JDBC Checker: A Static Analysis Tool for SQL/JDBC Applications

Carl Gould, Zhendong Su, and Premkumar Devanbu
Department of Computer Science
University of California, Davis
{gould,su,devanbu}@cs.ucdavis.edu

Abstract

Many data-intensive applications dynamically construct queries in response to client requests and execute them. A familiar example is a Java servlet that creates string representations of SQL queries and then sends the queries along a JDBC connector to a database server for execution. The servlet programmer enjoys static checking via Java’s strong type system. However, the Java type system does little to check for possible errors in the dynamically generated SQL query strings. For example, a type error in a generated selection query (e.g., comparing a string attribute with an integer) can result in an SQL runtime exception. Currently, such defects must be rooted out through careful testing, or (worse) might be found by customers at runtime. In this paper, we describe JDBC Checker, a sound static analysis tool to verify the correctness of dynamically generated query strings. We have successfully applied the tool to find known and unknown defects in realistic programs using JDBC. We give a short description of our tool and discuss some experimental results in this paper.

1 Introduction

In data-intensive applications, it is quite common for the implementation code to dynamically construct database query strings and execute them. For example, a typical Java servlet web service constructs SQL query strings and dispatches them over a JDBC connector to an SQL-compliant database. In this example scenario, the servlet program generates and manipulates SQL queries as string data. We refer to the servlet code as a meta-program in the Java language [2] that manipulates object-programs in SQL.

We give a concrete example (see below) to explain our analysis technique. Suppose a programmer is developing a front-end servlet application to a JDBC-driven database back-end. The program is part of a grocery store system, and the database has a table INVENTORY, containing a list of all items in the store. This table has three columns: RETAIL, WHOLESALE, and TYPE, among others. The RETAIL and WHOLESALE columns are both of type integer, indicating their respective monetary values in cents. The TYPE column is also of type integer, representing the product type-codes of the items in the table. In the grocery store database, there is another table TYPES used to look up type-codes. This table contains the columns TYPECODE, TYPEDESC, and NAME, of the types integer, varchar (a string), and varchar, respectively.

The following example code fragment illustrates some common errors that programmers might make when programming Java servlet applications:

```java
ResultSet getPerishablePrices(String lowerBound) {
    String query = "SELECT \$\ || " + (RETAIL/100) FROM INVENTORY "+ "WHERE ";
    if (lowerBound != null) {
        query += "WHOLESALE > " + lowerBound + " AND ";
    }
    query += "TYPE IN (\" + getPerishableTypeCode() + \"");
    return statement.executeQuery(query);
}

String getPerishableTypeCode() {
    return "SELECT TYPECODE, TYPEDESC FROM TYPES "+ "WHERE NAME = 'fish' OR NAME = 'meat';";
}
```

The method getPerishablePrices constructs the string query to hold an SQL SELECT statement to return the prices of all the perishable items, and executes the query. It uses the string returned by the method getPerishableTypeCode as a sub-query. In the code, || is the concatenation operator, and the clause TYPE IN (...) checks whether the type-code TYPE matches any of the type-codes of the perishable items. If lowerBound is “595”, then the query to be executed is:

```sql
SELECT \$\ || (RETAIL/100) FROM INVENTORY
WHERE WHOLESALE > 595 AND TYPE IN
(SELECT TYPECODE, TYPEDESC FROM TYPES
WHERE NAME = 'fish' OR NAME = 'meat');
```
Several different runtime errors can arise with this example. We list them below; it is important to bear in mind that none of these errors would be caught by Java’s type system:

**Error (1).** The expression `$' || (RETAIL/100)` concatenates the character `'` with the result of the numeric expression `RETAIL/100`. While some database systems perform implicit type-casting on the numeric result (changing it to a string), many do not, and issue a runtime error.

**Error (2).** Consider the expression `WHOLESALE > lowerBound`. The variable `lowerBound` is declared as a string, and the `WHOLESALE` column is of type integer. As long as `lowerBound` is a string that only represents a number, there should not be any type errors. However, this is risky: nothing (certainly not the Java type system itself) keeps the string variable `lowerBound` from containing non-numeric characters.

**Error (3).** The string returned by the method `getPerishableTypeCode()` constitutes a sub-query that selects two columns from the table `TYPES`. Because the `IN` clause of SQL supports only sub-queries returning a single column, a runtime error would arise. This can happen if the method `getPerishableTypeCode()` did return a single column before, but was inadvertently changed to return two columns.

This specific combination of Java as the meta-language and SQL as the manipulated object-language is widely used today. The databases receiving these SQL queries certainly perform syntax and semantic checking of the queries. But because these queries are dynamically generated, any errors a database discovers are reported at runtime. It would be desirable to catch these errors statically in the source code.

In this paper, we present JDBC Checker, a static analysis tool, to flag potential errors or verify their absence in dynamically generated SQL queries. Our approach is based on an interesting combination of well-known automata-theoretic techniques [6] and a variant of the context-free language (CFL) reachability problem [7, 8]. We give an outline of our tool architecture in Figure 1. Before applying our tool, one needs to identify all the hotspots in the Java program, which are simply locations with a call to the method `executeQuery` (such as `return statement.executeQuery(query)` in our example). Our analysis consists of two main steps. In the first step, we generate a finite state automaton which conservatively approximates the set of SQL strings, using the techniques of [1]. This yields an FSA which captures a set of possible SQL strings. We then preprocess the automaton to produce a directed graph labeled with the keywords, primitives, and literals of SQL. In the second step, we apply CFL-reachability to perform semantic checking of the object-programs. One application of CFL-reachability is used to find typing context and scoping information. This is followed by a second application of CFL-reachability to perform type checking on the generated programs, treating SQL’s type system as a context-free language. Semantic errors, if found, are reported during both phases. It is worth pointing out the main difference of our analysis from a traditional analysis such as an SQL type checker. In the standard setting, a single query is analyzed at execution time, whereas we statically analyze a potentially infinite set of queries. Our analysis is sound in the sense it does not miss any errors that we are checking for, and if it does not find any errors, then it is guaranteed that such errors do not occur at runtime. Please see [3] for a more detailed description of our analysis technique and a discussion of related work.

In the rest of the paper, we report our experience in applying the tool to some realistic JDBC programs.
### Table 1: Benchmark results.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Java Program (Lines)</th>
<th>Size (Columns)</th>
<th>Analysis Time (sec)</th>
<th>Errors Found</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Automaton Generation</td>
<td>Semantic Analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Warnings</td>
<td>Total Errors</td>
</tr>
<tr>
<td>Smi</td>
<td>1359</td>
<td>1</td>
<td>35/27</td>
<td>1.5</td>
</tr>
<tr>
<td>CFWorkshop</td>
<td>36</td>
<td>5</td>
<td>47/52</td>
<td>0.6</td>
</tr>
<tr>
<td>TicTacToe</td>
<td>2888</td>
<td>2</td>
<td>134/121</td>
<td>6.4</td>
</tr>
<tr>
<td>WebBureau</td>
<td>50</td>
<td>10</td>
<td>152/162</td>
<td>0.5</td>
</tr>
<tr>
<td>Checkers</td>
<td>6615</td>
<td>4</td>
<td>181/138</td>
<td>11.8</td>
</tr>
<tr>
<td>JuegoParadis</td>
<td>6135</td>
<td>13</td>
<td>259/206</td>
<td>27.0</td>
</tr>
<tr>
<td>Reservations</td>
<td>2385</td>
<td>22</td>
<td>368/383</td>
<td>1.7</td>
</tr>
<tr>
<td>Offe/cTalk</td>
<td>5812</td>
<td>29</td>
<td>655/525</td>
<td>7.0</td>
</tr>
<tr>
<td>PurchaseOrders</td>
<td>642</td>
<td>51</td>
<td>1324/1373</td>
<td>1.3</td>
</tr>
</tbody>
</table>

2 Implementation and Results

We implemented JDBC Checker, a prototype implementation of our algorithm to detect programming errors in Java/JDBC applications. As any SQL developer will attest, every database vendor implements a different version of SQL, which makes portability of tools such as ours difficult. We have implemented our analysis for the SELECT statement specified by the grammar for Oracle version 7 [4]. This grammar is a subset of what is specified in the SQL-92 standard. Adding support for other statements or different vendors should be simple. With the goal of having a sound analysis, we have built a strict semantics into our tool: if a program is deemed type-safe by our analysis, it should be type-safe on any database system. Because the semantics of many database systems is not as strict as the one enforced by our tool, the tool may report an error which some database systems consider legitimate.

JDBC Checker is implemented in Java and uses the string analysis in [1] for computing the FSA, which in turn uses the Soot framework [10] to parse class files and compute interprocedural control-flow graphs. We have tested our tool on various code bases, including student team projects from an undergraduate software engineering class, sample code from online tutorials available on the web, and code from other projects made available to us. Table 1 lists the benchmarks and summarizes our results. For each benchmark, we list the Java program size (number of lines of source code), number of hotspots in the program, number of columns in the database schema, generated automaton size (number of edges and nodes), analysis time (split into automaton generation and semantic analysis), numbers of various warnings and errors found (cf. Table 2). Note that the benchmarks are sorted by automaton size, because it is a good measure of the complexity of the programs for our analysis. All experiments were done on a machine with a 2GHz Intel Xeon processor and 1 GB RAM, running Linux kernel 2.4.20. The results indicate that our analysis is rather precise, i.e., with low false positive rates. Because our analysis is sound, if the tool does not report any error on a program, then we have verified that the program is type-correct. In addition, although we have not tuned the performance of our implementation, the analysis is still quite efficient; it was able to analyze each of our benchmarks within a matter of minutes. We expect our analysis to scale to large systems, because analyses based on the same underlying algorithms have been shown to scale to millions of source lines of C [5, 9]. Further experiments are needed to verify our claim. Perhaps one of the most difficult obstacles is to obtain large programs on which to apply our analysis.

Table 2 shows a breakdown of the kinds of errors that we found in the benchmarks. We next explain these errors in more detail:

- **Concatenation of fields with wrong types.** This is the same error as in the concatenation `'$$'|| | (RETAIL/100)` (Section 1). After discovering this error in porting a program to a different (more strict) database, our tool has been used to find all instances of the error in the “PurchaseOrders” program.

- **Possibly unquoted string.** Assume we have a comparison such as `NAME = \alpha`, where \alpha represents an unknown string. If there are no quotes in a string that \alpha possibly represents, then such an error occurs.

- **Quoting a numerical value.** This happens when a numerical value is quoted but still treated as a numerical literal. This is a common error found in student projects. They were using MySQL, which permits numerical literals to be quoted. Many other database systems consider this an error because quoted numerical literals are of type varchar.

- **Ambiguous column selection.** Our tool detected such an error in some sample code from a tutorial website (http://web-bureau.com/modules/sql.php). This error is quite subtle, and it appears unknown. The particular statement is:

```sql
SELECT customer_id FROM customers c, orders o
WHERE c.customer_id = o.customer_id;
```

The error is that the database does not know which table’s `customer_id` to choose. Certainly, it seems not matter
<table>
<thead>
<tr>
<th>Error Kind</th>
<th>Description</th>
<th>Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Mismatch</td>
<td>Concatenation of fields with wrong types.</td>
<td>PurchaseOrders</td>
</tr>
<tr>
<td>Type Mismatch</td>
<td>Possibly unquoted string compared with a varchar column.</td>
<td>PurchaseOrders</td>
</tr>
<tr>
<td>Type Mismatch</td>
<td>Quoting a numerical value and treating it numerical.</td>
<td>Checkers, Office-Talk, TicTacToe</td>
</tr>
<tr>
<td>Semantic Error</td>
<td>Ambiguous column selection.</td>
<td>WebBureau</td>
</tr>
<tr>
<td>Semantic Error</td>
<td>Column not found.</td>
<td>Checkers, PurchaseOrders</td>
</tr>
<tr>
<td>False Error</td>
<td>Column not found (due to imprecision of the string analysis).</td>
<td>PurchaseOrders, JuegoParadis</td>
</tr>
<tr>
<td>Warning</td>
<td>Comparing a numerical value with a possibly non-numerical value.</td>
<td>PurchaseOrders, TicTacToe</td>
</tr>
</tbody>
</table>

Table 2. Breakdown of errors and warnings.

which customer_id to select in this particular statement. In general, however, with an arbitrary WHERE clause, such queries must be considered poorly constructed.

Column not found. This error happens when a column name does not exist in any of the tables in the FROM clause. We found two distinct causes of this error—one a real error and the other a false error:

Real error. The schema of the database does not include this column. This can be caused by either selecting a non-existent column, or missing the quotes around a literal, and thus being treated as a column.

False error. This is due to the imprecision in the string analysis that we use. These errors can easily be filtered out by modifying the string analysis. Please see [3] for a more detailed discussion regarding these errors.

Warning. We found one type of warning in our benchmarks. It is the same as the one illustrated in our running example in Section 1: to compare a numerical column with a possibly non-numerical value at runtime.

Demonstration

We will demonstrate the operation of the JDBC Checker with an example. We will explain and illustrate each step of the checking process, indicating the finite state automaton abstraction of the query strings and explaining with an example the use of CFL-reachability in our tool. Finally, we will illustrate the process of relating a reported error back to a defect in the original source code.

Acknowledgments

We thank Anders Møller and Aske Simon Christensen for useful discussions about this research and the following students for providing us with source code of their projects in the evaluation of our tool: Michael van der Bokke, Stanislav Kaushanskiy, Tom Ristenpart, Matt Roper, Brian Toone, Gary Wassermann, and David Xu.

References