CONI: Detecting Database Connector Bugs via State-Aware Test Case Generation

Wenqian Deng
KLISS, BNRist, School of Software,
Tsinghua University,
Beijing, China

Jie Liang*
KLISS, BNRist, School of Software,
Tsinghua University,
Beijing, China

Zhiyong Wu
KLISS, BNRist, School of Software,
Tsinghua University,
Beijing, China

Jigzhou Fu
KLISS, BNRist, School of Software,
Tsinghua University,
Beijing, China

Mingzhe Wang
KLISS, BNRist, School of Software,
Tsinghua University,
Beijing, China

Yu Jiang*
KLISS, BNRist, School of Software,
Tsinghua University,
Beijing, China

Abstract—**Database connectors are widely used in many applications to facilitate flexible and convenient database interactions. Potential bugs in database connectors can lead to various abnormal behaviors within applications, such as returning incorrect results or experiencing unexpected connection interruption. However, existing DBMS fuzzing works cannot be directly applied to testing database connectors as they mainly focus on SQL generation and use a small subset of connector interfaces. Automated test case generation also struggles to generate effective test cases that explore intricate interactions of database connectors due to a lack of domain knowledge.**

The main challenge in testing database connectors is generating semantically correct test cases that can trigger various connector state transitions. To address that, we propose CONI, a framework designed for detecting logic bugs of database connectors with state-aware test case generation. First, we define the database connector state model by analyzing the corresponding standard specification. Building upon this model, CONI generates interface call sequences within test cases to encompass various state transitions. After that, CONI generates suitable parameter values based on the parameter information and contextual information collected during runtime. Then the test cases are executed on a target and a reference database connector. Inconsistent results indicate potential bugs. We evaluated CONI on 5 widely-used JDBC database connectors, namely MySQL Connector/J, MariaDB Connector/J, AWS JDBC Driver for MySQL, PGJDBC, and PG JDBC NG. In total, CONI reported 44 previously unknown bugs, of which 34 have been confirmed.

I. INTRODUCTION

Database connectors, also known as database drivers, are intermediary software components allowing applications to communicate with databases [1]. Database connectors translate application requests into commands that databases understand and then translate the results from the database into a format that the application can use. Through database connectors, developers can perform various operations such as connecting to the database, executing queries, and managing transactions. Database connectors serve as crucial intermediaries, facilitating communication between applications and databases. Therefore, ensuring the reliability of database connectors is important, as any shortcomings or bugs in their functionality can impact the overall performance of the software system.

To provide applications with increased flexibility and convenience in their interactions with databases, most database vendors implement database connectors following unified standards such as JDBC (Java Database Connectivity) [2] and ODBC (Open Database Connectivity) [3]. Due to the richness of functionality provided by these standards and the variety of use cases for database connectors, database connector implementations may have many potential bugs. The issue tracking systems for popular JDBC implementations such as PGJDBC [4] and MariaDB Connector/J [5] have already collected thousands of bug reports in recent years. These bugs can result in various abnormal behaviors within the application, such as returning incorrect results or experiencing unexpected connection interruptions [6, 7, 8, 9].

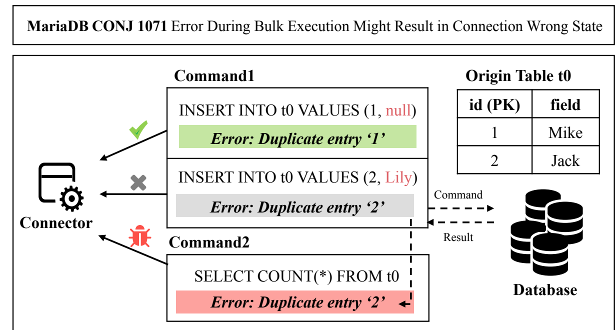


Fig. 1: A real-world bug example of MariaDB Connector/J. An error in the second batch INSERT (i.e., duplicate primary key) caused the following SELECT to return the prior error message instead of query results.

Figure 1 illustrates a recently fixed bug in MariaDB Connector/J [7]. This bug directly affects the functionality of applications that use the connector. In detail, a sequence of statements is executed through batch processing (i.e., `executeBatch()`). The second INSERT statement fails because of the duplicate entry (i.e., 2), and then all subsequent SELECT queries on the

*Jie Liang and Yu Jiang are the corresponding authors.

connection return misaligned results. The developers clarify that the bug stems from an issue in the batch operation, where the batch is split into multiple queries sent to the server. This issue arises due to a change in the type of parameter data within the batch. Specifically, in Figure 1, the "`field`" in the first INSERT statement is of type `string`, while the other has a `null` value. If an error occurs during any of these statements, the connector may fail to read the complete results of all statements, leading to an incorrect connection state. Subsequent commands on this corrupted connection may retrieve residual data from the preceding error, yielding inaccurate query results.

Although there are many existing DBMS-related fuzzing works, they are hard to be directly applied to testing database connectors. On the one hand, most fuzzers primarily concentrate on generating effective SQL queries [10, 11, 12, 13, 14], whereas database connectors are not directly involved in the execution of these queries. Therefore, it is challenging for existing methods to assess the intricate logic embedded within database connectors thoroughly. On the other hand, existing fuzzers exhibit limited interaction with the database connector. They either refrain from utilizing the database connector entirely or leverage only a small subset of its interfaces to transmit SQL queries. For example, SQLANCER [11] relies on JDBC solely to execute SQL queries and retrieve results, without exploring additional functionalities like modifying configuration properties or batch execution. In addition, works like RANDOOP [15] and EVOSUITE [16] have already explored how to generate test cases automatically for programs [15, 16, 17]. However, due to the lack of domain knowledge, these tools can only generate test cases that cover limited scenarios and logic, typically identifying issues like null pointer errors or boundary condition handling errors. Testing connectors requires various interactions with databases, but these tools are almost incapable of generating effective database connections. Therefore, there is a great need for a framework designed specifically for testing database connectors.

The main challenge in testing database connectors lies in *generating semantically correct test cases capable of triggering various connector state transitions.* A test case for testing database connectors comprises the connector interface call sequences and the corresponding input parameter values. Ensuring the semantic correctness of a test case poses a significant hurdle. This correctness encompasses both the accuracy of the interface call sequence and the correctness of the input parameter values. Arbitrarily combining interface call sequences can lead to a combinatorial explosion and introduce semantic errors due to dependency relationships. Additionally, incorrect input parameter values can cause database connectors to return meaningless results or throw errors directly. Furthermore, even with semantically correct test cases, they may still not be sufficient to trigger a wide range of connector state transitions. Database connectors often have internal states, which can change based on factors such as query execution and data processing. To thoroughly test a connector, generating test cases that can trigger various state transitions is essential.

In this paper, we propose CONI for testing database connectors with state-aware test case generation. To the best of our knowledge, it is the first framework specifically designed for testing database connectors. To generate semantically correct test cases, we first model the specification of the database connector standard, identifying the state information. Based on the defined state model, CONI generates semantically correct interface call sequences. Then, CONI generates suitable parameter values based on both the parameter information and contextual information to ensure the correctness of input parameter values. To trigger a wide range of connector state transitions, CONI categorizes interface methods into two sets: one for interface methods that cause state transitions and another for those that do not, assigning a higher weight to the former. After generating test cases, CONI uses differential testing [18] to detect bugs in database connectors by running the same test cases on two different but compatible connectors and identifying inconsistencies in their results.

We applied CONI to 5 widely-used JDBC database connectors mainly provided by major database vendors, including MySQL Connector/J [19], MariaDB Connector/J [5], AWS JDBC Driver for MySQL [20], PGJDBC NG [21], and PostgreSQL JDBC Driver [4], reporting a total of 44 previously unknown bugs. Among them, 34 bugs have been confirmed by the developers of corresponding database connectors.

This paper has the following main contributions:

- We propose state-aware test case generation specifically designed for testing database connectors. Our approach leverages a detailed state model to generate semantically correct interface call sequences and parameter values within test cases.
- We implemented our approach in CONI. It was evaluated on 5 real-world database connectors and 44 bugs were reported, of which 34 have been confirmed.

II. BACKGROUND

Database Connectors. Database connectors [1], also known as database drivers, are software components that enable applications to connect and interact with database management systems (DBMSs). The basic workflow of database connectors is shown in Figure 2. Database connectors initiate the connection with databases and configure connection parameters. Subsequently, they transmit queries to the database for execution, retrieve and process the results as required, and ultimately close the connection to free up resources. For each action in the workflow, the connector standard specifications provide a set of available interface methods that can help facilitate a variety of operations. The database connector can perform a series of functions in the workflow through a sequence of interface calls and the corresponding parameter values.

The workflow indicates that database connectors maintain internal states during the execution of interface call sequences. For example, when a database connection is established, the state of the connection in the database connector changes from closed to open. Modeling these states can help generate semantically correct test cases.

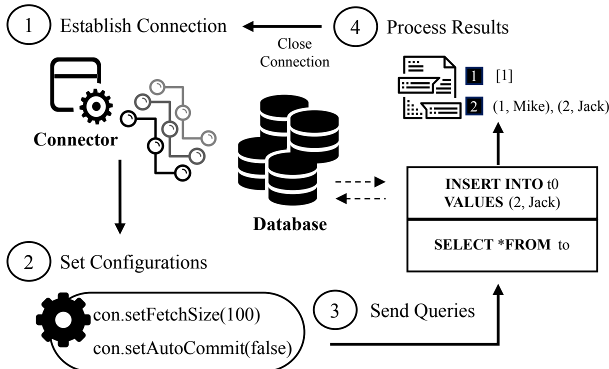


Fig. 2: The basic workflow of database connectors.

Database Connector Specification. Database connectors are usually implemented according to standard specifications to afford applications greater flexibility and convenience. These standard specifications strictly define the parameters, expected operations, and return values (e.g., the error code) of interface methods for interactions between applications and databases. We use JDBC [2] as an example to introduce the specifications of database connectors. JDBC is a widely used standard defining a set of interface methods for Java applications. It supports most DBMSs, allowing developers to switch between the underlying DBMS without changing the codebase if needed.

Interface Method Definition in JDBC. We extract interface method definitions from connector standard specifications such as JDBC. For example, `boolean Statement.execute(String sql)` is an interface method definition. It contains four parts: return type (i.e., `boolean`), interface method (i.e., `Statement.execute`), parameter type (i.e., `String`), and parameter name (i.e., `sql`).

Challenges in Testing Database Connectors. As Figure 3 shows, a *test case* for database connectors always consists of two parts: *a sequence of interface calls* (surrounded by a red dashed line), and each connector interface call needs to be instanced with the corresponding *parameter values* (covered by a blue background).

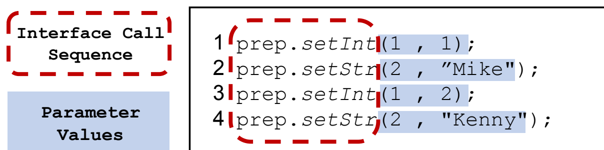


Fig. 3: An example of test cases for database connectors.

The main challenge in testing database connectors is to *generate semantically correct test cases that can trigger various connector state transitions.* First, the semantic correctness of a test case is important but hard to ensure. The semantic correctness encompasses both the correctness of the interface call sequence and the correctness of the input parameter values. Randomly generating sequences by combining various interface methods not only leads to the issue of combinatorial explosion but also results in a large number of invalid combinations due to the semantic relationships between interface methods. For example, in the testing process, to perform operations on a result set, such as accessing metadata or reading the result data, it is necessary to execute a query statement and obtain the result set beforehand. If the semantic correctness of the interface call sequences is not ensured, it may lead to compilation errors. Besides, the parameter values of the interface call also directly influence the semantic correctness. For example, the interface call `stmt.executeQuery`(“`random`”) is semantically incorrect due to being filled with a random string instead of SQL. Consequently, if parameter values are generated randomly, the database connector will directly throw an error, failing to test the deep logic of database connectors.

Additionally, *semantically correct test cases may not be sufficient to trigger various connector state transitions.* Database connectors have internal states that govern their behaviors, undergoing changes influenced by factors like connection status, query execution, and data processing. To ensure comprehensive testing of a connector, it is crucial to formulate test cases encompassing diverse scenarios that can induce a wide range of state transitions. Consequently, effective testing of connectors necessitates the creation of semantically correct test cases with nuanced behaviors.

III. State-Aware Test Case Generation

We design state-aware test case generation to address the challenges in synthesizing semantically correct test cases capable of triggering various connector state transitions for database connector testing. As Figure 4 shows, firstly we build the connector state model by analyzing the corresponding standard specifications, iterating through interface methods, and manually defining the pre-state and post-state for each method. Secondly, Coni synthesizes several semantic-correct interface call sequences according to the constraints in the predefined state model. Thirdly, Coni generates suitable parameter values and passes them to each interface call based on the parameter information and contextual information collected during runtime. Upon generating test cases, Coni utilizes differential testing to pinpoint bugs in database connectors. This involves executing identical test cases on two connectors and comparing their results to identify inconsistencies.

A. State Model Establishment

The interface methods and states of the database connector are closely related. For example, a database connector can establish or close connections, causing the connection state to change. Similarly, it can create a new result set or close an existing one, leading to changes in the result set's state. Consequently, constructing a state model accurately describes the intricate relationship between these interface methods and states and thus facilitates the generation of semantically correct interface call sequences.

State. The state represents the current status of a database connector when it calls interface methods. Generally, a state refers to the values of all relevant attributes at a specific point in time. A database connector has three stateful attributes:

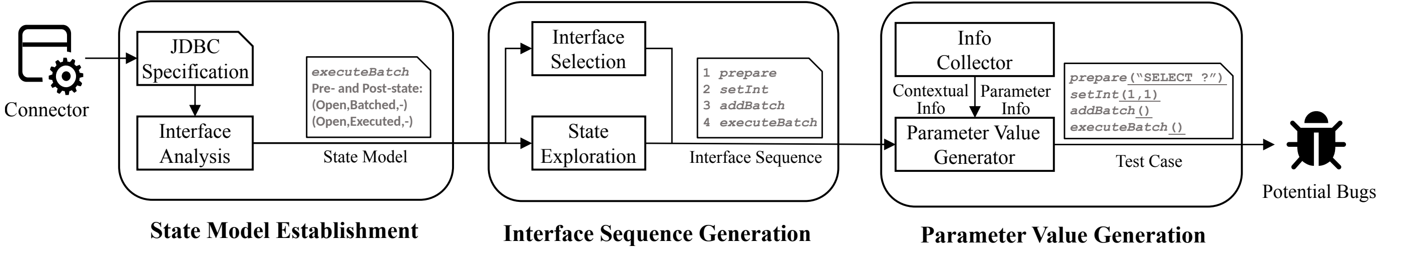


Fig. 4: The Overview of CONI. First, we analyze specifications to define interface pre- and post-states manually, storing this in the state model. Second, CONI generates interface call sequences based on the state model. Third, CONI generates suitable parameter values and passes them to interface calls based on the parameter information and runtime contextual information. Finally, CONI uses differential testing on target and reference connectors to uncover logic bugs through inconsistent results.

connection, statement, and result set. Let C denote the state of the connection, O represent the state of the statement, and R represent the state of the result set. S represents the basic state model of connectors, which is composed of the three states above:

$$S = (C, O, R)$$

We define the values for C, O, and R as follows according to the definitions in the connector standard specification:

$$C \in \{Open, Close\}$$
$$O \in \{Created, Batched, Prepared, Filled, Executed, Close\}$$
$$R \in \{Created, Scrolled, Updated, Close\}$$

Function. Through the analysis of interface methods in the specification, we find that the interactions between connectors and databases involve continuous transitions in states. To conveniently represent the state transitions within the state model, we define *function* objects, which explicitly indicate the state transitions before and after an interface method call. We first define the connector state before and after an interface method call as the *pre-state* and *post-state*, respectively. Then we use F to denote connector functions and define as follows, wherein, I represents the corresponding interface method, S_{pre} represents the pre-state of that interface method, and S_{post} represents the post-state of that interface method:

$$F = (I, S_{pre}, S_{post})$$

The fundamental principle for determining pre- and post-state for an interface method is as follows: the pre-state is determined based on the objects it depends on, and the post-state is determined by its return values. Specifically, the interface method's dependencies define the pre-state, for example, `Statement.executeQuery` requires both statement and connection objects to be open, thus its pre-state is (`Open, Open, -`). The post-state is defined by the interface method's output, for instance, `Statement.executeQuery` returns a new result set, and therefore the post-state updates to (`Open, Open, Created`).

The state model of the database connector is composed of the defined *functions*. To establish the state model, we first retrieve interface methods from the specification and manually analyze the possible pre- and post-states of each interface method based on its input objects and output values. It is a one-time effort since modeling of a specification can be applied to all connectors that implement that standard specification. Then, All the state information is stored as the predefined state model in the form of functions for future use of CONI. Through the state model, CONI can determine whether the state transitions of an interface call sequence are legal, helping generate semantically correct interface call sequences. Table I shows examples of functions extracted from JDBC.

TABLE I: A partial display of the functions extracted from JDBC. The "-" in the table represents that the state can be any state.

Interface Method	Pre-state	Post-state
DriverManager.getConnection	Close, Close, Close	Open, Close, Close
Connection.createStatement	Open, Close, Close	Open, Created, Close
Connection.close	Open, Close, Close	Close, Close, Close
Statement.executeBatch	Open, Batched, -	Open, Executed, -
Statement.close	Open, Not Close, Close	Open, Close, Close
ResultSet.next	Open, -, Created Open, -, Scrolled	Open, -, Scrolled
ResultSet.updateObject	Open, -, Scrolled	Open, -, Updated
ResultSet.close	Open, Not Close, Not Close	Open, Not Close, Close

B. Interface Call Sequence Generation

We design the state-aware interface call sequence generation to generate semantically correct test cases with various connector state transitions. **The insight of the method is that in a semantically correct sequence of interface calls, the post-state of the previous interface method must be equal to the pre-state of the next interface method.** Therefore, with the state information stored in the functions, CONI can generate semantically correct sequences of interface calls. For example, after executing `DriverManager.getConnection`, the state of the database connector changes from `(Close, Close, Close)` to `(Open, Close, Close)`. Next, CONI can invoke `Connection.createStatement` as its pre-state matcheste the current state `(Open, Close, Close)`.

While semantically correct test cases are important, they may not be sufficient to trigger various connector behaviors. Database connectors typically operate based on various

internal state transitions, which are influenced by factors like whether connectors are connected, executing queries, or processing data. Therefore, **another key aspect in creating interface call sequences is ensuring they effectively trigger various state transitions.** Figure 5 shows an example of two generated test cases. Both test cases are semantically correct. However, the second test case is capable of triggering the bug shown in Figure 1, whereas the first test case does not trigger any bugs. When these two test cases are analyzed with state transitions, it is found that the first test case does not make any state changes. In contrast, the second test case goes through multiple state transitions. Based on this discovery, test cases with more state transitions need to be generated to uncover more scenarios in the connectors.

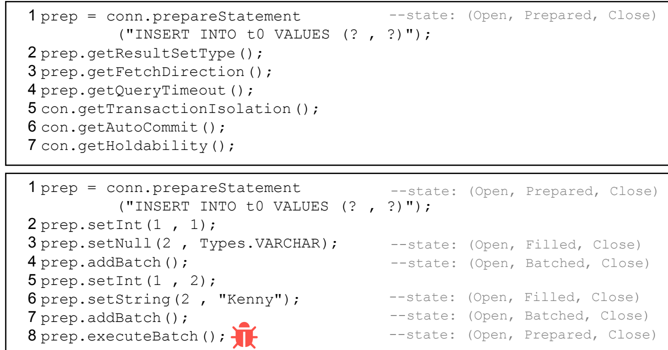


Fig. 5: Both test cases are semantically correct, but only the second triggers the bug in Figure 1. The discrepancy is due to the first test case does not induce any state changes, while the second test case undergoes multiple state transitions.

Algorithm 1 shows the overall process of interface call sequence generation. In general, CONI first reads the predefined state model and categorizes it in two function sets F_c and F_s: F_c for functions that cause state transitions and F_s for those with consistent pre- and post-states. Then, CONI represents the current connector state using the variable S and initializes it (Line 1). CONI defines the initial state as (`Close, Close, Close`). After that, CONI generates a random number N as the interface call sequence size of the test case and selects functions whose pre-state is equal to the current state to add them to the interface call sequence (Lines 5–9). Then, CONI updates the current connector state based on the post-state of the currently selected function (Line 10).

In detail, to trigger a wider range of connector state transitions, CONI selects functions from two different sets F_c and F_s separately when generating the test case (Lines 5–9), assigning a higher weight to functions that cause state transitions. CONI first selects multiple random functions from F_s that do not change the current state (Line 5). These functions enrich the behavior of the connector in the current state. Then, CONI selects a single function from F_c that changes the current state (Line 7-8). The selected function ensures that the connector transitions to a new state. By utilizing the selection approach, CONI ensures that the connector can generate diverse behaviors in a given state, while also preventing it from getting stuck in a single state without state transitions. This approach balances exploring behaviors in current states and promoting state transitions, leading to more effective bug detection.

Algorithm 1: Interface Call Sequence Generation

Input: Predefined function set F_c with inconsistent pre- and post-state, F_s with same pre- and post-state
Output: A newly generated interface call sequence T

1 $S \leftarrow$ The initial state of connector;
2 $N \leftarrow$ Random number of interface calls in T;
3 $T, i \leftarrow \emptyset, 0$;
4 **while** $i < N$ **do**
5 $Avail_s \leftarrow$ `selectRandomAvailableFunc` (F_s, S);
6 $T \leftarrow$ Append Set $Avail_s$ to T;
7 $Avail_c \leftarrow$ `selectRandomAvailableFunc` (F_c, S);
8 $F \leftarrow$ Select a random function $F \in Avail_c$;
9 $T \leftarrow$ Append F to T;
10 $S \leftarrow F.S_{post}$;
11 $i \leftarrow$ The current size of T;
12 **end**
13 **return** T
14 **Function** `selectRandomAvailableFunc`($FuncList$, $CurState$)**:**
15 $AvailList \leftarrow \emptyset$;
16 **foreach** $Func \in FuncList$ **do**
17 **if** $Func.S_{pre} == CurState$ *and* $getRandomBool() == True$ **then**
18 $AvailList \leftarrow$ Append $Func$ to $AvailList$;
19 **end**
20 **end**
21 **return** $AvailList$;
22 **End Function**

C. Parameter Value Generation

The generated interface call sequences in test cases cannot be executed directly, since they still lack suitable parameter values. CONI collects the *parameter information* and *contextual information* to instantiate the interface call sequences.

Parameter Information. CONI uses two parts from the interface method definition to generate parameter values: parameter type and parameter name. ①*Parameter Type*: This helps CONI determine the initial direction for parameter value generation. ②*Parameter Name*: Parameters contain rich semantic information that cannot be directly inferred from the parameter types. For instance, certain interface methods require SQL as input with the parameter type `String`. The parameter names can reveal semantic information, such as `sql` if SQL is required. Therefore, CONI also saves parameter names to assist in parameter generation. To collect parameter types and names, CONI iterates interface method definitions in the specification and stores corresponding information.

Contextual Information. To generate suitable parameter values for some special parameters such as SQL, it is not only necessary to collect the parameter information but also to maintain the contextual information during runtime. The contextual information consists of two parts: the database schema and the previously generated values. ①*Database Schema*: The

Algorithm 2: Parameter Value Generation

Input: A given interface call sequence T, parameter information $Info$
Output: T with newly generated parameter values

```
1  Context ← ∅;
2  foreach Function F ∈ T do
3      foreach Parameter P ∈ F do
4          Name_p, Type_p ← getParmInfo(P, F, Info);
5          if isContextualType(Name_p, Type_p) then
6              PreviousValue ←
                 getDependentValue(F, Context.Values);
7              P.Value ←
                 genValueOnDependency(PreviousValue);
8          end
9          else if isSQLType(Name_p, Type_p) then
10             P.Value ← genRandomSQL();
11             Context.Schema ← Update if SQL changes
                 schema;
12         end
13         else if isConfig(Name_p, Type_p) then
14             P.Value ←
                 genRandomConfigValue(Name_p);
15         end
16         else if isPrimitiveType(Name_p, Type_p) then
17             P.Value ←
                 genRandomValueOnType(Type_p);
18         end
19     end
20     Context.Values ← Save the generated values for F;
21     return T
22 end
```

connector needs to interact extensively with the database. To enable CONI to generate meaningful parameter values, it is necessary to save and maintain the database schema. For example, to generate a valid SQL query, CONI needs to know schema information such as table names and columns. ②*Previously Generated Values*: Some interface methods are related, and the parameter values of subsequent interface calls may depend on the previous interface calls. For example, When CONI first calls `prepareStatement` to cache an SQL statement and then calls `setInt` to set values, it must refer to the previously generated values of `prepareStatement`. To collect contextual information, CONI maintains the database schema and the previously generated values during runtime. For database schema, CONI tracks it by updating table and column objects when executing SQL queries. For previously generated values, taking the second test case in Figure 5 as an example, CONI saves the parameter values of the first call `prepareStatement` (e.g., `INSERT INTO t0 VALUES (?, ?)`). When generating values for the subsequent call `setInt`, CONI looks up *previously generated values* from `prepareStatement` to get the SQL statement and analyzes it to determine the column type in `t0` using *database schema*, then generates suitable values.

Algorithm 2 illustrates the process of parameter value generation for a given interface call sequence. CONI first iterates over each parameter in each function of the given sequence and obtains the corresponding parameter type $Type_p$ and name $Name_p$ from the collected parameter information (Lines 2–4). Then, CONI determines which method to use for generating parameter values based on the parameter type and name. CONI has established a mapping from parameter names to corresponding generation methods for non-primitive parameters (Lines 5–17).

In detail, if the parameter is contextual, in other words, it has a dependence on previously generated values, the dependent values for the parameter are first queried based on the dependency relationship (Lines 5–7). This dependency relationship is predefined and stored in the form of a dictionary. If the parameter is SQL, CONI generates a random SQL query and updates the schema if the SQL query alters the database (Lines 9–11). Moreover, if the parameter is configuration, CONI generates valid configuration values based on the specification (Lines 13–15). Finally, if the parameter does not fall into any above conditions, it is considered a primitive type, and a random method corresponding to its parameter type is called to generate values randomly (Lines 16–17). For primitive types, CONI deliberately creates invalid or extreme values for interface calls to cover more edge cases. After traversing all parameters of a function and generating values, CONI saves the generated values in the contextual information (Line 20).

IV. IMPLEMENTATION

We implement CONI on JDBC since Java is one of the most popular programming languages [22] and JDBC is used by tens of thousands of developers worldwide [23]. We define the pre-state and post-state for 70 interfaces to build the state model, which consists of 25 different states and 91 transitions between them. Then the state model is stored in files for use by CONI. The details of the state model are listed on CONI's website[1]. The implementation of CONI comprises two main components: test case generation and bug detection.

In test case generation, when generating parameter values, CONI generates SQL queries according to a subset of the SQL-92 BNF rules [24]. CONI focuses on basic CREATE TABLE, INSERT, and SELECT clauses and thus can easily adapt to various databases. Additionally, CONI generates valid configuration values by building a customized dictionary for configurations. In bug detection, CONI employs differential testing in database connectors by running identical test cases on two compatible connectors and comparing their results for consistency. CONI wraps the results as strings in a wrapper result object with a flag for errors and an empty string for null values, to facilitate the comparison between results. If the results are different, CONI may discover a potential logic bug and the bug will be reported to developers for their verification. Differential testing is effective for CONI because many connectors, like MariaDB Connector/J and MySQL Connector/J, are designed to be compatible, which means both connectors can send the same commands to the same database and get the results. CONI conducts comparisons on MySQL-compatible and PostgreSQL-compatible connectors.

[1]https://github.com/THU-WingTecher/Coni

V. Evaluation

To evaluate the effectiveness and efficiency of CONI in detecting bugs in database connectors, we sought to answer the following questions:

- **RQ1**: How well does CONI perform in detecting bugs in real-world connectors?
- **RQ2**: How does CONI perform compared to other existing testing techniques?
- **RQ3**: How is the effectiveness of techniques in state-aware test case generation?

A. Evaluation Setup

Tested Database Connectors. We tested CONI on 5 JDBC connectors, namely MySQL Connector/J [19], MariaDB Connector/J [5], AWS JDBC Driver for MySQL [20], PGJDBC NG [21], and PostgreSQL JDBC Driver [4]. These database connectors are primarily sourced from popular database vendors such as MySQL [25], MariaDB [26], PostgreSQL [27], and AWS [28]. PGJDBC NG is a popular open-source connector for PostgreSQL. We chose the latest version available at that time for experiments, which is MariaDB Connector/J 3.3.0, MySQL Connector/J 8.1.0, AWS MySQL JDBC 1.1.9, PG JDBC 42.7.2, PG JDBC NG 0.8.9.

Basic Setup. The experiments were conducted on a machine running 64-bit Ubuntu 20.04 with an AMD EPYC 7742 Processor @ 2.25 GHz. We ran CONI in a docker container with 5 CPU cores and 40 GiB of main memory. For the databases, we chose the one compatible with the corresponding connector. In detail, when testing MariaDB Connector/J, MySQL Connector/J, and AWS MySQL JDBC we used MySQL 8.0.32 as the database for the connection, when testing PG JDBC and PG JDBC NG, we used PostgreSQL 15.

B. Database Connector Bugs

Overall Results. CONI has reported a total of 44 bugs on five database connectors. Among them, 34 reported bugs have been confirmed as previously unknown bugs, and 10 bugs are still being analyzed. At the time of writing this paper, 17 confirmed bugs have been fixed. Among fixed bugs, 1 was labeled as critical and 10 were labeled as major due to their impact on many users. Besides, the developers of MariaDB Connector/J and AWS MySQL JDBC have added 14 of our test cases to their official test sets.

Statistic. Table II shows the statistics of bugs, including the number of reported bugs and confirmed bugs among all the reported bugs in each connector. CONI reported 44 bugs in the five database connectors. Specifically, CONI reported 14, 15, 6, 3, and 6 bugs in MariaDB Connector/J, MySQL Connector/J, AWS MySQL JDBC, PG JDBC, and PG JDBC NG, respectively. They were detected because CONI synthesizes various semantic correct test cases with state-aware test case generation. These test cases can explore various behaviors of the database connector and detect previously unknown bugs.

Case Study. To give an intuition on what kinds of bugs CONI can detect, we show a selection of confirmed bugs. For brevity, the bugs are reduced to demonstrate the core problem instead of providing the original test cases. Then, we analyze why CONI can find these bugs to show its effectiveness.

TABLE II: Number of previously unknown bugs reported and confirmed by CONI

Connector	Version	Reported	Confirmed
MariaDB Connector/J	3.3.0	14	11
MySQL Connector/J	8.1.0	15	15
AWS MySQL JDBC	1.1.9	6	6
PG JDBC	42.7.2	3	2
PG JDBC NG	0.8.9	6	0
Total	-	44	34

a) Case 1: OutOfMemoryError Bug in MariaDB Connector/J: Listing 1 shows details messages of a `OutOfMemoryError` bug detected in MariaDB Connector/J. The test case in Listing 1 triggers a "java.lang.OutOfMemoryError" of the database connectors and causes critical memory allocation failure during routine operations, which can lead to application crashes and instability in production environments. This bug occurs when running a `execute()` method after setting a huge fetch size. The cause of this bug is that MariaDB Connector/J does not limit the size of the fetch size, leading to potential Out-of-memory risks.

The reason for detecting the bug by CONI. The bug is invoked by an unexpected parameter value for the `setFetchSize` interfaces, which is rarely set by users as well as testing tools. As a result, the bug hides for over 5 years. CONI can correctly fill the interface with random parameter values and keep the semantic correctness of test cases, which could explore uncovered codes by existing tools.

Listing 1: OOM bug in MariaDB Connector/J

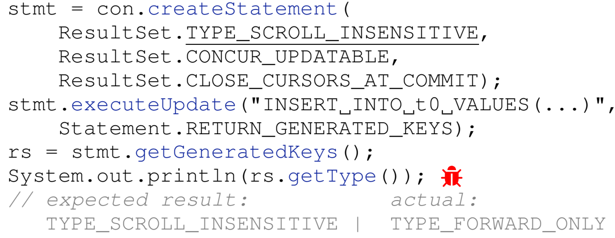
```
stmt = con.createStatement();
stmt.setFetchSize(1600000000);
stmt.execute("SELECT id FROM t0 WHERE id >= 1");
// java.lang.OutOfMemoryError: Java heap space
    error
```

b) Case 2: ResultSet Type Mismatch Bug in MariaDB Connector/J: The following Listing 2 shows a bug identified in MariaDB Connector/J, which affects the `Statement.getResultSetType()` method, causing it to fail in changing the `ResultSet` type as expected. This type mismatch can lead to unexpected behavior in applications that rely on the `ResultSet` type for processing, potentially causing logic errors or exceptions in `ResultSet` handling. This bug occurs when a `Statement` is created `TYPE_SCROLL_INSENSITIVE`. However, the following `ResultSet` retrieved from the `getGeneratedKeys()` method after an update operation does not reflect the type set initially. Instead, it defaults to `TYPE_FORWARD_ONLY`.

The reason for detecting the bug by CONI. The bug is caused by an oversight in handling modifications to the `ResultSet` metadata. CONI can trigger this bug because it changes the configuration values when creating `Statement` and `ResultSet` objects, instead of using the default values. Therefore,

when querying the configuration values again, it is discovered that the configuration has not been successfully changed.

Listing 2: Type mismatch bug in MariaDB Connector/J

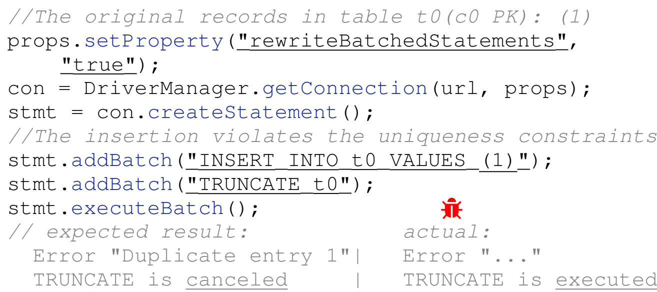


```
stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE,
    ResultSet.CLOSE_CURSORS_AT_COMMIT);
stmt.executeUpdate("INSERT INTO t0 VALUES(...)",
    Statement.RETURN_GENERATED_KEYS);
rs = stmt.getGeneratedKeys();
System.out.println(rs.getType());
// expected result:             actual:
   TYPE_SCROLL_INSENSITIVE |   TYPE_FORWARD_ONLY
```

c) Case 3: Inconsistent Statement Execution Bug in MySQL Connector/J: In Listing 3, an inconsistency bug has been found in the MySQL Connector/J when the `rewriteBatchedStatements` is set to true. This inconsistency can lead to data integrity issues and unpredictable application behavior, as developers cannot rely on the consistent execution of batches. In detail, with `rewriteBatchedStatements` set to true, according to the developers of MySQL Connector/J, when one of the queries in the batch violates the database's uniqueness constraints and fails, the subsequent queries in the batch will not be executed. However, in our test case, we find that the preceding statement executes successfully (i.e., `TRUNCATE`), resulting in the data in table `t0` truncated.

The reason for detecting the bug by CONI. This bug is caused by setting configuration values `rewriteBatchedStatements` and executing SQL queries that violate database constraints. CONI can discover this bug because it first changes the configuration value when connecting to the database, and then executes randomly generated SQL queries in a batch. Therefore, by comparing the return results of different connectors, CONI notices an inconsistency and thus discovers this bug.

Listing 3: Inconsistent statement execution bug in MySQL Connector/J



```
//The original records in table t0(c0 PK): (1)
props.setProperty("rewriteBatchedStatements",
    "true");
con = DriverManager.getConnection(url, props);
stmt = con.createStatement();
//The insertion violates the uniqueness constraints
stmt.addBatch("INSERT INTO t0 VALUES (1)");
stmt.addBatch("TRUNCATE t0");
stmt.executeBatch();
// expected result:                actual:
  Error "Duplicate entry 1"|       Error "..."
  TRUNCATE is canceled     |       TRUNCATE is executed
```

In summary, CONI has reported a total of 44 bugs on five database connectors, among them 34 have been confirmed. The results indicate that CONI can detect previously unknown bugs in database connectors, which adequately answers RQ1.

C. Comparison with Existing Techniques

Compared Techniques. To the best of our knowledge, CONI is the first fuzzing framework for database connectors. Therefore, we selected closely related work as the evaluation baseline. RANDOOP [15] and EVOSUITE [16] are two prominent and open-source automated tools for generating test cases of Java programs, while CONI generates test cases for JDBC, which can be considered as a Java library. In addition, SQLANCER [11] is a popular open-source tool for testing databases using JDBC. We implemented SQLANCER^{+} by adapting the target database connector and collecting results from different connectors to identify inconsistencies. We evaluated these techniques using two metrics, namely branch coverage and unique detected bugs. Coverage was collected using Jacoco [29] instrumentation. The bugs were deduplicated automatically by comparing the interface call information and the returned result, identifying inconsistencies. Then the deduplicated bugs were reported to developers for their verification. We ran the testing tools on five tested database connectors for 24 hours and collected the branch coverage and unique detected bugs.

Coverage. Table III displays the number of branches covered by each technique in 24 hours. The result shows that CONI covered 5950, 6608, and 6587 more branches than SQLANCER^{+}, RANDOOP, and EVOSUITE respectively. The main reason that CONI covered more branches is that CONI is capable of generating effective test cases that cover various functional scenarios. Specifically, SQLANCER^{+} only uses the database connector to send SQL queries, utilizing `getConnection` to create a database connection, `createStatement` to create a statement, and `execute` to execute SQL queries. For RANDOOP and EVOSUITE, the main problem is that due to the lack of domain knowledge, these tools can only generate test cases that cover limited scenarios and logic. Testing connectors requires complex interactions with databases, but these tools, without the state model, are almost incapable of generating effective database connections. CONI attempts to alter configurations in the database connector, perform batch executions, and modify query results. Therefore, CONI can cover more branches than other techniques.

TABLE III: Number of branches covered by each technique in 24 hours

Connector	CONI	SQLANCER^{+}	RANDOOP	EVOSUITE
MariaDB Connector/J	1073	466	583	581
MySQL Connector/J	2430	1256	1473	1489
AWS MySQL JDBC	2826	1445	1734	1739
PG JDBC	1660	987	1197	1181
PG JDBC NG	2425	1796	1621	1597
Total	10414	5950	6608	6587
Improvement	-	4464↑	3806↑	3827↑

Bugs. Table IV shows the number of detected bugs in database connectors by each technique in 24 hours. During the evaluation, CONI found 5, 6, 3, 2, and 5 bugs in MariaDB Connecor/J, MySQL Connector/J, AWS MySQL JDBC, PG JDBC, and PG JDBC NG respectively, while other techniques did not find any bug. The main reason is as follows: SQLANCER^{+} did not call the interface methods that can trigger bugs, nor did it attempt to pass illegal values or set configurations for interface methods. For example, SQLANCER^{+} cannot find the

bug in Listing 1 because it did not call the `setFetchSize` method. Randoop and EvoSuite focused on unit testing, which can only generate the test cases for several simple methods. Although they reported some potential errors during the experiment, they were all false positives which are `NullPointerException` caused by invalid sequences of interface calls. The performance improvement of Coni is primarily attributed to the validity and complexity of the generated test case. Coni generates test cases that explore more state space because the state-aware method covers more interface methods and their interactions.

TABLE IV: Number of detected bugs in database connectors by each technique in 24 hours.

Connector	Coni	SQLancer^{+}	Randoop	EvoSuite
MariaDB Connector/J	5	0	0	0
MySQL Connector/J	6	0	0	0
AWS MySQL JDBC	3	0	0	0
PG JDBC	2	0	0	0
PG JDBC NG	5	0	0	0
Total	21	0	0	0
Improvement	-	21↑	21↑	21↑

In summary, Coni is unique in its ability to find bugs in database connectors, and compared to other techniques, it can cover more branches within the database connectors, which adequately answers RQ2.

D. Effectiveness of Test Case Generation

To understand the contribution of each technique in Coni, we implemented two variants of Coni$^{!s}$ and Coni$^{!p}$. Coni$^{!s}$ disabled the state-aware interface call sequence generation and generated random sequences. Coni$^{!p}$ disabled the parameter value generation method and generated the values randomly. To avoid early errors in test cases from failed database connections, we used `DriverManager.getConnection` to build valid connections first for Coni$^{!s}$ and Coni$^{!p}$ and then subsequent interface call sequences or input parameter values were generated randomly.

Bugs. Table V shows the number of detected bugs by Coni$^{!s}$, Coni$^{!p}$ and Coni. Specifically, Coni found 11 and 14 more bugs than Coni$^{!s}$ and Coni$^{!p}$ respectively. The evaluation results show that state-aware interface sequence call generation and the parameter value generation method help Coni detect more potential bugs in database connectors. The main reason is that Coni, by generating effective test cases, can trigger various scenarios in the database connector, while Coni$^{!s}$ and Coni$^{!p}$ cause many interface calls to throw errors at runtime, preventing the execution of subsequent logic.

Semantic Correctness Rate. Figure 6 shows the semantic correctness rate of test cases of Coni$^{!s}$, Coni$^{!p}$, and Coni. In our evaluation, we considered an interface call semantically correct if it did not throw an error. The result shows that the semantic correctness rate of test cases of Coni accounted for 88.2%, 87.9%, 85.1%, 81.9%, and 84.8%, on MariaDB Connecor/J, MySQL Connector/J, AWS MySQL JDBC, PG JDBC, and PG JDBC NG respectively. Specifically, Coni

TABLE V: Number of detected bugs in database connectors by Coni$^{!s}$, Coni$^{!p}$, and Coni in 24 hours

Connector	Coni$^{!s}$	Coni$^{!p}$	Coni
MariaDB Connector/J	3	2	5
MySQL Connector/J	2	1	6
AWS MySQL JDBC	1	1	3
PG JDBC	1	1	2
PG JDBC NG	3	2	5
Total	10	7	21

included 8.4%, 11.7%, 8.9%, 8.9%, and 12.1% more semantic correct interface calls than Coni$^{!s}$, and 47.9%, 43.5%, 41.4%, 43.7%, and 45.3% more than Coni$^{!p}$. It indicates that Coni is capable of generating more effective test cases with a higher rate of semantic correctness. The main reason is that through the state model, Coni can ensure that dependencies between interface calls in the sequence are met, reducing the likelihood of errors during interface execution. By collecting rich information to generate parameter values, Coni can produce valid parameter values, thus decreasing the probability of execution termination due to invalid parameters during interface execution.

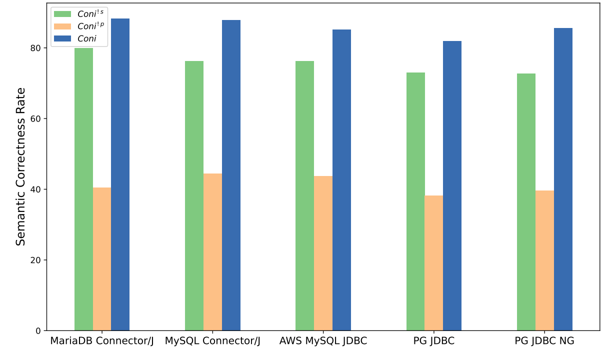


Fig. 6: The semantic correctness rate of the test cases of Coni$^{!s}$, Coni$^{!p}$, and Coni in 24 hours.

In summary, the state model and parameter generation methods play an important role in the performance of Coni. These two methods not only help Coni to increase the semantic correctness of its test cases but also enable it to trigger more bugs, which adequately answers RQ3.

VI. Discussion

State Model Automation. We manually decide the pre-state and post-state for each interface method in the state model establishment process in Figure 4. This manual confirmation is necessitated by the intricate semantics of interfaces, which challenge the formulation of generalized automated rules. However, extracting states from the specification is a one-time effort since modeling a standard specification such as JDBC can be applied to all connectors that implement that specification. To model more efficiently, the latest techniques such as Large Language Models (LLMs) can be used as

auxiliary tools to help analyze interface documentation, code comments, and method names to predict the pre and post-states, thereby automating the process [30].

Adaptability of CONI. CONI can easily adapt to a wide range of JDBC database connectors to generate effective test cases with minimal effort. Since CONI models the JDBC standard specification, it inherently provides compatibility with any database connector based on the JDBC standard. The primary task may involve minor updates to the configuration dictionaries, as different database connectors may have their own specific configurations. Differential testing is performed by comparing the results of compatible connectors separately. Many other DBMSs have multiple JDBC implementations (e.g., SQLite [31], ClickHouse [32], Oceanbase [33]), and CONI can also test these. To migrate CONI to another standard, such as ODBC, the primary task would be to redefine the state model of the ODBC interface methods.

Detecting Other Potential Bugs. In addition to identifying logic bugs, CONI can also uncover other types of bugs in connectors by altering the comparison criteria. For example, CONI can detect configuration-related bugs by comparing different configurations of the same connector. Many connectors provide a set of available connection configurations; for instance, the MySQL Connector/J documentation lists numerous configuration options, such as `rewriteBatchedStatements`. For most configurations, changing the values should not lead to inconsistent results. Furthermore, CONI can detect regression bugs by comparing different versions of the same connector. If the newer and older versions produce different results for the same command, it could indicate a potential regression error.

Objects for Comparison. The effectiveness of CONI in differential testing depends on the comparison objects used. If two connectors produce consistent results due to identical logic errors, CONI may fail to detect these errors. To mitigate this, a practical solution is to expand the range of connectors used for comparison in CONI. For instance, to thoroughly test the AWS MySQL JDBC connector, it can be compared not only with MySQL Connector/J but also with MariaDB Connector/J. This broader comparison increases the likelihood of uncovering unique errors specific to each connector.

VII. RELATED WORK

DBMS Fuzzing. Fuzzing [34] is an automated software testing technique. When applying fuzzing techniques to test DBMSs, the main challenge is to generate correct and effective SQL queries [11, 12, 13, 35, 36, 37]. For example, SQLANCER [11] generates valid SQL queries based on AST. SQUIRREL [12] employs mutation-based fuzzing on DBMSs. Griffin [13] introduces metadata graphs for SQL, providing a grammar-free method for mutating SQL test cases. LEGO [35] enhances DBMS fuzzing by generating SQL sequences with a diverse range of types. Unicorn [36] proposed the hybrid input synthesis and designed the time-series model to generate time-series queries. These works are difficult to directly adapt to testing database connectors, since they focus on SQL generation, while database connectors don't execute SQL queries. Additionally, many existing fuzzers do not use database connectors or only use a limited subset of them to send SQL queries and get results, which makes it difficult to test the deep logic of database connectors. It is necessary to apply fuzzing to database connectors to ensure the security and reliability of the interaction between databases and applications.

Test Case Generation. Many works have already explored how to generate test cases automatically [15, 16, 17, 38, 39, 40]. RANDOOP [15] generates unit tests for Java programs via a feedback-directed method. JDriver [17] utilizes dependency analysis to automate the construction of driver classes for fuzz testing, simplifying the fuzzing process and enhancing the detection of vulnerabilities. JFD [39] have further refined fuzz driver synthesis through static and value set analyses, creating an effective mechanism for Java library testing. EVOSUITE [16] applies a novel hybrid approach that generates and optimizes whole test suites to satisfy a coverage criterion. However, these methods have not adequately incorporated connector domain knowledge and also lack an effective test oracle, making it difficult to detect logic errors in database connectors effectively. CONI builds a state model and collects rich information to generate test cases. It can effectively test the deeper logic of database connectors and easily adapt to many connectors.

Database Connector Testing. Existing work primarily focuses on testing applications that use database connectors [41, 42, 43]. JDBC checker [41] employed static analysis of queries to detect potential bugs in SQL/JDBC applications. JDAMA [42] generates and executes mutated SQL queries, comparing results to assess the robustness of the SQL/JDBC application against data variations. STAF [44] is a testing framework for ODBC drivers, which requires the manual creation of a test case template in XML format beforehand, and then proceeds to generate different parameters for testing. Tools like JDBC checker and JDAMA did not focus on potential bugs within the database connection implementation. Therefore, they are unable to detect errors within database connectors. CONI addresses the lack of testing for database connectors and can effectively detect potential bugs within these connectors by generating effective test cases. Additionally, compared to STAF, CONI does not require a manual creation for test cases, saving human effort and achieving effective automated test case generation.

Model-Based Testing. Model-based testing [45] is a technique where test cases are derived from a model of the system under test. This approach ensures comprehensive coverage by systematically exploring different states and transitions within the model [46] and has been applied in various fields, such as web applications and embedded systems [47, 48, 49]. UPPAAL TRON [47] uses timed automata to model the expected behavior of the system, enabling thorough and automated verification of real-time properties. Pinheiro et al. [48] proposed a model-based testing approach that uses UML protocol state machines to generate test cases for RESTful web services. TorX [49] integrates automatic test generation,

test execution, and test analysis in an on-the-fly manner based on the ioco-test theory. However, previous methods have been ineffective in modeling database connectors. CONI uniquely models connectors to generate semantically correct test cases that trigger various state transitions, carefully managing interactions between interface methods via state-aware test case generation technique.

Differential Testing. Differential testing is an important approach in detecting logic bugs in software systems such as DBMSs [50, 51, 52, 53]. The basic idea of differential testing is to identify bugs or inconsistencies in software by comparing the outputs of different implementations of the same functionality. RAGS [50] pioneered the concept of validating SQL outputs by comparing execution results across multiple DBMS vendors. Apollo [51] takes a unique approach by comparing the execution speed of identical SQL queries on different versions of the same DBMS. This technique is particularly useful for detecting performance bugs that might arise in newer DBMS versions. Serving as a test oracle, CONI leverages differential testing to scrutinize for logic errors in database connectors. By employing semantically correct test cases, the tool detects potential logic errors when the target and reference connectors yield disparate results. This comparative analysis enhances the tool's capability to identify inconsistencies and potential issues within the database connectors.

VIII. CONCLUSION

This paper proposes CONI, a framework designed to test database connectors through state-aware test case generation. To our knowledge, it is the first framework designed specifically for database connectors. CONI mainly focuses on generating semantically correct test cases that can trigger various connector state transitions. CONI utilizes the state model to generate semantically correct interface call sequences and collects rich information to generate suitable parameter values. Our experiment results show that CONI detected many previous unknown bugs in database connectors, indicating its effectiveness in testing database connectors.

ACKNOWLEDGMENT

We appreciate the valuable comments provided by the reviewers. This research is partly sponsored by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62302256, 92167101, 62021002), and Chinese Postdoctoral Science Foundation (2023M731953).

REFERENCES

[1] "Database connection," https://en.wikipedia.org/wiki/Database_connection, 2024, accessed: August 19, 2024.

[2] "Java database connectivity," https://en.wikipedia.org/wiki/Java_Database_Connectivity, accessed: August 19, 2024.

[3] "Open database connectivity," https://en.wikipedia.org/wiki/Open_Database_Connectivity, accessed: August 19, 2024.

[4] "pgjdbc," https://github.com/pgjdbc/pgjdbc, accessed: August 19, 2024.

[5] "mariadb-connector-j," https://github.com/mariadb-corporation/mariadb-connector-j, 2024, accessed: August 19, 2024.

[6] "Binary bool values are not decoded correctly · Issue #2639 · pgjdbc/pgjdbc," https://github.com/pgjdbc/pgjdbc/issues/2639, 2022, accessed: August 19, 2024.

[7] "[conj-1071] error during bulk execution might result in connection wrong state - jira," https://jira.mariadb.org/browse/CONJ-1071, 2023, accessed: August 19, 2024.

[8] "[conj-1091] can't make a connection when the Read Replica DB is in a hang state when SocketTimeout=0 set - Jira," https://jira.mariadb.org/browse/CONJ-1091, 2023, accessed: August 19, 2024.

[9] "MySQL Bugs: #109013: useServerPrepStmts and useLocalTransactionState could cause rollback failure," https://bugs.mysql.com/bug.php?id=109013, 2022, accessed: August 19, 2024.

[10] Z.-M. Jiang, J.-J. Bai, and Z. Su, "DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation," in *32nd USENIX Security Symposium*, 2023, pp. 4949–4965.

[11] M. Rigger and Z. Su, "Testing Database Engines via Pivoted Query Synthesis," in *14th USENIX Symposium on Operating Systems Design and Implementation OSDI 20)*, 2020, pp. 667–682.

[12] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu, "Squirrel: Testing Database Management Systems with Language Validity and Coverage Feedback," in *The ACM Conference on Computer and Communications Security (CCS), 2020*, 2020.

[13] J. Fu, J. Liang, Z. Wu, M. Wang, and Y. Jiang, "Griffin: Grammar-Free DBMS Fuzzing," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[14] J. Liang, Z. Wu, J. Fu, Y. Bai, Q. Zhang, and Y. Jiang, "WingFuzz: Implementing Continuous Fuzzing for DBMSs," in *2024 USENIX Annual Technical Conference*, 2024, pp. 479–492.

[15] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 2007, pp. 75–84. [Online]. Available: https://doi.org/10.1109/ICSE.2007.37

[16] Y. Lin, J. Sun, G. Fraser, Z. Xiu, T. Liu, and J. S. Dong, "Recovering Fitness Gradients for Interprocedural Boolean Flags in Search-based Testing," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 440–451.

[17] Z. Huang and Y. Wang, "JDriver: Automatic Driver Class Generation for AFL-Based Java Fuzzing Tools," 2018.

[18] "Differential testing," https://en.wikipedia.org/wiki/Differential_testing, 2024, accessed: August 19, 2024.

[19] "mysql-connector-j," https://github.com/mysql/mysql-connector-j, 2024, accessed: August 19, 2024.

[20] "aws-mysql-jdbc," https://github.com/awslabs/aws-

mysql-jdbc, 2024, accessed: August 19, 2024.
[21] "pgjdbc-ng," https://github.com/impossibl/pgjdbc-ng, accessed: August 19, 2024.
[22] "TIOBE Index," https://www.tiobe.com/tiobe-index/, accessed: August 19, 2024.
[23] "Maven JDBC," https://mvnrepository.com/open-source/jdbc-drivers, accessed: August 19, 2024.
[24] "SQL1992," http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt, accessed: August 19, 2024.
[25] "Mysql," https://www.mysql.com/, accessed: August 19, 2024.
[26] "Mariadb," https://mariadb.org/, accessed: August 19, 2024.
[27] "Postgresql," https://www.postgresql.org/, accessed: August 19, 2024.
[28] "Aws," https://aws.amazon.com/, accessed: August 19, 2024.
[29] "JaCoCO," https://www.jacoco.org/jacoco/, accessed: August 19, 2024.
[30] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an LLM to Help With Code Understanding," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.
[31] S. Bhosale, M. T. Patil, and M. P. Patil, "Sqlite: Light database system," *Int. J. Comput. Sci. Mob. Comput*, vol. 44, no. 4, pp. 882–885, 2015.
[32] "Clickhouse," https://clickhouse.com/docs/en/interfaces/jdbc, accessed: August 19, 2024.
[33] "Oceanbase," https://en.oceanbase.com/docs/common-oceanbase-connector-j-en-10000000001092963, accessed: August 19, 2024.
[34] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Commun. ACM*, vol. 33, no. 12, Dec. 1990.
[35] J. Liang, Y. Chen, Z. Wu, J. Fu, M. Wang, Y. Jiang, X. Huang, T. Chen, J. Wang, and J. Li, "Sequence-oriented DBMS Fuzzing," in *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2023.
[36] Z. Wu, J. Liang, M. Wang, C. Zhou, and Y. Jiang, "Unicorn: Detect Runtime Errors in Time-series Databases with Hybrid Input Synthesis," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*. ACM, 2022, pp. 251–262.
[37] J. Fu, J. Liang, Z. Wu, and Y. Jiang, "Sedar: Obtaining High-Quality Seeds for DBMS Fuzzing via Cross-DBMS SQL Transfer," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.
[38] E. Albert, I. Cabanas, A. Flores-Montoya, M. Gomez-Zamalloa, and S. Gutierrez, "jPET: An Automatic Test-Case Generator for Java," 2011.
[39] Z. Chen and Y. Wang, "JFD: Automatic Java Fuzz Driver Generation," 2021.
[40] M. Chen, X. Qiu, W. Xu, L. Wang, J. Zhao, and X. Li, "UML Activity Diagram-based Automatic Test Case Generation for Java Programs," *The Computer Journal*, vol. 52, no. 5, pp. 545–556, 2009.
[41] C. Gould, Z. Su, and P. Devanbu, "JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications," in *Proceedings. 26th International Conference on Software Engineering*. IEEE, 2004, pp. 697–698.
[42] C. Zhou and P. Frankl, "Mutation Testing for Java Database Applications," pp. 396–405, 2009.
[43] Y. Shin, L. A. Williams, and T. Xie, "SQLUnitgen: Test case generation for SQL injection detection," North Carolina State University. Dept. of Computer Science, Tech. Rep., 2006.
[44] D. Ye, "Automated Testing Framework for ODBC Driver," *Journal of Software Engineering and Applications*, vol. 04, no. 12, p. 688, Dec. 2011.
[45] S. R. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based Testing in Practice," in *Proceedings of the 21st international conference on Software engineering*, 1999, pp. 285–294.
[46] M. Utting, A. Pretschner, and B. Legeard, "A Taxonomy of Model-based Testing Approaches," *Software testing, verification and reliability*, vol. 22, no. 5, pp. 297–312, 2012.
[47] "Uppaal," https://uppaal.org/, accessed: August 19, 2024.
[48] P. V. P. Pinheiro, A. T. Endo, and A. Simao, "Model-based Testing of RESTful Web Services Using UML Protocol State Machines," in *Brazilian workshop on systematic and automated software testing*. Citeseer, 2013, pp. 1–10.
[49] G. Tretmans and H. Brinksma, "Torx: Automated Model-based Testing," in *First European Conference on Model-Driven Software Engineering*, 2003, pp. 31–43.
[50] D. R. Slutz, "Massive Stochastic Testing of SQL," vol. 98, pp. 618–622, 1998.
[51] J. Jung, H. Hu, J. Arulraj, T. Kim, and W. Kang, "APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems (to appear)," in *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, Aug. 2020.
[52] Z. Cui, W. Dou, Q. Dai, J. Song, W. Wang, J. Wei, and D. Ye, "Differentially Testing Database Transactions for Fun and Profit," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.
[53] J. Liang, Z. Wu, J. Fu, M. Wang, C. Sun, and Y. Jiang, "Mozi: Discovering DBMS Bugs via Configuration-Based Equivalent Transformation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.