructors' arguments observed in our empirical study:

Case 1: If the required arguments of mo_k are strings or primitive types that can be identified in source code by exactly matching assignment statements with variable names, SENSOR extracts their corresponding assigned values directly from the source code.

Case 2: In the case where the arguments of mo_k are the input arguments of the caller method $m_i \in M_k$, SENSOR recursively searches the corresponding invocation context for caller method m_i.

Case 3: For the arguments whose assigned values are returned by the other API calls, SENSOR recursively finds invocation contexts of the required API calls and the class instances providing the API calls, following the above steps.

Case 4: For the required arguments that are the instances of other classes, SENSOR recursively constructs such class instances following the above steps. Moreover, if the required arguments are strings or primitive types whose valid values cannot be exactly extracted from source code, SENSOR randomly assigns values to them. In the cases of recursively constructing class instances or searching for valid arguments from the intermediate invocation contexts (e.g., *Cases 2* and *3*), the searching process is terminated if the recursion depth is greater than DN, or it cannot generate an instance of the current class (e.g., the class is not instantiable for accessibility reasons). In this manner, for each class C_i, we can obtain a set of possible object constructors MOI_i, and each constructor $mo_k \in MOI_i$ corresponds to a set of invocation contexts extracted from the source code.

Similarly, SENSOR constructs a pool of divergence arguments for all the public APIs defined in the client project, by searching their invocation contexts from the source code. For each API, SENSOR first identified its series of caller methods in the client project, and then iteratively searches the required arguments from the code snippets of each caller method. As such, each API also corresponds a set of possible invocation contexts. Specifically, in the cases that the arguments are strings or primitive types, the arguments are the input arguments of the caller methods, and the arguments are returned by other API calls, SENSOR extracts the API's invocation contexts in the same manner of *Cases 1-3* as described in CIP construction. While if the API's required argument is a class instance, SENSOR randomly assigns it one of the instances of this class from CIP. Note that, SENSOR assigns it *null* value if there are no such class instances in CIP.

Seeding strategy. During the insertion of new statements into a test case, EVOSUITE tries to resolve dependencies either by re-using objects declared in earlier statements of the same test, or by recursively inserting new API calls to generate new instances of the required dependency objects [26], [27]. Whenever EVOSUITE attempts to generate a class instance or an API call, SENSOR selects one of the instances of that class from CIP or one set of possible divergence arguments of that API from AAP, with probability P_{OC} to replace it. Note that if CIP/AAP does not contain the required class instance/API's divergence arguments, such replacement operations are not performed.

In the presence of an object constructor/API call and its various corresponding invocation contexts, setting the probability P_{OC} to choose one of them is challenging. In our approach, we set a probability P_{OC} based on the complexity of a constructor's/API's invocation contexts. The selection strategy favors the class instance/API's divergence arguments with a low complexity. Also, it dynamically adjusts the probability P_{OC} according to the number of times that a class instance/API's divergence arguments has been selected during the test generation process, to diversify candidate class instances/API's divergence arguments. Thus, we have

$$P_{OC} = 1/Depth_{arg} \times 1/T_s \tag{1}$$

where $Depth_{arg}$ is an indicator of the invocation contexts' complexity, which represents the recursion depth for constructing the involved class instances or searching valid arguments from the intermediate invocation contexts; and T_s is the number of times that a class instance/API's divergence arguments has been seeded into the tests.

4.4 Test outcome comparison

For each isomerous conflicting API pair, SENSOR generates tests to trigger their executions on the actually-loaded and the shadowed class versions where they are defined, respectively. It repeats the above test generation process for RN times and then compares the test outcomes.

Triggering isomerous conflicting API pairs. For each isomerous conflicting API pair $CAP\langle C_i' :: m_k, C_i :: m_k\rangle$, SENSOR performs the following tasks, where class C_i' is the shadowed version and class C_i is the loaded version.

(1) First, it configures the dependency management file (i.e., `pom.xml`) to force the build tools to load the class version C_i' containing the API $C_i'::m_k$.

(2) Then, it identifies a set of entry methods EM defined in the classes of the client project that directly or indirectly references API $C_i'::m_k$.

(3) For each entry method $C_p :: m_k \in EM$, SENSOR generates tests for the class C_p where method m_k is defined. SENSOR repeats the above test generation process for RN times and records the test outcomes. In our approach, we assume that the original dependency paths from entry method $C_p :: m_k$ to the API $C_i' :: m_k$, will not trigger test failures, unless the test inputs are invalid. Thus, after filtering out all the failing tests, we obtain a set of tests T_k ($|T_k| \leq RN$) for triggering API $C_i'::m_k$.

(4) If $T_k \neq \emptyset$, SENSOR configures the `classpath` to load the class verion C_i containing API $C_i::m_k$. And then it runs each test $t_u \in T_k$ on the program with the class version C_i and records their corresponding test outcomes. In this

manner, SENSOR ensures that a pair of conflicting APIs receive the same inputs during the testing process.

(5) Compare the outcomes of each test $t_u \in T_k$ after running on the above two versions of code.

Outcome comparison. SENSOR takes the following two types of semantic inconsistencies into account, in the comparison process:

- *Variable states.* It considers three types of affected variables, including: (a) each input argument of entry method whose type is an object; (b) each global variable referenced by entry method; and (c) return variable of the entry method. As a result, if the state of any affected variable is different across the executions on the above two versions of code, SENSOR regards the behaviors as inconsistent.
- *Test outcomes.* If a test succeeds to run on one version of code but fails on the other, a semantic inconsistency is noted.

To be more specific, when each test $t_u \in T_k$ separately runs on the two versions of code, we record the state of each affected variable in two cases: (1) if the variable is of string or primitive type, we collect its state based on program instrumentation technique with the aid of ASM[1]; (2) if the variable whose type is an object, we output its state into a stream based on object serialization analysis[2]. Then, we compare their recorded values to check whether this affected variable's state is consistent across two versions of code. SENSOR considers the isomerous conflicting APIs that induce one or more inconsistent variable states or test outcomes when executing one or more generated tests, as the cases that could cause SC issues. Finally, it illustrates their impacts on the program semantics of the client project, to help developers further diagnose the SC issues.

5 EVALUATION

This section presents our experimental results through answering the following research questions:

- **RQ3 (Effectiveness):** How effective is SENSOR in detecting SC issues?
- **RQ4 (Usefulness):** Can SENSOR detect unknown SC issues and provide useful diagnosis information?

5.1 Experimental design

5.1.1 RQ3

To study RQ3, we first collected a high quality ground truth dataset and then applied SENSOR to this dataset to assess its effectiveness in detecting SC issues.

Collection for the ground truth dataset. We consider 316 isomerous conflicting API pairs that can cause `AssertionErrors` when executing 316 tests collected in our empirical study (in Section 3), as the ones introducing SC issues into their client projects. We labeled the isomerous conflicting API pairs that will not cause SC issues in client projects, based on the following steps:

(1) We mined the historical commits of open-source projects on `GitHub` and identified the commits that only upgraded/downgraded a library version in the projects' dependency management scripts (e.g., `pom.xml`). We consider that the above change of a library version does not affect the client project's program behaviors, if its corresponding commit satisfies all the following conditions:

- All the tests triggered by a continuous integration build tool (e.g., `TravisCI` [29]) can pass the revised project version, after this commit is submitted.
- In the case that the client project is still active (code commit records can be found in repository within six months), there are no version changes for this library, in the next 24 months after the above commit being merged. Besides, during the above period, there are no issues or commits whose descriptions and logs mentioning the semantic issues caused by this revised library version. The rational is that a recent study [30] found that bugs are usually repaired within 2 years across different projects since they were introduced into the project.

(2) For an identified semantic-preserving library version change, we labeled the isomerous conflicting API pairs in the original and revised versions of this library, which can be covered by the client project's tests (triggered by the continuous integration build tool) without errors, as the ones that will not introduce SC issues.

Eventually, we collected 150 isomerous conflicting API pairs that will not introduce SC issues into their client projects and 316 isomerous conflicting API pairs definitely causing SC issues.

Comparison. We compared SENSOR with the following three state-of-the-art test generation techniques in terms of their effectiveness in covering the identified isomerous conflicting APIs in the above ground truth dataset:

- RIDDLE [7], which is designed for generating tests to trigger the program execution from an entry method of client project to reach an identified conflicting API that is missing in the actually-loaded library. RIDDLE works based on short-circuiting strategy, and a GA with a designed fitness function (also adopted by SENSOR) aiming to maximize the possibility of covering the identified target APIs.
- EVOSUITE [26] with SENSOR's fitness function. EVOSUITE is chosen as a baseline approach because it uses a GA to derive test suites for classes and integrates static and dynamic constant pools to seed arguments [31]. To be more specific, it seeds primitive types and strings from source code or bytecode, and numerical values and strings observed during program execution, with a certain probability. However, such seeding strategy does not map the collected values into the arguments of API calls and object constructors, as it does not consider their corresponding invocation contexts. In the comparison, we adapted EVOSUITE with SENSOR's fitness function.
- JTEXPERT [25], a search-based approach for object-oriented test data generation. It works based on a proposed class instance generator and a seeding strategy to boost the search. JTEXPERT instantiates a class using different means of instantiations (e.g., static methods returning an instance of a given class, static fields that are instances of a given class, etc.), but does not seed the arguments of class constructors from source code, during test generation process.

1. ASM (https://asm.ow2.io/) is a Java bytecode manipulation and analysis framework.

2. Object serialization analysis is used to write a complete state of an object and of any objects that it references into an output stream [28].

All of the above three baseline approaches derive test suites for a given target class. In the comparison, we consider a class that contains an entry method directly or indirectly referencing the identified isomerous conflicting APIs as their target class. As each isomerous conflicting API pair may correspond to a set of target classes defining the above entry methods, we configure these three baseline approaches to iteratively generate tests for each identified target class. Such a configuration is also adopted by SENSOR. Finally, we consider they detect an SC issue, if their generated tests can trigger the isomerous conflicting API pairs in the ground truth dataset and capture the variable state or test outcome inconsistencies.

Metrics. The outcomes of a test generation technique can be categorized as follows: (1) *True Positive* (TP): The inconsistent behavior identified by a technique between a conflicting API pair is a real SC issue. (2) *False Positive* (FP): The inconsistent behavior identified by a technique between a conflicting API pair is not a real SC issue. (3) *True Negative* (TN): No inconsistent behavior is identified by a technique between a conflicting API pair, and it is not a real SC issue. (4) *False Negative* (FN): No inconsistent behavior is identified by a technique between a conflicting API pair, but it is a real SC issue. Based on the outcomes, we use *Recall*, *Precision*, and *F-measure* to evaluate the technique's performance, which are defined as follows.

$$Precision = TP/(TP + FP) \quad (2)$$

$$Recall = TP/(TP + FN) \quad (3)$$

$$F\text{-}measure = 2 \times Precision \times Recall/(Precision + Recall) \quad (4)$$

Precision evaluates whether a technique can detect SC issues precisely. *Recall* evaluates the capability of a technique in detecting all the SC issues. *F-measure* takes the *Precision* and *Recall* into consideration, and weights these two metrics equally [32].

Experimental setting. For SENSOR and three baseline approaches, we set the time budget for the evolutionary search to 800 seconds and repeated the test process for RN = 10 times on each code version with different random seeds.

5.1.2 RQ4

To answer RQ4, we conducted experiments on 140 open-source Java projects that were randomly sampled from `GitHub` using three criteria: (1) it has received more than 50 stars or forks (i.e., popularity); (2) it references multiple versions of libraries or classes detected by static analysis and contains at least one commit after April 2020 (i.e., actively-maintained); (3) it is not included in the subject set of our empirical study in Section 3 (i.e., new validation). We leveraged SENSOR to generate issue reports that include: (a) the root causes of SC issues (i.e., conflicting library versions introduced in a project and the ones would be loaded and shadowed, respectively, and the isomerous conflicting API pairs inducing semantic inconsistencies); (b) the original and actual dependency paths that invoke the isomerous conflicting API pairs; (c) the generated test cases that can trigger the executions of the isomerous conflicting API pairs; (d) the differences in the test outcomes or variable states of the client project after executing the generated test cases. We submitted the issue reports to the corresponding developers via the projects' issue tracking systems and evaluated the usefulness of SENSOR based on developers' feedback.

TABLE 2: The experimental results on the ground truth dataset

	TP	FP	TN	FN	$Precision$	$Recall$	$F\text{-}measure$
SENSOR	229	26	124	87	0.898	0.725	0.802
RIDDLE	21	3	147	295	0.875	0.067	0.124
EVOSUITE	27	4	146	289	0.871	0.085	0.156
JTEXPERT	13	2	148	303	0.867	0.041	0.079

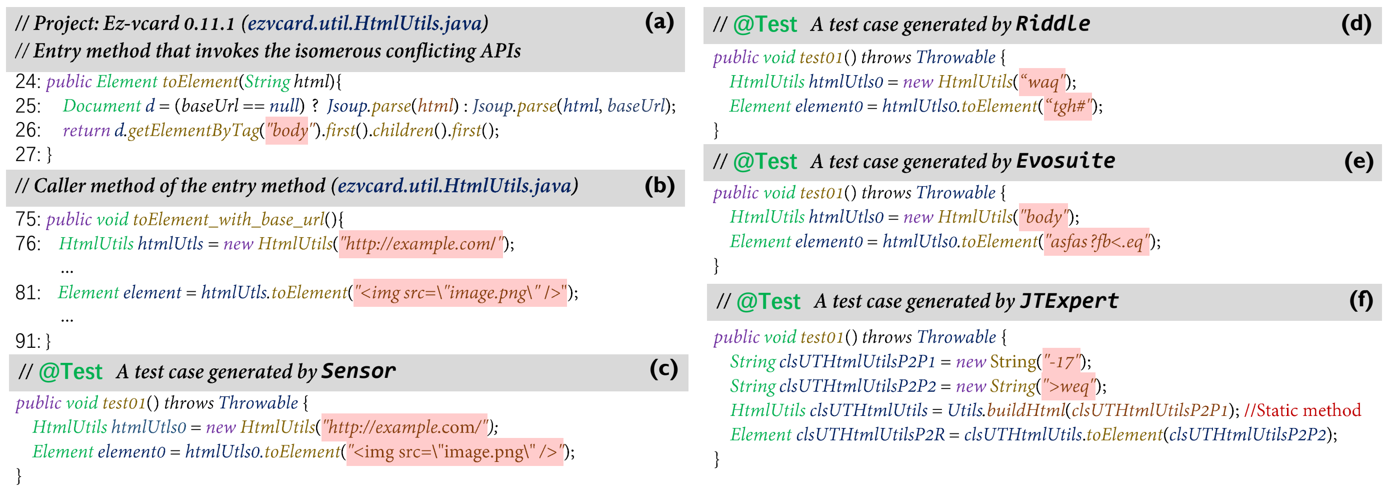
5.2 RQ3: Effectiveness of Sensor

5.2.1 Overall effectiveness.

Table 2 shows the experimental results on the ground truth dataset. RIDDLE identified 24 SC issues with 3 (11.1%) false positives. Besides, it did not capture any inconsistent behaviors between 442 isomerous conflicting API pairs, with 295 (66.7%) false negatives. For EVOSUITE (with SENSOR's fitness function) and JTEXPERT, they identified 31 (with 4 false positives) and 15 (with 2 false positives) SC issues, respectively, and their corresponding $Recall$ are 0.085 and 0.041, respectively. Our approach, SENSOR, identified 255 SC issues with 26 (10.2%) false positives, which achieves a $Precision$ of 0.898. For the isomerous conflicting API pairs that will not cause SC issues in client projects, SENSOR successfully identified 211 of them, with 87 (41.2%) false negatives, leading to a $Recall$ of 0.725. In terms of the $F\text{-}measure$, SENSOR also significantly outperformed all the baseline approaches.

By manually checking the true positive SC issues detected by RIDDLE, EVOSUITE and JTEXPERT, we found that in these cases, all the invocation depths from the entry methods of client projects to the conflicting APIs are less than 3. All the object constructors in the test cases generated by the above three techniques that successfully captured the SC issues, do not need arguments. However, the initialization of the required variables could be found in the method bodies of the above specific simple constructors. Therefore, they could easily trigger the target branches with valid program semantics. Note that the above true positive SC issues were also detected by our technique. For 229 true positive cases detected by SENSOR, the average invocation depth from the entry methods to the conflicting APIs is 7.4. The seeded arguments by SENSOR greatly increases the possibility of reaching the target branches and expose inconsistent behaviors, compared with RIDDLE, EVOSUITE and JTEXPERT.

To be more specific, 197 SC cases were only exposed by SENSOR, while failed to be captured by the three baseline approaches. It is largely ascribed to the effectiveness of SENSOR's seeding strategy, which considers both arguments of class instances and APIs with invocation contexts. For example, as shown in Figure 6 (a), to expose the SC issue in project `Ez-vcard` 0.11.1, a test should first instantiate an object of class `ezvcard.util.HtmlUtils` and then call the entry method `toElement(String)` that indirectly invoke isomerous conflicting APIs. Figure 6 (c) shows a test case generated by SENSOR, which exactly seeded the arguments required by object constructor `HtmlUtils(String)` and API `toElement(String)` from their caller method body (Figure 6



```
// Project: Ez-vcard 0.11.1 (ezvcard.util.HtmlUtils.java)                          (a)
// Entry method that invokes the isomerous conflicting APIs
public Element toElement(String html){
    Document d = (baseUrl == null) ? Jsoup.parse(html) : Jsoup.parse(html, baseUrl);
    return d.getElementByTag("body").first().children().first();
}
// Caller method of the entry method (ezvcard.util.HtmlUtils.java)                  (b)
public void toElement_with_base_url(){
    HtmlUtils htmlUtls = new HtmlUtils("http://example.com/");
    ...
    Element element = htmlUtls.toElement("<img src=\"image.png\" />");
    ...
}
// @Test  A test case generated by Sensor                                           (c)
public void test01() throws Throwable {
    HtmlUtils htmlUtls0 = new HtmlUtils("http://example.com/");
    Element element0 = htmlUtls0.toElement("<img src=\"image.png\" />");
}
```

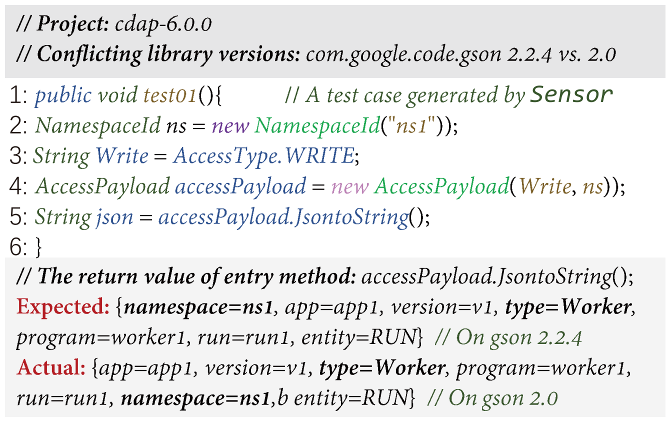
```
// @Test  A test case generated by Riddle                                           (d)
public void test01() throws Throwable {
    HtmlUtils htmlUtls0 = new HtmlUtils("waq");
    Element element0 = htmlUtls0.toElement("tgh#");
}
// @Test  A test case generated by Evosuite                                         (e)
public void test01() throws Throwable {
    HtmlUtils htmlUtls0 = new HtmlUtils("body");
    Element element0 = htmlUtls0.toElement("asfas?fb<.eq");
}
// @Test  A test case generated by JTExpert                                         (f)
public void test01() throws Throwable {
    String clsUTHtmlUtilsP2P1 = new String("-17");
    String clsUTHtmlUtilsP2P2 = new String(">weq");
    HtmlUtils clsUTHtmlUtils = Utils.buildHtml(clsUTHtmlUtilsP2P1); //Static method
    Element clsUTHtmlUtilsP2R = clsUTHtmlUtils.toElement(clsUTHtmlUtilsP2P2);
}
```

Fig. 6: Illustrating the effectiveness of SENSOR's seeding strategy

(b)), and then successfully captured inconsistent behaviors on conflicting API versions. However, the test generated by RIDDLE (Figure 6 (d)) with random arguments failed to reach the target APIs. Although EVOSUITE can seeds primitive types and strings from source code or bytecode (static constant pool), and numerical values and strings observed during program execution (dynamic constant pool) with a certain probability, it does not consider the mappings between collected values and the arguments of APIs and object constructors. As shown in Figure 6 (e), it assigned argument `"body"` of method `Document.getElementByTag(String)` to object constructor `HtmlUtils(String)`, and generated a random argument `"asfas?fb<.eq"` for entry method `toElement(String)`. Consequently, it failed to expose such SC issue. While JTEXPERT used an alternative strategy to seed the object of class `ezvcard.util.HtmlUtils` (using a static method that returns this class instance), but without seeding its required arguments from source code. Then, the test as shown in Figure 6 (f) has less capability in revealing inconsistent behaviors.

We further investigated the reasons why SENSOR generated false positive and negative cases of SC issues and summarized them below.

False positive examples. The main cause of false positives generated by SENSOR is that the inconsistent behaviors are caused by the non-deterministic or random variable states, which are benign for the client projects. For example, as shown in Figure 7, a test case generated by SENSOR captured the inconsistent return values of the entry method in project `cdap 6.0.0`, on conflicting versions of library `com.google.code.gson`. The return value is a `Json` string, which has the same attributes but different declaration orders on these two library versions. However, the attributes are stored in an unordered collection and the sequence of traversing the attributes is non-deterministic in the program. Such differences do not affect the semantics of client project, and therefore it did not catch developers' attention. The other false positives detected by both SENSOR and the three baseline approaches are similar cases, in which the inconsistent behaviors affected by conflicting API pairs (e.g, non-deterministic text formats, random values, etc.), are benign for the program semantics.



```
// Project: cdap-6.0.0
// Conflicting library versions: com.google.code.gson 2.2.4 vs. 2.0
public void test01(){          // A test case generated by Sensor
NamespaceId ns = new NamespaceId("ns1"));
String Write = AccessType.WRITE;
AccessPayload accessPayload = new AccessPayload(Write, ns));
String json = accessPayload.JsontoString();
}
// The return value of entry method: accessPayload.JsontoString();
Expected: {namespace=ns1, app=app1, version=v1, type=Worker,
program=worker1, run=run1, entity=RUN}  // On gson 2.2.4
Actual: {app=app1, version=v1, type=Worker, program=worker1,
run=run1, namespace=ns1,b entity=RUN}  // On gson 2.0
```

Fig. 7: A false positive example of SENSOR

False negative examples. We manually investigated the 87 false negative cases detected by SENSOR, and divided them into the following two categories:

- *The inconsistent branches within conflicting API pairs cannot be reached* (53/87 = 60.9%). In these cases, the required object constructors and APIs have multiple caller methods with different invocation contexts in the source code, and the ones seeded by `Sensor` could not trigger the inconsistent branches within conflicting API pairs, even over the ten runs.
- *The program crashed before reaching the conflicting API pairs* (34/87 = 39.1%). The 34 false negative cases were caused by the inaccurate arguments extracted by SENSOR, which led to program crashes before triggering the conflicting API pairs. By further investigation, we found that the inaccurate arguments are manifested into two patterns: (1) *the required arguments are affected by a series of method calls that cannot be extracted from the source code exactly* (20/34 = 58.9%), and (2) *part of a constructor's arguments cannot be found from the source code, to which random values are assigned* (14/34 = 41.2%).

5.2.2 Effectiveness on producing divergence arguments.

Let N_c be the number of classes in a project for which SENSOR could construct instances; N_a be the number of public APIs for which SENSOR could extract divergence arguments; NC and NA be the total number of classes and public APIs in this project, respectively; NI be the average number of instances with different divergence arguments constructed for each class in a project; and NP be the average number of invocation contexts with different divergence arguments extracted for each API call. Then, N_c/NC and N_a/NA indicate SENSOR's capability on constructing CIP and AAP. In our ground truth dataset, the 466 isomerous conflicting API pairs are selected from 220 Java projects. As shown in Figures 8(a) and 8(b), the box plots show the distribution of indicators N_c/NC and NI in these projects. On average, SENSOR could construct instances with extracted arguments for 75.8% of classes in the projects. We looked into the code and found that SENSOR could not instantiate the remaining classes mainly because their required arguments are not provided in the source code, or for the accessibility reason. On average, in these projects, SENSOR constructed 3.21 instances for each class with divergence arguments.

From Figures 8(c) and 8(d) (distribution of indicators N_a/NA and NP), we can tell that, on average, SENSOR could extract divergence arguments for 68.3% public APIs in these projects. And for each API call, on average, SENSOR found 3.6 invocation contexts from source code with different divergence arguments. By investigation, the main reasons why the APIs' required arguments cannot be extracted from source code are: they are affected by a series of method calls that cannot be exactly identified, and the APIs are only provided for invoking by third-party projects, as such their invocation contexts cannot be found in the source code. Specially, in project `FluentLenium-3.9.0`, there are 89 object constructors and 152 APIs having more than ten invocation contexts from which SENSOR can extract divergence arguments. The diverse constructed class instances and API calls significantly increase the probability of capturing inconsistent behaviors with different invocation contexts.

Let S_c represent the number of class instances that are successfully seeded by SENSOR from CIP into a test case; S_a denote the number of API calls with arguments that are successfully seeded by SENSOR from AAP into a test case; and NO and ND be the total number of instantiated classes and inserted API calls in this test case, respectively. Then, S_c/NO and S_a/ND are the seeding rates of class instances and API calls by our approach in a test case, respectively. Figures 8(e)-(h) show the value distributions of S_c/NO and S_a/ND in the 1307 generated test cases that successfully captured the SC issues and the 2124 test cases that did not reach the target APIs (including the ones triggering crashes), over the RN = 10 runs, respectively. As we can see, the average value of S_c/NO in Figure 8(e) is 0.23 higher than that in Figure 8(f), and the average value of S_a/ND in Figure 8(g) is 0.17 higher than that in Figure 8(h). The results demonstrated the validity of our seeded divergence arguments.

Furthermore, suppose that $Argu_s$ is the number of arguments required by a constructor/API that can be extracted from the source code, and $Argu_t$ is the total number of required arguments in this constructor/API. For the seeded class instances and API calls in the 1307 test cases that successfully exposed SC issues, the average value of $Argu_s/Argu_t$ is 0.25 higher than that in the 2124 test cases failed to trigger the target APIs. This validates the correctness of the arguments extracted by SENSOR for constructing class instances and API calls.

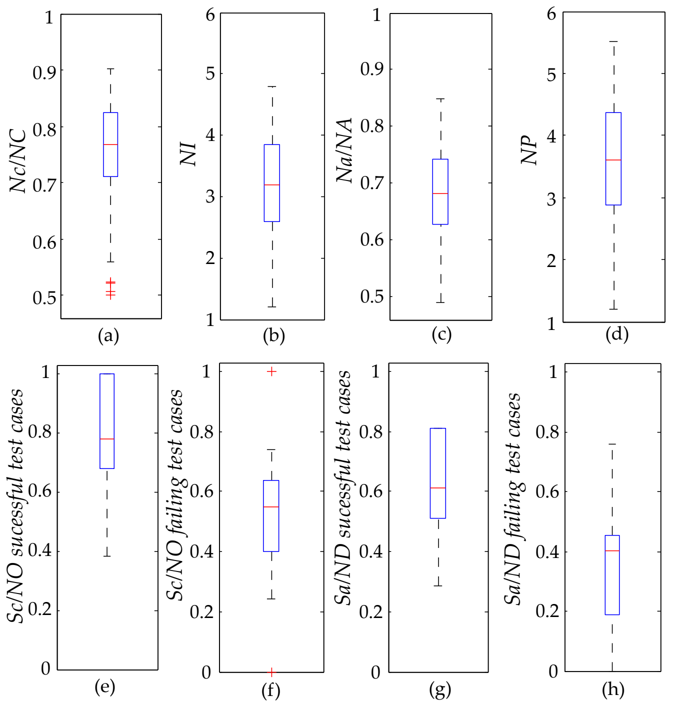


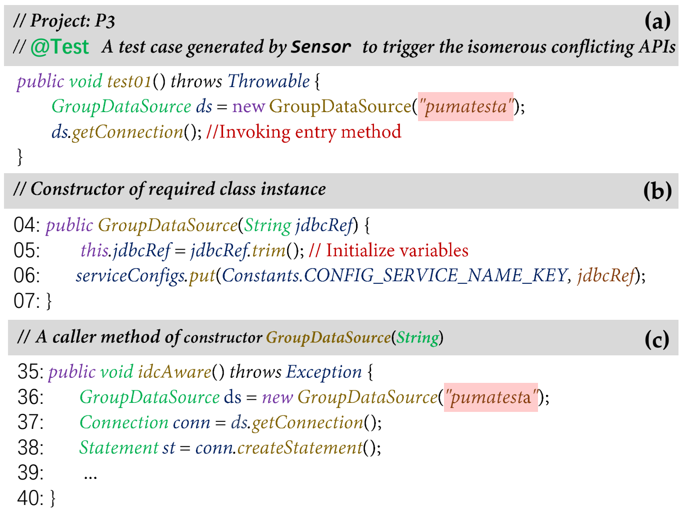
Fig. 8: Effectiveness on producing divergence arguments

TABLE 3: The results on anonymized industry projects

Projects	LOC	CA	ICA	TP	FP
P1	>100K	103	44	7	2
P2	>100K	78	9	3	0
P3	>100K	213	36	10	2
P4	>100K	52	20	5	2
		89	25	6	1
P5	>50K	31	7	2	1
P6	>50K	64	12	3	0
		45	11	3	1
P7	>20K	23	7	2	0
P8	>20K	42	8	3	0
P9	>20K	15	4	1	1
P10	>20K	7	2	1	0
$Precision$	46 / 56 = 0.821				

5.2.3 Effectiveness on industrial projects.

To further assess SENSOR's effectiveness, we applied it to the industrial projects in the `Neusoft` Co. Ltd (SSE: 600718) and received an assessment report [33]. `Neusoft` [11] is the largest IT solutions & services provider in China, which has considerable large-scale Java projects with hundreds of third-party libraries. Diagnosing SC issues is one of the key challenges for their developers. Table 3 reports the results of applying SENSOR to ten industrial subjects that comprise over 0.58 million lines of code. We invited nine developers who participated in development of the selected projects to verify the detected SC issues. We did not evaluate the $Recall$



```
// Project: P3                                                                    (a)
// @Test  A test case generated by Sensor  to trigger the isomerous conflicting APIs
public void test01() throws Throwable {
    GroupDataSource ds = new GroupDataSource("pumatesta");
    ds.getConnection(); //Invoking entry method
}
// Constructor of required class instance                                        (b)
public GroupDataSource(String jdbcRef) {
    this.jdbcRef = jdbcRef.trim(); // Initialize variables
    serviceConfigs.put(Constants.CONFIG_SERVICE_NAME_KEY, jdbcRef);
}
// A caller method of constructor GroupDataSource(String)                       (c)
public void idcAware() throws Exception {
    GroupDataSource ds = new GroupDataSource("pumatesta");
    Connection conn = ds.getConnection();
    Statement st = conn.createStatement();
    ...
}
```

Fig. 9: An example failing test generated by SENSOR for industrial project P3

in this experiment since it is difficult to obtain the complete set of SC issues in the projects.

In Table 3, column *"LOC"* represents lines of source code of the project; columns *"CA"* and *"ICA"* represent the number of conflicting API pairs and isomerous conflicting API pairs caused by two conflicting library versions in the project, respectively. Among the 56 detected SC issues, 46 were confirmed by developers as true positives (TPs) and 10 were labeled as false positives, leading to a *Precision* of 0.821. By communicating with the projects' developers, we found that the main cause of the above false positives (FPs) is the same as that in the open-source projects. In these cases, the inconsistent variable states affected by the conflicting API pairs are benign for the semantics of industrial subjects. Specially, by looking into the test cases that failed to trigger the target APIs, we found that many of them have already been seeded with divergence arguments by SENSOR but still could not work since they need additional environmental dependencies. As shown in an example test generated for detecting an SC issue in project P3 (Figure 9), the called entry method is implemented based on the Java Database Connectivity (JDBC) [34] technique to access virtual data source. Although SENSOR extracted object constructor `GroupDataSource(String)`'s arguments `"pumatesta"` from source code, this test triggered a `java.sql.SQLException` when invoking `GroupDataSource.getConnection()`, due to lack of an environmental configuration of JDBC database.

In particular, we received positive feedback from the `Neusoft`'s testing team, on the high precision of SENSOR. Such results indicate that SENSOR not only achieves significant effectiveness on open-source projects, but also performs great on industrial subjects.

5.3 RQ4: Usefulness of Sensor

SENSOR successfully detected 306 SC issues from 50 projects among all the 140 projects. Note that the SC issues caused by a pair of conflicting library versions were merged into one issue report. Altogether, we submitted 54 issue reports. As shown in Table 4, 38 out of 54 issue reports (70.4%) were confirmed by developers as real bugs; 32 out of 38 confirmed reports (84.2%) were quickly fixed; and 6 of them (15.8%) were in the process of being fixed. Among the 16 unconfirmed issue reports, two were labeled as false positives and the others are not confirmed mainly due to inactive maintenance of the corresponding release versions.

The reasons of the false positive cases, are similar as those we previously discussed. The developers considered the reported issues as benign after checking the impacts of our reported inconsistent behaviors across two versions. For example in `Issue #1897` [35], the isomerous conflicting API pair `<hashCode():Junit.4.4, hashCode():Junit.4.12>` indirectly affects the state of an `HashMap` variable used in an entry method of project `Aws-sdk-java`. APIs `hashCode():Junit.4.4` and `hashCode():Junit.4.12` differ in the way of constructing the value `Key`. However, both versions of this API can ensure the uniqueness of `Key` used to map its associated `Value`. Therefore, the developer considered it as a false positive issue, since it would not introduce bugs.

Encouragingly, we have received developers' positive feedbacks on the reported SC issues and our tool. Developers in `Issue #11` [36] agreed that the provided test case indeed triggered realistic program behaviors, which has facilitated their diagnosis for semantic conflicts. In particular, a developer confirmed the usefulness of our approach:

"I encountered the same problem when using `MiA`. I just noticed something strange happened in the [min, max] range of operation progresses. Thanks for your test case. It helped me reproduce this issue. By amazing coincidence, I got the similar outputs as your test."

Besides, developers have expressed great interests in our detection technique for SC issues. For instance, in `Issue #288` [37], an experienced developer [38] in the `Ontop` community has been looking for a technique of such kind to detect semantic conflicts:

" I am very interested in the detection method so that in the future we will have a more systematic approach to avoid the issue related to such conflict, which is in general quite subtle and difficult to debug."

SENSOR was highly recognized in the assessment report [33] provided by `Neusoft`:

"On average, it took about 20.5 hours to obtain the diagnosis report for a large-scale `Neusoft` project, and the run time depends on the number of conflicting APIs in the project. Although the testing task is time-consuming, SENSOR did a great job in automatically detecting SC issues. The generated diagnosis reports indeed helped us identify many issues that could hardly be found using our existing test suites."

The above results and developers' feedback demonstrate that the information (e.g., test cases) provided by SENSOR is useful for developers to diagnose the SC issues in practice.

5.4 Discussions

5.4.1 What types of behavioral differences can SC issues cause?

We further analyzed the root causes and distributions of the behavioral inconsistencies induced by the identified isomerous conflicting API pairs of all the subjects as shown in Table 4. Statistically, for the 306 isomerous conflicting API pairs that affect the client projects' program behaviors, we further categorized the exposed behavioral inconsistencies

TABLE 4: The SC issues reported by SENSOR

ID	Project	Version	Star	Fork	Issue ID	#SC issues	Status
1	Htm.java	c874f02	296	156	#550	1	Pending
2	EasyTransaction	cadfc96	2k	779	#144	1	Pending
3	Apache/Reef	8eb98d9	92	102	#2056	1	Fixed
4	Hydra	2c9039f	83	47	#364	1	Confirmed
5	Motan	522f450	5.4k	1.8k	#800	15	Fixed
6					#809	15	Fixed
7	Restx	5639370	434	78	#297	3	Fixed
8	Netty-rest	fa9bc00	103	42	#8	12	Fixed
9					#9	9	Fixed
10	Ff4j	8e57626	832	216	#336	10	Fixed
11	Aws-sdk-java	71050e8	3.4k	2.6k	#1897	1	False positive
12	Retrofit	7128bf3	34.8k	6.7k	#3018	12	Fixed
13	Guagua	91c849a	72	40	#103	19	Fixed
14	Jss7	47a32ba	104	162	#309	1	Pending
15	Product-iots	08219aa	120	176	#1911	1	Pending
16	Atom-hopper	86e8872	58	48	#301	1	Fixed
17	Quick-media	6b80d86	441	240	#41	1	Fixed
18	Ontop	2dee196	325	122	#287	12	Fixed
19					#288	17	Fixed
20	Odo	ce1299b	141	28	#173	1	Fixed
21	Java-design-patterns	f006782	55.2k	19.7k	#868	1	Fixed
22	Hmily	da05d51	2.7k	1.1k	#86	1	Fixed
23	Ninja	dc66bf2	1.9k	517	#654	1	Fixed
24	Openstack-java-sdk	426c193	177	203	#214	1	Fixed
25	Javacpp	123ce0d	3k	457	#295	1	False positive
26	MiA	7ff43cc	342	222	#11	1	Confirmed
27	Vertx-examples	dbf0519	2.6k	1.9k	#335	1	Confirmed
28					#336	1	Confirmed
29	Rest-assured	bbede9f	4.5k	1.4k	#1143	1	Fixed
30	Yawp	a415710	137	22	#121	1	Fixed
31	Apache/Hive	fdc2c5c	3k	3.3k	#21374	1	Fixed
32	Weixin-java-springmvc	a900254	333	223	#17	3	Fixed
33	Nutzboot	2addfed	342	118	#199	2	Fixed
34	FastjsonExploit	94d3a38	336	107	#6	1	Fixed
35	Metrics-cloudwatch	42000ff	23	52	#4	1	Pending
36	Sentinel	9936b4d	13.7k	4.8k	#1730	1	Fixed
37	SmartIM	b49de00	131	40	#12	11	Fixed
38	KafkaExample	7e4c35d	116	82	#2	5	Pending
39	Querydsl	58174db	3k	662	#2647	18	Fixed
40	DataLink	dd8d26a	655	287	#52	10	Pending
41	Confluent-Kafka-Certification	9971b93	72	54	#5	4	Pending
42	Yql-plus	0ba7dcf	32	52	#119	29	Fixed
43	Jprotobuf	ed10550	633	228	#141	4	Fixed
44	Freedomotic	1b1a94e	339	484	#490	1	Fixed
45	Reactive-grpc	ce3b3b2	537	78	#226	4	Fixed
46	Hangout	7c21e9c	461	190	#163	5	Fixed
47	Xenqtt	8a37a34	23	58	#4	9	Pending
48	Distributed-redis-tool	17344e4	565	297	#24	21	Pending
49	Webmagic	4b90227	9.3k	3.9k	#951	10	Pending
50	Kafka-avro-course	c19c767	102	112	#8	4	Pending
51	Netbout	65b989e	37	53	#1159	1	Confirmed
52	Qconfig	16d584f	181	61	#17	5	Pending
53	Database	64bb8ca	518	117	#180	11	Confirmed
54	Halo-dal	bf4b588	84	55	#8	1	Pending

The detailed information is available at: https://sensordc.github.io/

// Adding new control branches (Issue #2056 in project Apache/Reef) **(a)**

```
//Loaded version
if (ANNOTATIONS && anns != 0) {
    //Statements
}
if (ANNOTATIONS && ianns != 0) {
    //Statements
}
```

```
//Shadowed version
if (ANNOTATIONS && anns != 0) {
    //Statements
}
if (ANNOTATIONS && ianns != 0) {
    //Statements
}
if (ANNOTATIONS && tanns != 0) {
    //Statements
}
```

// Function implementation inconsisitency (Issue #1143 in project Rest-assured) **(b)**

```
//Loaded version
public Object includeFilterInstance(SerializationConfig config, BeanPropertyDefinition
                                    forProperty, Class<?> filterClass) {
  return null;
}
//Shadowed version
public Object includeFilterInstance(SerializationConfig config,
                                    BeanPropertyDefinition forProperty, Class<?> filterClass) {
  return (VirtualBeanPropertyWriter) this.beanFactory.CreateBean(implClass);
}
```

// Strengthening precondition of a method (Issue #9 in project Netty-rest) **(c)**

```
//Loaded version
public void setNamesSize(HashSet<String> names){ //names cannot be null
this.size = names.size;
}
//Shadowed version
public void setNamesSize(HashSet<String> names){ //names can be null
if (names != null)
this.size = names.size;
else
this.size = 0;
}
//Code snippet of entry method in client project
A.setNamesSize(entryNames);
```

// Weakening postcondition of a method (Issue #809 in project Motan) **(d)**

```
//Loaded version
public PropertyNamingStrategy namingStrategyInstance(MapperConfig<?> config, Class<?>
                                                     implClass){
  return null;
}
//Shadowed version
public PropertyNamingStrategy namingStrategyInstance(MapperConfig<?> config, Class<?>
                                                     implClass){
  return (PropertyNamingStrategy) this.beanFactory.createBean(implClass);
}
//Code snippet of entry method in host project
propertyNamingStrategy propertyNamingStrategy = A.namingStrategyInstance(entryClass);
propertyNamingStrategy.toString();
```

Fig. 10: Examples of behavioral differences

into three types: (a) 268 of them (87.6%) only cause *variable state inconsistencies*; (b) 9 of them (2.9%) only lead to *test outcome inconsistencies*; (c) 29 of them (9.5%) result in both *variable state* and *test outcome* inconsistencies.

By manually examining the source code of isomerous conflicting API pairs, we found that *variable state inconsistencies* are mainly caused by adding or deleting control branches in one version of the conflicting API (e.g., issue #2056 [39] in project `Apache/Reef` as shown in Figure 10(a)) or inconsistent function implementations between conflicting API pairs (e.g., issue #1143 [40] in project `Rest-assured`as shown in Figure 10(b)). In addition, strengthening the precondition or weakening postcondition of a referenced method will lead to *test outcome inconsistencies*. A method's precondition is the condition that a caller must satisfy before calling the method, and a method's postcondition is the condition that a callee must satisfy before returning from the method [2]. For instance, in issue #9 [41] of project `Netty-rest` (Figure 10(c)), replacing the shadowed version of a method with the loaded version resulted in strengthening the precondition of this conflicting method. Therefore, the caller in client project will trigger a `NullPointerException` when the caller of method `setNamesSize()` in the client project passes an empty `HashSet` to it. Similarly, in issue #809 [42] of project `Motan` (Figure 10(d)), referencing the actually loaded version of a method to substitute the shadowed version will weaken its postcondition, which can trigger a crash in the caller. The above cases will break the compatibility of libraries in the client projects.

5.4.2 How do developers fix the SC issues?

The SC issues merged in an issue report are caused by a pair of conflicting libraries, as such they can be solved by a fixing patch. By looking into the 32 fixed issue reports listed in Table 4, we identified the following four fixing solutions, and the mappings between issue reports and such solutions are described in Figure 11:

Solution 1.	*Restx, #297; Netty-rest, #8; Netty-rest, #9; Product-iots, #1911; Atom-hopper, #301; Hmily, #86; Openstack-java-sdk, #214; Rest-assured, #1143; Apache/Hive, #21374; SmartIM, #12; Yql-plus, #119; Hangout, #163; Querydsl, #2647*
Solution 2.	*Motan, #809; Quick-media, #41; Ontop, #287; Ontop, #288; Odo, #173; Java-design-patterns, #868; Weixin-java-springmvc, #17; Nutzboot, #199; FastjsonExploit, #6*
Solution 3.	*Retrofit, #3018; Ninja, #654; Yawp, #121; Sentinel, #1730; Jprotobuf, #141; Freedomotic, #490; Reactive-grpc, #226*
Solution 4.	*Apache/Reef, #2056; Motan, #800; Aws-sdk-java, #1897*

Fig. 11: Mappings between issue reports and fixing solutions

- ***Solution 1.*** *Harmonizing the library version.* 40.6% (13/32) of reported issues are solved by upgrading or downgrading the library versions, since such conflicting libraries are direct dependencies and their higher or lower versions can provide client projects with their expected program behaviors (e.g., issue #297 [43] in project `Restx`).
- ***Solution 2.*** *Adjusting the libraries' declaration order on the `classpath`.* Libraries' declaration order in the dependency management script (i.e., `pom.xml`) determines their loading priorities on the `classpath` (i.e., Maven's *first declaration wins loading strategy*) [5]. 28.2% (9/32) of reported issues are worked around by reversing declaration order of conflicting library versions in `pom.xml`.
- ***Solution 3.*** *Keeping one library version and deleting the other conflicting ones.* 21.9% (7/32) of reported issues are fixed by deleting the unexpected library versions from dependency management script (e.g., issue #3018 [43] in project `Retrofit`).
- ***Solution 4.*** *Declaring a library version as direct dependency to shadow the other conflicting ones.* According to Maven's *nearest wins loading strategy*, the version that appears at the

nearest to the root (client project) of the dependency tree will be loaded, if multiple versions of the same library are referenced by a project [5]. 9.4% (3/32) of reported issues are resolved by declaring the expected library version as a direct dependency to shadow the indirectly introduced conflicting versions (e.g., issue #2056 [43] in project `Apache/Reef`).

The above four solutions can be applied to the scenario that adopting a higher or lower library version can resolve the SC issues, as the 32 issue reports satisfy **(adopting lower version: 3/32 *v.s.* adopting higher version: 29/32)**. Otherwise, according to the empirical findings in Wang et al.'s work [5], all the conflicting versions should be loaded by customizing classloader via dynamic module system framework such as `OSGI` [44], or using `Maven-Shade-Plugin` [45] to rename and relocate the conflicting classes on the `classpath`. Classloader customization comes with additional costs and often requires developers to have a deeper understanding of the classloader mechanism. Besides, using such framework also requires the client project to undergo a series of laborious refactoring operations. `Maven-Shade-Plugin` can allow multiple versions of the same class to be referenced by the client project. However, this tool may cause runtime crashes since its limitations in dealing with Java reflection mechanism.

5.4.3 *When and how can developers use* SENSOR*?*

Due to the loading rules of build tools and asynchronous evolution of libraries, dependency conflicts can easily be introduced into Java projects, when: (1) adding a new library, (2) deleting an existing library, (3) migrating one library to another, (4) upgrading a library, and (5) downgrading a library. If the directly or indirectly introduced multiple versions of a library contain the APIs with identical signature but semantically inconsistent implementations, an SC issue will happen. To timely identify the semantic inconsistencies, developer can apply SENSOR to their projects after changing their dependency configurations.

SENSOR is a fully automated SC issue diagnosis tool. It takes a project's source code (for extracting divergence arguments) and binary code (for static analysis) as inputs. And then, it outputs an SC issue diagnosis report, including: (1) conflicting library versions in the dependency tree (i.e., the loaded and shadowed versions), (2) isomerous conflicting API pairs, the implementation differences between them, and original and actual dependency paths that invoke the isomerous conflicting API pairs, (3) generated tests for triggering the inconsistent behaviors, and (4) test outcome comparison results. The detailed explanations can be referred to `README` file in SENSOR's code repository.

5.4.4 *Can* SENSOR *be applied to the projects written in other programming languages?*

SENSOR's technical architecture for diagnosing SC issues can be generalized to the projects written in other programming languages. However, there are three steps should be adapted based on the programming language's characteristics:

- *Identifying dependency conflicts.* Library or class loading rules of build tools are various among different programming language communities. For instance, Python projects using version ranges (e.g., $\langle \geq 3.0 \wedge <4.0 \rangle$) to reference third-party libraries, and automatically download and install the newest library version satisfying such version constraints [46]. The dependency conflicts can be introduced into Python projects whenever a referenced library releases a newer version, even if the projects' dependency management scripts have no changes. Therefore, the dependency conflict detection should be adapted according to the specific manifestation patterns and root causes in different programming language communities.
- *Analyzing the isomerism between conflicting API pairs.* In our approach, we combine call graph analysis with ASTs (using the existing tool GUMTREE) to identify the isomerism between conflicting API pairs for Java projects. Since it is challenging to perform precise static analysis for software projects written in the interpreted languages, e.g., Python, Javascript, etc., this step should be performed with the aid of an alternative code difference analysis approach.
- *Generating tests.* SENSOR is built on top of EVOSUITE, which is a recognized effective test generation technique for Java projects. Combining the proposed seeding strategy of divergence arguments with the existing test generation technique designed for other programming language projects is feasible. However, the effectiveness of such combination is affected by the technical maturity of the alternative test generation technique.

5.4.5 *What are the limitations of* SENSOR*?*

Similar to other test generators, the effectiveness of SENSOR is limited in its capability of constructing instances of anonymous classes. Besides, our technique can only randomly assign values for a parameterized object constructor, if its invocation contexts cannot be identifed from the source code, which may affect SENSOR's effectiveness.

In terms of static analysis, SENSOR cannot deal with the Java reflection mechanism when identifying the original and actual dependency paths that invoke the isomerous conflicting API pairs. In future, we plan to combine TAMIFLEX [47] (an effective technique aiding static analysis in the presence of reflection and customized class loaders) with SOOT to improve SENSOR's capability of detecting reflection-related SC issues.

6 THREATS TO VALIDITY

Ground truth dataset collection. Collecting the ground truth dataset of SC issues is challenging and can be a threat to the evaluation results. To avoid introducing noises in our dataset, we upgraded/ downgraded the actually-loaded library versions in a series of Java projects. After altering the library versions, we selected the conflicting API pairs that could trigger the `AssertionErrors` when executing the projects' associated tests, as the cases that definitely caused SC issues.

Validity of developers' feedback. In this paper, we rely on developers' feedback to validate the effectiveness and usefulness of SENSOR on both industrial and open source projects. However, there might be different opinions towards the validity of the issue reports for different developers. To mitigate such threat, for the industrial subjects, we invited nine original developers with the domain knowledge of the selected projects for verification. For the open source projects, we did not encounter the controversies for all the evaluated subjects. Therefore, the received feedback demonstrate the effectiveness and usefulness of our approach.

7 Related Work

Dependency conflict. Library conflicts are challenging to detect for a program analysis and difficult to avoid for library developers. Determining whether two or more libraries cannot be built together is an important issue in the quality assurance process of software projects. Blincoe et at. [48] conducted an in-depth study of millions of dependencies across multiple software ecosystems. They found that using a range of versions to declare dependencies could facilitate the automated repairing for dependency conflict issues, when adopting semantic versioning strategies. Yet, since the vast majority of Java projects declare the fixed versions of their referenced third party libraries, semantic versioning does not play a major role in repairing dependency conflict issues in the Java ecosystem.

Ghorbani et al. [49] formally defined eight inconsistent modular dependencies that may arise in Java-9 applications, and proposed a technique Darcy to detect and repair such specified inconsistent dependencies. So far, there are a significant fraction of Java projects that do not adopt Java-9 mechanism. Therefore, an effective approach to diagnosing the SC issues is still urgently needed in the Java ecosystem.

Suzaki et al.' approach [50] mainly focused on conflicts on resource access, conflicts on configuration data, and interactions between uncommon combinations of packages and categorized them to provide useful suggestions on how to prevent and detect such problem. Patra et al. [51] were the first researchers studying the detection strategy for conflicts among JavaScript libraries. They tackled the huge search space of possible conflicts in two phases, i.e., identifying potentially conflicting library pairs and synthesizing library clients to validate conflicts. Soto-Valero et al. [52] presented quantitative empirical evidence about how the immutability of artifacts in `Maven Central` supports the emergence of natural software diversity.

Afterwards, Wang et al. [5] conducted a study to characterize the manifestations of dependency conflicts in Java projects, and presented a static analysis technique to diagnose dependency conflict issues. In approach [7], they also developed Riddle to generate tests to collect crashing stack traces to facilitate dependency conflict diagnosis. Riddle designs a guidance criteria to evolve sets of candidate tests using a GA to maximize the possibility of triggering conflicting APIs. To overcome the unsolvable branch conditions on the path that invokes the target API, it performs mutation operations for each branch to make them be evaluated to be `True` or `False`. However, Riddle cannot effectively detect SC issues, due to its randomly generated arguments and such mutation operations. In our approach, to expose SC issues between conflicting library versions, we designed `Sensor`, an automated technique injecting divergence arguments extracted from source code to generate tests.

Differential testing and analysis. Differential testing and analysis techniques have been used to find bugs across many types of programs [53], [54], [55], [56], [57], [58], [59], [60], [61], [62], [63], [64]. Zhang et al. [65] implemented an isomorphic regression testing approach named Ison, which compares the behaviors of modified programs to check whether abnormal behaviors are induced in the new code versions. Xie et al. [66] presented a differential unit testing technique, Diffut, which compared the methods between different program revisions. Petsios et al. [67] proposed an effective technique Nezha to trigger semantic bugs, using gray-box and black-box mechanisms to generate inputs for differential testing. Nezha is applicable to detecting the semantic differences among different libraries providing similar functionalities. Compared with the above differential testing techniques, Sensor has a different goal: detecting semantic conflicts combining the client projects' invocation contexts.

Wang et al.'s approaches [68], [69] are closely related to our work, which focused on the asynchronous evolution of software libraries and client software. They referred to the case that an API signature remained identical in a new library version but its behaviors changed, as *behavioral backward incompatibilities* (BBIs). In approach [68], they conducted an empirical study to understand the status, major reasons and impact of BBIs of Java software libraries. In approach [69], they proposed DeBBI, a technique which detects BBIs via cross-project testing and analysis. Our work differs from [68], [69] in two aspects: (1) although BBIs have similar symptoms as semantic conflicts, they happen in different scenarios. In approach [69], Wang et al. discussed the impacts of BBIs when client projects upgrade their direct dependencies. While SC issues are caused by loading one library version and shadowing the others, when multiple versions of a library are directly or transitively introduced in a client project. (2) Wang et al. used the client project's associated test code (designed by original developers) to validate the behaviors of APIs with identical signatures but changed implementations in the new library version. However, such cross-version testing could not be effectively adapted to detect SC issues, due to the limited coverage capability of associated tests for transitive dependencies. Hence, we proposed Sensor to generate tests with divergence arguments to expose inconsistent behaviors caused by dependency conflicts.

Test input generation. Existing automated testing generation approaches use many techniques to create inputs for exercising a software under test with minimal human efforts, including feedback-directed random test generation [70], [71], [72], search-based techniques [73], [74], seeding strategies [75], [76], [77], [25], [78], [79], and symbolic reasoning-based test generators [61], [80], [81], [82]. Xu et al. [83] presented a mining approach to building a decision tree model according to the test inputs generated from Java bytecode. It converts Java bytecode into the Jimple representation, extracts predicates from the control flow graph of the Jimple code, and uses these predicates as attributes for organizing training data to build a decision tree. Dallmeier et al. [84] proposed an improved dynamic specification mining technique, Tautoko, to generate test cases. Since previous specification mining technique entirely depends on the observed executions, the resulting specification may be too incomplete to be useful if not enough tests are available. To address this problem, Tautoko explores previously unobserved aspects of the execution space. Their evaluation results shown that the enriched specifications cover more general behaviors and much more exceptional behaviors. Toffola et al. [85] proposed an approach to extract literals from thousands of tests and to adapt information retrieval techniques to find values suitable for a particular domain.

Despite all successes, test generation still suffers from non-trivial limitations in exposing SC issues. First, approach [83] are more effective to deal with the code snippets with simple data types that can be easily convert into Jimple representations. While our empirical study results show that to trigger the real SC issues, effective test generation techniques should have the ability to construct divergence arguments for parameterized complex constructors. Second, the effectiveness of approaches [84], [85] entirely depends on the detection ability of existing test suites of projects under test. Our empirical study provides evidences that combining invocation contexts of constructors in source code can effectively improve the possibility of capturing inconsistent program behaviors caused by SC issues. In addition, existing techniques [75], [76], [77], [25], [78], [79] have proposed different seeding strategies for test input generation, especially for strings and primitive types. Since these strategies generate constructor arguments without considering their invocation contexts, they are not effective in constructing valid class instances to expose conflicting API pairs. SENSOR adopts a new seeding strategy of class instances inspired by the our empirical findings summarized in Section 3. It injects the invocation context information extracted from the source code into class instances with the aim of generating divergence arguments.

8 CONCLUSION AND FUTURE WORK

In this paper, we presented an effective and automated test generation technique SENSOR, which are capable of producing valid inputs to trigger the SC issues. The evaluation results show that SENSOR can achieve a *Precision* of 0.898 and a *Recall* of 0.725 on open source projects and a *Precision* of 0.821 on industrial subjects. SENSOR has detected 306 SC issues from 50 open source projects and submitted 54 issue reports to them. Encouragingly, 38 issue reports (70.4%) have been confirmed by developers as real SC issues. Although SENSOR is designed for detecting SC issues, it can be adapted to other problems arising from library evolution (e.g., safeguarding the reliability upgrading libraries). In future, we plan to combine symbolic execution or fuzzing techniques with our technique to improve its test input exploration capability.

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 and 2019. The work is supported by the National Natural Science Foundation of China (Grant Nos. 61932021, 61902056, 61977014, 61902329, 62002125, 61802164), Shenyang Young and Middle-aged Talent Support Scheme (Grant No. ZX20200272), the Fundamental Research Funds for the Central Universities (Grant No. N2017011), the Hong Kong RGC/GRF grant 16207120, MSRA grant, Guangdong Provincial Key Laboratory (Grant No. 2020B121201001) and Xiamen Youth Innovation Foundation (Grant No. 3502Z20206036).

REFERENCES

[1] L. Bao, Z. Xing, X. Xia, D. Lo, and A. E. Hassan, "Inference of development activities from interaction with uninstrumented applications," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1313–1351, 2018.

[2] K. Jezek, J. Dietrich, and P. Brada, "How java apis break–an empirical study," *Information and Software Technology*, vol. 65, pp. 129–146, 2015.

[3] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc, "A study of library migrations in java," *Journal of Software: Evolution and Process*, vol. 26, no. 11, pp. 1030–1052, 2014.

[4] C. Macho, S. McIntosh, and M. Pinzger, "Automatically repairing dependency-related build breakage," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 106–117.

[5] 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 Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 319–330.

[6] S. Liang and G. Bracha, "Dynamic class loading in the java [tm] virtual machine," *Acm sigplan notices*, vol. 33, pp. 36–44, 1998.

[7] Y. Wang, M. Wen, R. Wu, Z. Liu, S. H. Tan, Z. Zhu, H. Yu, and S.-C. Cheung, "Could I Have a Stack Trace to Examine the Dependency Conflict Issue?" in *Proceedings of the 41th International Conference on Software Engineering*, ser. ICSE. ACM/IEEE, 2019.

[8] J. Wang, G. Dong, J. Sun, X. Wang, and P. Zhang, "Adversarial sample detection for deep neural network through model mutation testing," *arXiv preprint arXiv:1812.05793*, 2018.

[9] "Issue #214 of project openstack-java-sdk," https://github.com/woorea/openstack-java-sdk/issues/214, 2020, accessed: 2020-01-31.

[10] "A pr of issue #214 in project openstack-java-sdk," https://github.com/woorea/openstack-java-sdk/pull/215, 2020, accessed: 2020-01-31.

[11] "Neusoft," https://www.neusoft.com/, 2020, accessed: 2020-01-31.

[12] "Rest-assured," https://github.com/rest-assured/rest-assured, 2020, accessed: 2020-01-31.

[13] "Java-design-patterns," https://github.com/iluwatar/java-design-patterns, 2020, accessed: 2020-01-31.

[14] "Hmily," https://github.com/yu199195/hmily, 2020, accessed: 2020-01-31.

[15] "Issue #1730 of project sentinel," https://github.com/alibaba/Sentinel/issues/1730, 2020, accessed: 2020-01-31.

[16] S. Artzi, S. Kim, and M. D. Ernst, "Recrash: Making software failures reproducible by preserving object states," in *European conference on object-oriented programming*. Springer, 2008, pp. 542–565.

[17] B. Korel and A. M. Al-Yami, "Assertion-oriented automated test data generation," in *Proceedings of IEEE 18th International Conference on Software Engineering*, 1996, pp. 71–80.

[18] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 313–324.

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

[20] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, vol. 15, 2011, p. 35.

[21] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," in *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1997, pp. 108–124.

[22] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155–1179, 2016.

[23] "Options of soot framework," https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/options/soot_options.htm, 2020, accessed: 2020-01-31.

[24] X. Xiao and C. Zhang, "Geometric encoding: forging the high performance context sensitive points-to analysis for java," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 188–198.

[25] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc, "Instance generator and problem representation to improve object oriented code coverage," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 294–313, 2015.

[26] G. Fraser and A. Arcuri, "Evosuite at the sbst 2016 tool competition," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*, 2016, pp. 33–36.

[27] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated unit test generation for classes with environment dependencies," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 79–90.

[28] M. Hericko, M. B. Juric, I. Rozman, S. Beloglavec, and A. Zivkovic, "Object serialization analysis and comparison in java and .net," vol. 38, no. 8, p. 4454, 2003.

[29] "Travis ci," https://travis-ci.org/, 2020, accessed: 2020-01-31.

[30] S. Kim and E. J. Whitehead Jr, "How long did it take to fix bugs?" in *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 2006, pp. 173–174.

[31] J. M. Rojas, G. Fraser, and A. Arcuri, "Seeding strategies in search-based unit test generation," *Software Testing, Verification and Reliability*, vol. 26, no. 5, pp. 366–401, 2016.

[32] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 15–25.

[33] "Assessment report," https://sensordc.github.io/, 2020, accessed: 2020-01-31.

[34] "Jdbc," https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/, 2020, accessed: 2020-01-31.

[35] "Issue #1897 of project aws-sdk-java," https://github.com/aws/aws-sdk-java/issues/1897, 2020, accessed: 2020-01-31.

[36] "Issue #11 of project mia," https://github.com/tdunning/MiA/issues/11, 2020, accessed: 2020-01-31.

[37] "Issue #288 of project ontop," https://github.com/ontop/ontop/issues/288, 2020, accessed: 2020-01-31.

[38] "An experienced developer of project ontop," https://github.com/ghxiao, 2020, accessed: 2020-01-31.

[39] "Issue #2056 of project apache reef," https://issues.apache.org/jira/browse/REEF-2056?filter=-2, 2020, accessed: 2020-01-31.

[40] "Issue #1143 of project rest-assured," https://github.com/rest-assured/rest-assured/issues/1143, 2020, accessed: 2020-01-31.

[41] "Issue #9 of project netty-rest," https://github.com/buremba/netty-rest/issues/9, 2020, accessed: 2020-01-31.

[42] "Issue #809 of motan," https://github.com/weibocom/motan/issues/809, 2020, accessed: 2020-01-31.

[43] "Issue #297 of project restx," https://github.com/restx/restx/issues/297, 2020, accessed: 2020-01-31.

[44] "Osgi," https://www.osgi.org/, 2020, accessed: 2020-01-31.

[45] "Maven-shade-plugin," http://maven.apache.org/plugins/maven-shade-plugin/, 2020, accessed: 2020-01-31.

[46] Y. Wang, M. Wen, L. Yepang, Y. Wang, Z. Li, C. Wang, H. Yu, C. Xu, Z. Zhu, and S.-C. Cheung, "Watchman: Monitoring dependency conflicts for python library ecosystem," in *Proceedings of the 41th International Conference on Software Engineering*, ser. ICSE. ACM/IEEE, 2020, pp. 125–135.

[47] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 241–250.

[48] J. Dietrich, D. Pearce, J. Stringer, A. Tahir, and K. Blincoe, "Dependency versioning in the wild," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 349–359.

[49] N. Ghorbani, J. Garcia, and S. Malek, "Detection and repair of architectural inconsistencies in java," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 560–571.

[50] C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen, and S. Zacchiroli, "Why do software packages conflict?" in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE Press, 2012, pp. 141–150.

[51] 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.

[52] C. Soto-Valero, A. Benelallam, N. Harrand, O. Barais, and B. Baudry, "The emergence of software diversity in maven central," in *MSR 2019-16th International Conference on Mining Software Repositories*. ACM, 2019, pp. 1–11.

[53] T. Zhang and M. Kim, "Automated transplantation and differential testing for clones," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE, 2017, pp. 665–676.

[54] P. Chapman and D. Evans, "Automated black-box detection of side-channel vulnerabilities in web applications," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 263–274.

[55] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *ACM SIGPLAN Notices*, vol. 51, no. 6. ACM, 2016, pp. 85–99.

[56] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 283–294.

[57] S. A. Chowdhury, T. T. Johnson, and C. Csallner, "Cyfuzz: A differential testing framework for cyber-physical systems development environments," in *International Workshop on Design, Modeling, and Evaluation of Cyber Physical Systems*. Springer, 2016, pp. 46–60.

[58] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 185–194.

[59] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 621–631.

[60] R. Lämmel and W. Schulte, "Controllable combinatorial coverage in grammar-based testing," in *IFIP International Conference on Testing of Communicating Systems*. Springer, 2006, pp. 19–38.

[61] 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.

[62] B. Ray, M. Kim, S. Person, and N. Rungta, "Detecting and characterizing semantic inconsistencies in ported code," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 367–377.

[63] H. Zhong, S. Thummalapenta, and T. Xie, "Exposing behavioral differences in cross-language api mapping relations," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2013, pp. 130–145.

[64] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao, "Detecting differences across multiple instances of code clones," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 164–174.

[65] 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.

[66] T. Xie, K. Taneja, S. Kale, and D. Marinov, "Towards a framework for differential unit testing of object-oriented programs," in *Second International Workshop on Automation of Software Test (AST'07)*. IEEE, 2007, pp. 5–5.

[67] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 615–632.

[68] S. Mostafa, R. Rodriguez, and X. Wang, "Experience paper: a study on behavioral backward incompatibilities of java software libraries," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 215–225.

[69] L. Chen, F. Hassan, X. Wang, and L. Zhang, "Taming behavioral backward incompatibilities via cross-project testing and analysis,"

in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 112–124.

[70] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip, "A framework for automated testing of javascript web applications," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 571–580.

[71] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 75–84.

[72] M. Pradel and T. R. Gross, "Fully automatic and precise detection of thread safety violations," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 521–530.

[73] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev, "Efficient search for inputs causing high floating-point errors," in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2014, pp. 43–52.

[74] 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.

[75] N. Alshahwan and M. Harman, "Automated web application testing using search based software engineering," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 3–12.

[76] J. M. Rojas, G. Fraser, and A. Arcuri, "Seeding strategies in search-based unit test generation," *Software Testing, Verification and Reliability*, vol. 26, no. 5, pp. 366–401, 2016.

[77] G. Fraser and A. Arcuri, "The seed is strong: Seeding strategies in search-based software testing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 121–130.

[78] P. McMinn, M. Shahbaz, and M. Stevenson, "Search-based test input generation for string data types using the results of web queries," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 141–150.

[79] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators," *Software Testing, Verification and Reliability*, vol. 16, no. 3, pp. 175–203, 2006.

[80] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.

[81] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[82] S. Thummalapenta, T. Xie, N. Tillmann, J. De Halleux, and Z. Su, "Synthesizing method sequences for high-coverage testing," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 189–206, 2011.

[83] W. Xu, T. Ding, H. Wang, and D. Xu, "Mining test oracles for test inputs generated from java bytecode," in *2013 IEEE 37th Annual Computer Software and Applications Conference*. IEEE, 2013, pp. 27–32.

[84] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, "Generating test cases for specification mining," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 85–96.

[85] L. Della Toffola, C.-A. Staicu, and M. Pradel, "Saying hi!is not enough: Mining inputs for effective test generation," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 44–49.

Ying Wang received her doctoral degree in software engineering from Northeastern University, China, in 2019. She is currently an associate professor at the Software College, Northeastern University, China. Her research interests include program analysis, dependency management, and software ecosystems. Her research work has been regularly published in top conferences and journals in the research communities of software engineering, including ICSE, ESEC/FSE, and TSE and so on. She joined Microsoft Research Asia Star Casting Scheme for Young Scholars (2020), and received Nominees Award for Outstanding Doctoral Dissertation of China Computer Federation (CCF) in 2020. More information about her can be found at: http://faculty.neu.edu.cn/swc/wangying/.

Rongxin Wu received the PhD degree from HKUST, in 2017. He is currently an associate professor at the Department of Cyber Space Security, Xiamen University. His research interests include program analysis, software security, and mining software repository. His research work has been regularly published in top conferences and journals in the research communities of program languages and software engineering, including POPL, PLDI, ICSE, FSE, ISSTA, ASE, TSE and EMSE and so on. He has served as a reviewer in reputable international journals and a program committee member in several international conferences (ASE 2021, ASE 2020 Tool Demo, ISSTA 2019 Tool Demo and so on). He is a two-time recipient of the ACM SIGSOFT Distinguished Paper Award. More information about him can be found at: https://wurongxin1987.github.io/wurongxin.xmu.edu.cn/

Chao Wang is a Master student at the Software College, Northeastern University, China. His main research interests include program analysis, dependency management, and software testing.

Ming Wen is now an associate professor at the School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China. He received his PhD from the department of computer science and engineering at the Hong Kong University of Science and Technology (HKUST). His research interests include program analysis, fault localization and repair, and software security. His research work has been regularly published in top conferences and journals in the research communities of software engineering, including ICSE, FSE, ASE, TSE and EMSE and so on. More information about him can be found at: https://justinwm.github.io.

Yepang Liu received the doctoral degree in computer science and engineering from The Hong Kong University of Science and Technology (HKUST). In 2018, he joined the Department of Computer Science and Engineering of the Southern University of Science and Technology (SUSTech) as a tenure-track assistant professor. His research interests include empirical software engineering, software testing and analysis, cyber-physical systems, mobile computing, and cybersecurity. He published widely in top software engineering venues and has received two ACM SIGSOFT Distinguished Paper Awards. He also participates actively in program and organizing committees of major international conferences and received ACM SIGSOFT Service Award for his contribution to the successful organization of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014).

Shing-Chi Cheung received his Bachelor's degree in Electrical and Electronic Engineering from the University of Hong Kong, and his PhD degree in Computing from the Imperial College London. After doctoral graduation, he joined the Hong Kong University of Science and Technology (HKUST) where he is a professor of Computer Science and Engineering. He founded the CASTLE research group at HKUST and co-founded in 2006 the International Workshop on Automation of Software Testing (AST). He serves on the editorial board of Science of Computer Programming (SCP) and Journal of Computer Science and Technology (JCST). He was an editorial board member of the IEEE Transactions on Software Engineering (TSE, 2006-9) and Information and Software Technology (IST, 2012-5). He participates actively in the program and organizing committees of major international software engineering conferences. He chaired the 19th Asia-Pacific Software Engineering Conference (APSEC) in 1996, 1997 and 2012. He was the General Chair of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014). He is an extended member of the ACM SIGSOFT executive committee. He owns four patents in China and the United States. His research interests lie in boosting the quality of software applications using program analysis, testing, debugging, synthesis, repository mining, and artificial intelligence. Target applications include Android apps, open-source software, deep learning systems, smart contracts and spreadsheets. He is a distinguished member of the ACM and a fellow of the British Computer Society.

Hai Yu received a B.E. degree in Electronic Engineering from Jilin University, China, in 1993 and a PhD degree in Computer Software and Theory from Northeastern University, China, in 2006. He is currently an Associate Professor of Software Engineering at the Northeastern University, China. His research interests include complex networks, chaotic encryption, software testing, software refactoring, and software architecture. At present, he serves as an Associate Editor for the International Journal of Bifurcation and Chaos, Guest Editor for Entropy, and Guest Editor for the Journal of Applied Analysis and Computation. In addition, he was a Lead Guest Editor for Mathematical Problems in Engineering during 2013. Moreover, he has served different roles at several international conferences, such as Associate Chair for the 7^{th} IWCFTA in 2014, Program committee Chair for the 4^{th} IWCFTA in 2010, Chair of the Best Paper Award Committee at the 9^{th} International Conference for Young Computer Scientists in 2008, and Program committee member for the $3^{rd} - 13^{th}$ IWCFTA and the 5^{th} Asia Pacific Workshop on Chaos Control and Synchronization.

Chang Xu received his doctoral degree in computer science and engineering from The Hong Kong University of Science and Technology, Hong Kong, China. He is a full professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nanjing University. He participates actively in program and organizing committees of major international software engineering conferences. He co-chaired the MIDDLEWARE 2013 Doctoral Symposium, FSE 2014 SEES Symposium, and COMPSAC 2017 SETA Symposium. His research interests include big data software engineering, intelligent software testing and analysis, and adaptive and autonomous software systems. He is a senior member of the IEEE and member of the ACM.

Zhiliang Zhu received an M.S. degree in Computer Applications and a PhD degree in Computer Science from Northeastern University, China. He is currently a full professor at Software College, Northeastern University, China. His main research interests include software engineering, complexity software systems, and applications of complex network theorie. He has authored and co-authored over 130 international journal papers and 100 conference papers. In addition, he has published five books, including *Introduction to Communication* and *Program Designing of Visual Basic .NET*, etc. He is also the recipient of nine academic awards at national, ministerial, and provincial levels. Prof. Zhu has served in different capacities at many international journals and conferences. Currently, he serves as Co-Chair of the $1^{st} - 13^{th}$ International Workshop on Chaos-Fractals Theories and Applications. He is a senior member of the Chinese Institute of Electronics and the Teaching Guiding Committee for Software Engineering under the Ministry of Education.