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

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. 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 in this paper.

We use a concrete example to explain our analysis technique. Suppose a programmer is developing a front-end servlet application to an SQL-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:

```
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'";
}
```

Several different runtime errors can arise with this example (*none of these errors* would be caught by Java's type system):

(1) The expression `'$' || (RETAIL/100)` concatenates the *character* `'$'` with the result of the *numeric* expression `RETAIL/100`.

(2) In the expression `WHOLESALE > lowerBound`, `WHOLESALE` column of type integer is compared with `lowerBound`, which is declared as a string. This is risky because nothing (certainly not the Java type system itself) keeps the string variable `lowerBound` from containing non-numeric characters.

(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.

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.

Our tool, JDBC Checker, is a static analysis tool used to flag potential errors or verify their absence in dynamically generated SQL queries. Our approach combines well-known automata-theoretic techniques and a variant of the context-free language (CFL) reachability problem [3]. We give an outline of our tool architecture in Figure 1. Our analysis consists of two main steps. In the first step, we gener-

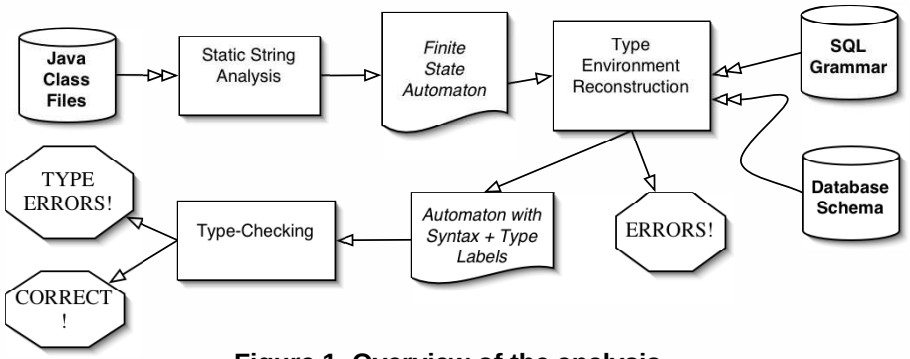

Figure 1. Overview of the analysis.

ate 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 [2] for a more detailed description of our analysis technique, experimental results, and a discussion of related work.

2. Experience

We implemented JDBC Checker, a prototype implementation of our algorithm to detect programming errors in Java/JDBC applications. We have implemented our analysis for the `SELECT` statement, using a subset of the grammar 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 [4] 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. The tool has found known and unknown defects in these programs. Our results indicate that the tool 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 test programs within a matter of minutes.

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.

References

[1] A. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Static Analysis Symposium*, pages 1–18, 2003.

[2] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the International Conference on Software Engineering*, 2004. To appear.

[3] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[4] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot–a Java optimization framework. In *Proc. IBM Centre for Advanced Studies Conference, CASCON'99*. IBM, Nov. 1999.