Command-Form Coverage for Testing Database Applications

William G.J. Halfond and Alessandro Orso
College of Computing
Georgia Institute of Technology
E-mail: {whalfond|orso}@cc.gatech.edu

Abstract

The testing of database applications poses new challenges for software engineers. In particular, it is difficult to thoroughly test the interactions between an application and its underlying database, which typically occur through dynamically-generated database commands. Because traditional code-based coverage criteria focus only on the application code, they are often inadequate in exercising these commands. To address this problem, we introduce a new test adequacy criterion that is based on coverage of the database commands generated by an application and specifically focuses on the application-database interactions. We describe the criterion, an analysis that computes the corresponding testing requirements, and an efficient technique for measuring coverage of these requirements. We also present a tool that implements our approach and a preliminary study that shows the approach's potential usefulness and feasibility.

1 Introduction

Database applications are an important component of many software systems in areas such as banking, online shopping, and health care. Because they often handle critical data, it is especially important that these applications function correctly. However, database applications have peculiar characteristics that can hinder the effectiveness of traditional testing approaches. One of these characteristics is the way interactions occur between the application and its underlying database(s). Most database applications dynamically generate commands in the database language (usually, SQL—Structured Query Language), pass these commands to the database for execution, and process the results returned by the database. Traditional code-based coverage criteria, such as statement or branch coverage, do not specifically target these generated commands. Therefore, even though they can reveal faults in the database application's code, they are often unable to reveal faults in the database commands generated by the application. Several researchers have proposed alternative criteria specifically targeted at database applications (e.g., [13, 17, 22]), but none of these approaches focuses on the coverage of dynamically-generated database commands.

To address this problem, we define a new test adequacy criterion that is specifically targeted at the interactions between an application and its database. Our criterion is based on coverage of all of the possible database command forms that the application under test can generate. Intuitively, command forms are database commands with placeholders for parts that will be supplied at runtime (e.g., through user input). To compute the set of command forms for an application, we defined a technique that builds on two previously-developed analyses [6, 12]. The technique takes as input the code of the application under test and produces a conservative approximation of the possible command forms that the application can generate. The command forms are represented as a Deterministic Finite Automaton (DFA) in which each complete path identifies a unique command form. To efficiently collect and compute coverage information, we leverage a technique for efficient path profiling by Ball and Larus [1] and apply it to the DFAs generated by our technique.

We implemented our approach in a prototype tool called DITTO (Database Interaction Testing TOol). DITTO lets developers assess the adequacy of an existing test suite with respect to application-database interactions. DITTO can also help testers generate test cases by providing feedback about which database command forms have not been exercised.

To evaluate our approach, we performed two preliminary studies on a real database application using DITTO. The first study is a proof-of-concept study that shows that our approach can be used to compute testing requirements and collect coverage information. In the second study, we assess the potential usefulness of our coverage criterion as compared to a more traditional structural coverage criterion.
The contributions of this paper are:

- A new coverage criterion for database applications that focuses on adequately exercising the interactions between an application and its underlying database.
- An efficient approach for (1) computing testing requirements, (2) instrumenting an application and collecting coverage information, (3) analyzing the coverage information and providing feedback to testers.

21st IEEE International Conference on Automated Software Engineering (ASE'06)
0-7695-2579-2/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

- The development of a tool, DITTO, that implements our approach.
- A preliminary study that shows the potential usefulness and feasibility of the criterion.

2 Background and Terminology

A *database application* is typically a multi-tiered application that interacts with one or more databases during execution. The top tier (*UI tier*) provides the user interface, the middle tier (*application tier*) implements the application's logic, and the bottom tier (*database tier*) is the database. At runtime, the application interacts with the database by generating commands in the database language and using an API to issue the commands to the database. The database executes the commands and returns the results to the application.

Because of their characteristics, database applications can be considered meta-programs that generate object programs to be executed on the database. In this case, the meta-language is the language used in the application tier—typically, one or more general purpose programming languages such as Java, C, Perl, or PHP—and the object language is usually the Structured Query Language (SQL). The meta-program creates database commands (i.e., the object program) by combining hard-coded strings that contain SQL keywords, operators, and separators with literals that can originate from the user or other sources of input. In most applications, the creation of a database command spans several statements and often involves multiple procedures. We refer to the parts of a database command that cannot be determined statically (e.g., substrings that correspond to user input) as the *indeterminate parts* of the command.

Within the meta-program, there are statements that perform API calls to issue commands to the database. Using the terminology introduced by Kapfhammer and Soffa [13], we call these statements *database interaction points*. Depending on the structure of the application and user input, a specific database interaction point can issue different types of database commands. To characterize the commands that can be generated at a database interaction point, we use the concept of database command form. A *database command form* (or simply *command form*) is an equivalence class that groups database commands that differ only in the possible value of their indeterminate parts. Intuitively, one can think of a command form as a template command string in which the parts of the database command that are statically defined by the application are specified, and the indeterminate parts are marked by a placeholder. In Section 4.1 we provide a concrete example of a database command form.

```
public ResultSet
searchBooks(String searchString, int searchType,
            boolean showRating, boolean groupByRating,
            boolean groupByISBN) {

 1. String[] searchFields = {"tiitle", "author", "isbn"};
 2. String queryStr= "SELECT title, author, description";
 3. if (showRating) {
 4.    queryStr += ", avg(rating) ";
    }
 5. queryStr += "FROM books WHERE ";
 6. if (searchType==2) {
 7.   queryStr += searchFields[searchType] + " = " +
                  searchString;
    }
 8. else {
 9.   queryStr += searchFields[searchType] + " = '" +
                  searchString + "' ";
    }
10. if (groupByRating) {
11.   queryStr += "GROUP BY rating ";
    }
12. else if (groupByISBN) {
13.   queryStr += " GROUP BY isbn ";
    }
14. return database.executeQuery(queryStr);
  }
```

Figure 1. Excerpt of database application.

3 Motivating Example

Traditional code-based coverage criteria focus on discovering errors in the application code and can result in very limited coverage of the SQL commands that an application can generate. To illustrate this limitation, Figure 1 shows a possible snippet of code from a database application. Method `searchBooks` has one database interaction point (line 14) and takes five inputs: a search string (`searchString`), an integer representing the search type (`searchType`), and a set of parameters for the search (`showRating`, `groupByRating`, and `groupByISBN`). The last four inputs determine how the hard-coded strings in the code will be combined to produce the final command. The value of the first parameter, `searchString`, is directly embedded in the database command.

This code compiles correctly, but it contains four faults that manifest themselves in the object language. Certain paths through the code generate illegal SQL commands that cause database errors and, ultimately, application failures.

1. At line 1, field "title" is misspelled as "tiitle." Because "tiitle" is not a legal column name in the table, it will cause an error if it is appended to the query at line 9.
2. If both of the appends at line 7 and line 11 are executed, there will be no space delimiter between the value of `searchString` and the "GROUP BY" clause.
3. In SQL, grouping functions such as `avg()` require a corresponding "GROUP BY" clause in the query. If `showRating` is true, but `groupByRating` and `groupByISBN` are not, this rule will be violated.
4. If the append at line 4 is not performed, there will be no space delimiter between "description" and the "FROM" clause.

IEEE COMPUTER SOCIETY

These faults manifest themselves in the generated object-program and not in the application code. Therefore, a traditional code-based adequacy criterion that requires the coverage of the application code would only detect such faults by chance. To illustrate, consider the following three test cases:

```
searchBooks("0123456789", 2, false, false, true)
searchBooks("Any Author", 1, false, false, false)
searchBooks("Any Author", 1, true, true, false)
```

These test cases achieve 100% branch (and statement) coverage of the example code, but reveal only one of the four faults in the code—the fourth one. Even using a stronger criterion, such as the all-paths criterion, could fail to expose all of the faults. A test suite could exercise all paths in the example code, but if zero is never used as a search type, the first fault would not be exposed. In the next section, we explain how our approach can provide the tester with a more effective criterion for testing interactions between applications and their underlying databases by focusing on the object program instead of the meta-program.

4 A Novel Approach for Testing Database Applications

Whereas traditional code-based adequacy criteria focus on the database application code, our approach focuses on testing the interactions between applications and underlying databases. In this sense, our approach complements existing testing criteria and ensures that database applications are more thoroughly tested. In this section, we discuss the four components of our approach: (1) a new coverage criterion for database applications, (2) a technique for computing testing requirements for the criterion, (3) a technique for efficiently collecting coverage data, and (4) a technique for analyzing and reporting coverage information.

4.1 Testing Requirements

The set of testing requirements for our criterion consists of all of the command forms for all of the database interaction points in the application under test. Because our goal is to exercise the interactions between an application and its underlying database, command forms represent a model of the database application at the right level of abstraction—they model all of the possible commands that the application can generate and execute on the database. Therefore, the number of command forms exercised by a test suite is likely to be a good indicator of the thoroughness of the testing of the interactions between the application and its database.

For our example code in Figure 1, the set of testing requirements consists of the command forms that can be executed at line 14, the only database interaction point. By looking at the different paths in the code, we can see that it can generate eighteen distinct command forms. For the sake of space, we only list one of them as an example:

```
SELECT title, author, description, avg(rating)
FROM books WHERE author = '*' GROUP BY rating
```

We use symbol $*$ as a placeholder for the indeterminate part of the command (in this simple case, the part corresponding to the value of `searchString`). All other parts of the database command, which can be determined statically, are specified in the command form.

4.2 Computing Command Forms

The main challenge when generating command forms is the accurate identification of the possible SQL commands that could be issued at a given database interaction point. Because these commands are generated at runtime and often inter-procedurally, this task requires the application of sophisticated program-analysis techniques. We perform this task in three steps.

In the *first* step, we leverage the Java String Analysis (JSA) developed by Christensen, Møller, and Schwartzbach [6]. Given a program P, a string variable[1] str, and a program point s, JSA analyzes P and computes a Non-deterministic Finite Automaton (NFA) that encodes, at the character level, all of the possible values that str can assume at s. JSA builds the NFA in a conservative way, by taking into account all string operations on str along program paths leading to s. We apply JSA to the command string variable used at each database interaction point and obtain an NFA for each string.

In the *second* step, we refine the NFAs by using a technique from our previous work [12]. This technique parses the character-level NFAs and produces corresponding SQL-level models by aggregating characters that correspond to SQL keywords and operators. Therefore, an SQL-level model is an NFA in which transitions correspond to SQL tokens (keywords, operators, and delimiters) and input placeholders, instead of single characters or character ranges (as in the original JSA models).

In the *third* step, we compute the set of command forms from the SQL-level models. We first determinize and then minimize the SQL-level models to obtain what we call an *SQL command form model*. By construction, the set of command forms for a specific database interaction point is exactly the set of all accepting paths in the command form model. To keep the number of requirements finite and avoid the need to enumerate all of the possible command forms, we adapt the efficient path profiling approach proposed by Ball and Larus [1]. Using this approach, we (1) transform any cyclic models into directed acyclic graphs and (2) assign integer *edge values* to a subset of the transitions in

[1] We use the term string to refer to all of the Java string-related classes, such as STRINGBUILDER, STRINGBUFFER, CHARACTER, and STRING.

IEEE COMPUTER SOCIETY

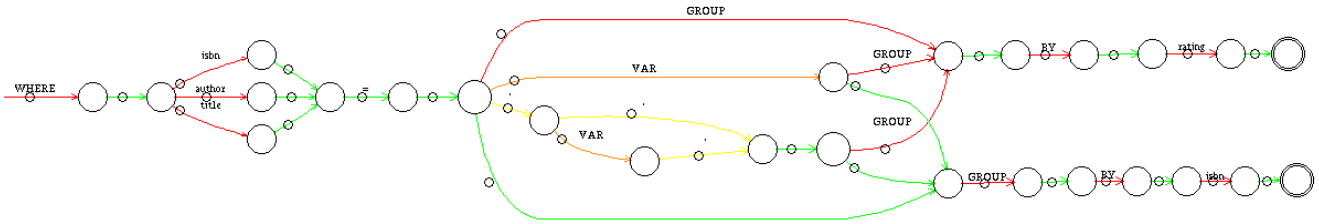

Figure 2. Excerpt of the command form model for the code in Figure 1.

the models, such that the sum of the edge values along each path is unique and the encoding is minimal. Since each command form corresponds to a unique path in the command form model, the unique integer associated with a path can be used as the ID for the corresponding command form. Moreover, because the path encoding is minimal, the largest path ID gives the total number of requirements for a database interaction point. This allows us to calculate the total number of testing requirements and assign unique IDs to requirements without having to enumerate all of the command forms.

As an example, Figure 2 shows an excerpt of the command form model for the database interaction point of the code in Figure 1. The command form shown in Section 4.1 corresponds to a specific path in this command form model.

The size of the command form model can, in the worst case, be quadratic with respect to the size of the program [6]. However, this worst case corresponds to a program that, at every statement, modifies the command string and has a branch. As Tables 1 and 2 show, the models tend to be linear with respect to the size of the application.

4.3 Coverage Collection

To measure the adequacy of a test suite with respect to our coverage criterion, we monitor the execution of the application and determine which command forms are exercised. We consider a command form associated with a database interaction point to be covered if, during execution, an SQL command that corresponds to the command form is issued at that point. An SQL command *corresponds* to a command form if they differ only in the value of the command form's indeterminate part For example, the query:

```
SELECT title, author, description, avg(rating)
FROM books WHERE author = 'Edward Bunker' GROUP
BY rating
```

would match the command form

```
SELECT title, author, description, avg(rating)
FROM books WHERE author = '*' GROUP BY rating
```

because the former can be obtained from the latter by replacing the $*$ placeholder with the string "Edward Bunker."

We collect coverage information by inserting a call to a monitor immediately before each database interaction point. At runtime, the monitor is invoked with two parameters: the string that contains the actual SQL command about to be executed and a unique identifier for the interaction point. First, the monitor parses the command string into a sequence of SQL tokens. Second, using the interaction point's identifier, it retrieves the corresponding SQL command form model. To find which command form corresponds to the command string, the monitor traverses the model by matching SQL tokens and transition labels until it reaches an accepting state. (Label $*$ can match any number of tokens.) At the end of the traversal, the path followed corresponds to the command form covered by the command string, and the ID of the command form is given by the sum of the edge values associated with the transitions in the traversed path. At this point, the monitor adds to the set of coverage data a pair consisting of the ID of the covered command form and the ID of the database interaction point.

4.4 Coverage Analysis and Reporting

Given a set of coverage data, the database command form coverage measure can be expressed as:

$$coverage = \frac{number\ of\ command\ forms\ covered}{total\ number\ of\ command\ forms}$$

The number of command forms covered is simply the number of unique entries in the coverage data. The total number of command forms is given, as discussed in Section 4.2, by the sum of each database interaction point's maximum command form ID. All command form IDs that do not appear in the coverage data correspond to command forms that were not covered during testing. Given an ID, we can easily reconstruct the string representation of the corresponding command form and show it to the testers. To do this, we use the same approach used to reconstruct paths from path IDs in Ball and Larus's profiling approach [1].

5 The DITTO Tool

To automate the use of our testing approach and enable experimentation, we designed and implemented a prototype tool called DITTO (Database Interaction Testing TOol). DITTO is implemented in Java, provides fully automated support for all aspects of our approach, and can guide the developer in testing database applications written in Java. Figure 3 provides a high-level view of DITTO's architecture. As the figure shows, DITTO has three main operating modes and consists of several modules.

IEEE COMPUTER SOCIETY

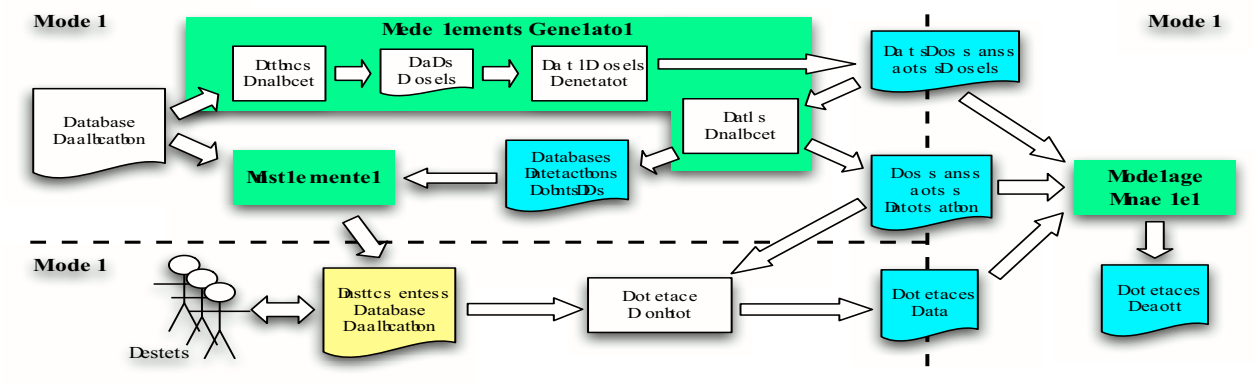

Figure 3. High-level overview of DITTO.

We expect that in a typical usage scenario DITTO would be used iteratively to support the testing process. Testers would create a set of test cases for their application or use a previously-developed test suite. Then they would use DITTO to instrument the application (Mode 1), run their test cases against the application (Mode 2), and get a coverage report (Mode 3). If testers are not satisfied with the level of coverage achieved, DITTO can provide detailed feedback about which command forms were not covered. The feedback can include both a visual display of the command form models, marked with coverage information, and a textual list of uncovered command forms. Testers can use this information to guide the development of new test cases. At this point, DITTO would be used again in Modes 2 and 3 to assess whether the additional test cases helped improved coverage. As in traditional testing, this process could continue until the testers are either satisfied with the coverage results or run out of resources.

5.1 Mode 1: Instrumentation

In Mode 1 DITTO generates the command form models and instruments the code for collecting coverage data.

To generate the command form models, DITTO statically analyzes the database application under test, as discussed in Section 4.2. For each database interaction point, the *String Analyzer* uses the JSA library [6] to produce an NFA model of the SQL command string used at that point. The *SQL-Model Generator* uses a modified version of our AMNESIA tool [12] to process the NFA models and generate the corresponding SQL command form models. Finally, the *Path Analyzer* takes as input the SQL command form models, annotates them with the edge values for the path encoding, and generates some command form information used for bookkeeping.

To produce coverage data at runtime, the *Instrumenter* modifies the code as described in Section 4.3. The *Instrumenter* inserts a call to the *Coverage Monitor* immediately before each database interaction point. The call to the monitor provides as parameters (1) the string variable that contains the SQL command about to be executed and (2) the unique identifier for the database interaction point. The instrumentation is performed using bytecode rewriting and leverages the Byte Code Engineering Library (BCEL – `http://jakarta.apache.org/bcel/`). For our example application, the *Instrumenter* would modify the database interaction point at line 14 (Figure 1) as follows:

```
...
monitor.log(<interaction point ID>,queryStr);
return database.executeQuery(queryStr);
...
```

5.2 Mode 2: Execution

In Mode 2 DITTO collects coverage data and records it for later analysis. The instrumented database application executes normally until it reaches a database interaction point. At this point, the string that is about to be submitted as an SQL command is sent to the *Coverage Monitor* together with the interaction point's ID. The monitor traverses the command form model for that interaction point, as described in Section 4.3, and logs the pair consisting of the ID of the covered command form and the ID of the database interaction point.

5.3 Mode 3: Analysis and Reporting

In Mode 3 DITTO computes the command form coverage measure and provides feedback to the testers. The *Coverage Analyzer* uses the coverage data collected in Mode 2 and calculates the coverage as described in Section 4.4. The test adequacy score alone does not give testers any information about which parts of the code were insufficiently exercised. To provide more detailed feedback, DITTO also allows testers to visually examine the command form models and see which paths were not covered by their tests. This information is visualized by coloring and annotating covered paths in the models. The testers can also list the command forms that were not covered in the model in textual format. Both of these feedback mechanisms provide testers with an intuitive way to understand coverage results and can guide further test-case development.

IEEE COMPUTER SOCIETY

6 Current Limitations

Stored procedures. A common development practice is to encapsulate sequences of SQL commands and save them in the database as stored procedures. Developers can then issue a SQL command that invokes the stored procedure, possibly with some input parameters. Our approach models calls to stored procedures just like any other command issued to the database, but does not consider the SQL commands within a stored procedure (as they are stored in the database and not explicitly generated by the application). In other words, our current approach treats stored procedures as atomic instructions. If needed, the coverage criterion could be expanded to include the contents of stored procedures.

External fragments. In some applications, developers input constant strings from external sources, such as files, and use these strings to build SQL commands. These external strings are typically SQL command fragments that contain SQL keywords and operators (in contrast with user input, which typically consists of string or numeric literals). This situation does not cause any conceptual problem for our approach, as an indeterminate part of a command form can match the tokens that correspond to external fragments. From a practical standpoint, however, simply considering external fragments as indeterminate parts may decrease the effectiveness of the criterion. (For an extreme example, consider the case in which all SQL commands are simply read from external files.) This limitation is mostly implementation related: we could extend our technique so that developers can specify which external fragments are used in their application, and the technique would account for these fragments when building the SQL command form model. We have not implemented this solution yet because none of the applications that we have examined so far uses external fragments.

Infeasibility. Infeasibility is one of the main problems for structural coverage criteria. Computing structural coverage requirements for an application typically involves some form of static analysis of the application's code. In general, because determining the reachability of a statement for a given program is an undecidable problem [20], static analysis tends to generate spurious requirements that cannot be satisfied. The presence of unsatisfiable requirements in a criterion makes it impossible to reach 100% coverage for that criterion and limits its usefulness. Infeasibility can affect the command form criterion by causing the presence of spurious command forms that do not correspond to any command that could be generated by the application. Intuitively, this problem should occur primarily because the string analysis may add to the model strings that are generated along infeasible paths. Therefore, we expect the infeasibility problem to affect us to a similar extent in which it affects path-based coverage criteria. As discussed in Section 9, we plan to investigate infeasibility issues for our criterion through empirical evaluation.

Servlet	*LOC*	*# Methods*
Header	130	9
AdvSearch	253	13
Default	693	26
CategoriesGrid	309	18
CardTypesGrid	270	17
OrdersRecord	463	20
MembersInfo	488	21
CardTypesRecord	368	18
Footer	129	9
Login	290	14
EditorialCatGrid	310	18
EditorialsGrid	325	18
ShoppingCartRecord	412	19
Registration	515	20
CategoriesRecord	368	18
EditorialsRecord	441	19
Books	534	22
EditorialCatRecord	365	18
MembersRecord	618	22
BookMaint	514	21
MyInfo	649	19
BookDetail	921	25
AdminBooks	609	22
OrdersGrid	602	20
ShoppingCart	705	21
AdminMenu	429	11
MembersGrid	578	20

Table 1. Summary information about Bookstore's servlets.

Analysis limitations. Our approach relies on the ability of the underlying string analysis to build the initial NFA models for the database interaction points. Imprecision (i.e., over-approximation) in the string analysis could limit the effectiveness of our criterion. For example, a worst case scenario in which the analysis generates an automaton that accepts any strings would result in command-form models that are covered by any test case that reaches the corresponding database interaction point. Note that manual inspections showed that imprecision was not an issue for any of the models that we generated in our evaluation.

7 Evaluation

In our evaluation, we performed two studies. The first one is a proof of concept study in which we used DITTO on a real database application to assess whether it was able to successfully generate test requirements and measure coverage. The second study explores the effectiveness of traditional coverage criteria in generating test suites that are adequate with respect to command-form coverage.

IEEE COMPUTER SOCIETY

Servlet	*# DIP*	*# Edges*	*# States*	*% Cov.*
Header	0	0	0	N/A
AdvSearch	1	16	17	N/A
Default	1	406	407	13%
CategoriesGrid	1	48	49	N/A
CardTypesGrid	1	48	49	N/A
OrdersRecord	2	371	259	<1%
MembersInfo	2	201	172	7%
CardTypesRecord	2	191	143	2%
Footer	1	1	2	N/A
Login	1	67	58	4%
EditorialCatGrid	1	48	49	N/A
EditorialsGrid	1	157	158	N/A
ShoppingCartRecord	2	296	210	N/A
Registration	3	478	309	<1%
CategoriesRecord	2	191	143	5%
EditorialsRecord	2	527	345	<2%
Books	1	7928	6267	N/A
EditorialCatRecord	2	191	143	2%
MembersRecord	3	1282	772	<1%
BookMaint	2	1163	681	<1%
MyInfo	2	588	354	N/A
BookDetail	4	1211	854	1%
AdminBooks	1	344	258	N/A
OrdersGrid	1	326	265	N/A
ShoppingCart	2	154	140	N/A
AdminMenu	1	1	2	N/A
MembersGrid	1	235	207	N/A

Table 2. Information on the SQL command form models for Bookstore.

For both studies, we used a database application called Bookstore (available at `http://www.gotocode.com`). Bookstore implements an online bookstore and uses Java servlets to implement the UI and application tiers. Table 1 shows summary information about each of the servlets in the application. For each servlet (*Servlet*), the table shows its size (*LOC*) and its number of methods (*# Methods*).

7.1 Study 1

The first study provides a proof of concept evaluation of DITTO by showing that it can work on a specific application. To achieve this goal, we used DITTO on Bookstore to generate testing requirements and measure command form coverage for a set of test cases. DITTO successfully computed the test requirements for each of the database interaction points and instrumented all of the servlets. The entire process of extracting the models took less than five minutes on a Pentium III machine with 1GB of memory running the GNU/Linux Operating System. We then deployed the instrumented servlets and ran a previously developed test suite against them.

Table 2 summarizes the results of the study. For each servlet, the table shows the number of database interaction points it contained (*#DIP*), the total number of states and transitions in the models (*#States* and *#Edges*), and the percentage of command-form coverage achieved during testing (*%Cov.*). Some of the servlets were not exercised by the test suite, and their coverage measure is reported as "N/A."

The test suite that we used in this study was developed in previous work [12] (and also used in related work [16]) to target specific security issues. It was not developed to achieve coverage, and we did not try to improve it because the goal of this study was not to test the subject application, but to demonstrate a successful use of DITTO. Even under these premises, the results provide some initial evidence that command-form coverage cannot be trivially achieved, and that specialized test cases may be needed to suitably exercise the interactions between applications and their underlying databases.

7.2 Study 2

The second study addresses the research question: *Does command-form coverage provide for a more thorough testing of database applications than alternative traditional approaches?* For this study, we selected branch coverage as the representative traditional criterion because it is widely used. A typical way to address this question would be to (1) create a number of branch-adequate test suites, (2) create the same number of adequate test suites for the command-form coverage criterion, (3) run both sets of test suites on several versions of an application with seeded faults, and (4) compare the fault-detection capability of both sets of test suites.

However, there is a significant technical challenge that complicates this type of evaluation: the lack of an effective way to automatically seed different types of SQL-related errors. Whereas there are mutant generation tools that can be used to seed traditional faults in programs, there are no such tools for SQL-related faults. Seeding the errors by hand or building an ad-hoc tool are less than ideal options because they would introduce problems of bias. Alternatively, collecting real database-command related faults from open-source projects would be a good solution, but may involve an extensive search and still result in too few data points to draw significant conclusions.

Due to these issues, we decided to use an indirect and approximated method to compare the effectiveness of our criterion with the effectiveness of the traditional branch-coverage criterion. The method that we use is to compute an upper bound to the number of command forms that could be exercised by a branch-adequate test suite and compare this number to the total number of command forms for the application. A higher total number of command forms would be an indication that branch coverage (and possibly other traditional testing criteria) may not adequately test interactions between the application and the database, and that command-form coverage may be needed. For instance, consider the example code in Figure 1. As discussed in Sec-

IEEE COMPUTER SOCIETY

tion 3, we could achieve 100% branch coverage of that code with just three test cases, each of which would exercise only one command form. Because there are eighteen possible command forms, it is clear that the considered branch-adequate test suite would not thoroughly exercise the SQL commands generated by the application.

The total number of command forms for an application is computed by DITTO, as discussed in Section 4.2. To calculate an upper bound to the number of command forms that would be executed in a servlet by a branch-adequate test suite, we use the cyclomatic complexity of the servlet. The cyclomatic complexity is an upper bound to the minimal number of test cases needed to achieve 100% branch coverage of a program [19]. An analysis of the servlets used in the study revealed that no test case can execute a database interaction point more than once (i.e., no database interaction point is in the body of a loop). Therefore, if we conservatively assume that each test case exercises a different command form, we can use the cyclomatic complexity as an upper bound to the number of possible command forms that a minimal, branch-adequate test suite would execute. In practice, however, such an assumption could vastly overestimate the number of command forms exercised by a branch-adequate test suite because many paths in the code do not actually generate a database command. To obtain a better estimate, we must compute the cyclomatic complexity on only the subset of the servlet code that is involved with creating, modifying, and executing database commands. We thus generate an executable backward slice [18] for each command string variable at each database interaction point using JABA[2] and compute the cyclomatic complexity only for the subset of the servlet in the slice. Because the JABA-based slicer that we use is still a prototype and requires a considerable amount of human intervention, in the study we consider only a subset of the Bookstore servlets.

The results of our analysis are shown in Table 3. For each servlet considered, we report the number of database interaction points (*#DIP*), the number of command forms (*# Command Forms*), and the cyclomatic complexity of the servlet's slice. As the data shows, the number of command forms is considerably higher than the cyclomatic complexity in several cases, and the average number of command forms per database interaction point (253) is almost five times the average cyclomatic complexity (57). Because the numbers we used in the study are estimates, and we only considered a small number of servlets, we cannot draw any definitive conclusion from the study. Nevertheless, this preliminary study indicates that command-form coverage may result in a more thorough testing of database interactions than traditional coverage criteria. These results encourage further research and a more extensive empirical evaluation.

[2] http://www.cc.gatech.edu/aristotle/Tools/jaba.html

Servlet	*# DIP*	*# Command Forms*	*Cyclomatic Complexity*
MyInfo	1	6	136
BookDetail	4	1583	150
AdminBooks	1	617	31
OrdersGrid	1	394	26
ShoppingCart	2	20	28
AdminMenu	1	1	6
MembersGrid	1	162	21

Table 3. Results of the evaluation.

8 Related Work

The problem of ensuring the correctness of database applications has been approached in several different ways. The approaches most closely related to ours are those that also propose new test adequacy criteria for database applications. Within this group, there are two types of criteria, those that focus on data-flow and those that focus on the structure of the SQL commands sent to the database. Suárez-Cabal and Tuya [17] propose a structural coverage criterion that requires the coverage of all of the conditions in a SQL command's "FROM," "WHERE," and "JOIN" clauses. This criterion is analogous to the multiple condition coverage criterion [5], but applied to SQL clauses instead of code predicates. This work differs from ours in that it focuses on SQL commands that are completely statically defined and only considers coverage of a subset of the SQL language, namely, conditions in queries' clauses. In contrast, our technique considers coverage of all types of SQL commands, including dynamically-constructed ones. Also similar to our criterion are the criteria proposed by Willmor and Embury [22]. In particular, they propose the *all database operations* criterion, which requires the execution of all of a program's database interaction points. Our proposed criterion subsumes this criterion because it requires not only the execution of each database interaction point, but also the coverage of all of the command forms that can be generated at that point.

Kapfhammer and Soffa [13] propose a set of data-flow test adequacy criteria based on definition-use coverage of database entities. These entities can be defined at different levels of granularity that include the database, relation, attribute, record, and attribute-value level. These criteria parallel conventional data-flow testing criteria [10], but are defined and evaluated based on database entities instead of program variables. Willmor and Embury [22] refine these criteria and expand them to accommodate database transactions. Both approaches differ from ours in that they focus on covering all of the definitions and uses of database entities instead of the different command forms. The data-flow criteria do not subsume command form coverage because at a database interaction point it is possible to have several command forms that exercise the same set of database en-

IEEE COMPUTER SOCIETY

tities. In this case, satisfying the data-flow criteria would not satisfy command form coverage. The data flow criteria are complementary to ours; they target faults related to the definition and use of database entities, whereas our approach targets errors in the database commands generated by a database application.

The work of Chan and Cheung [2] is similar to ours in that it aims to thoroughly test database applications by taking into account the generated SQL commands. Their approach translates SQL commands into Relational Algebra Expressions (RAE), converts the RAE into the meta-language of the application, and replaces the SQL command in the application with the generated code. After the transformation, they use standard white-box testing criteria to test, albeit indirectly, the SQL commands. To illustrate their approach, consider the case in which the developer issues a SQL `JOIN` command. They would first convert the SQL command into equivalent statements in the meta-language. In this case, the `JOIN` would be translated into two nested `for` loops (the `JOIN` command is similar to a cross product between two database tables). Testers would then create test cases to properly exercise the additional `for` loops in the code. (Using our approach, the `JOIN` command would simply be counted as an additional command form to be covered.) Chan and Cheung's approach enforces a thorough testing of database applications, but it has several limitations when compared to our approach. First, and most importantly, the translation of the SQL commands into RAE requires that the SQL commands be statically defined as constant strings. This is a fundamental limitation because it precludes the usage of the technique on the many database applications that build command strings by appending different substrings along non-trivial control-flow paths. Our approach does not have this problem because the static analysis can typically account for all possible commands, including dynamically-constructed ones. Another limitation is that the RAE is less expressive than SQL, so certain SQL commands cannot be translated and will not be adequately tested. We are not affected by this issue because we measure coverage directly on the database command forms.

Another proposed approach is to perform static verification of the possible SQL commands. Christensen, Møller, and Schwartzbach introduce the Java String Analysis (JSA) [6] and use it to extract non-deterministic finite automata that represent the potential SQL commands that could be generated at a given database interaction point. They then intersect the automata with a regular language approximation of SQL to determine if the commands are syntactically correct. Gould, Su, and Devanbu propose JDBC Checker [11], which builds on JSA and adds type analysis to statically verify that dynamically generated commands are type-safe. This type of verification is powerful, but does not necessarily eliminate the need for testing. First, it is not always possible to check SQL commands statically (e.g., in cases where the application allows keywords or operators to be specified at runtime). Second, there are limitations in the type of errors that can be detected by these techniques. For instance, consider the third error in our example from Section 3. This type of fault would not be detected by Christensen, Møller, and Schwartzbach's approach because their syntax checking is not expressive enough to represent the constraints violated by the error. Although our approach uses similar models as these techniques, it uses them for measuring the thoroughness of a test suite with respect to command forms instead of for verifying them. Our technique, although less complete than the ones based on static verification, does not have the limitations of these techniques and may reveal faults that these techniques cannot reveal.

Other approaches, such as SQL DOM [14] and Safe Query Objects [7], propose to change the way developers construct SQL commands. Instead of having developers create SQL commands using string concatenation, they offer a specialized API that handles all aspects of creating and issuing SQL command strings. The main benefit of these approaches is that they can enforce a more disciplined usage of SQL, and thus prevent many errors. However, these approaches require developers to learn a new API and development paradigm and, most importantly, cannot be easily applied to legacy code.

Finally, other related work focuses on test case generation for database applications and regression testing. Although related to our approach, they have different goals and are mostly orthogonal to our work. AGENDA [3, 4, 9] is a framework for automatic generation and execution of unit tests for database applications that is loosely based on the category partition testing method [15]. AGENDA takes as input information about the logical database model (e.g., schema information, database states, and logical constraints) and combines it with tester input to generate test cases for the database. Zhang, Xu, and Cheung propose a technique for generating database instances to support testing [24]. The technique uses a constraint solver to identify which values a database should contain to ensure that the different conditions and predicates in an application's SQL commands will be exercised. Similarly, Willmor and Embury [23] propose a mechanism that allows developers to specify database states that are relevant for a test suite and can then appropriately populate the database. Daou and colleagues use a firewall-based approach for regression testing of database applications [8], while Willmor and Embury propose regression testing based on definition-use analysis of the SQL commands in an application [21].

9 Conclusion

In this paper, we addressed a common problem that arises when testing database applications: how to adequately test

IEEE COMPUTER SOCIETY

the interactions between an application and its underlying database. To address this problem, we introduced an approach based on a new test adequacy criterion called command form coverage. This criterion requires the coverage of all of the command forms that a given application can issue to its database.

We also presented DITTO, a prototype tool that implements our approach. DITTO generates testing requirements for our criterion, measures the adequacy of a test suite with respect to the criterion, and provides feedback to testers about which requirements were not covered during testing.

Finally, we presented two preliminary studies. The first one is a feasibility study that shows that DITTO can successfully extract testing requirements and measure coverage for a real database application. The second study provides analytical evidence that traditional code-based testing criteria may be inadequate in the case of database applications. The results of the studies, although preliminary, are encouraging and motivate further research.

There are several possible directions for future work. *First*, we will perform a more extensive empirical evaluation of our approach. We will identify additional subjects and fault information for these subjects by performing a survey of existing database applications. We will then use these subjects to (1) assess the effectiveness of our criterion in revealing database-application-specific errors, (2) further compare our criterion and traditional code-based criteria, and (3) study infeasibility and other analysis-related issues for our approach. *Second*, we will investigate whether we can improve the effectiveness of our approach by leveraging information about the database used by the application under test (e.g., the database schema). *Finally*, we will investigate the application of our technique to other domains, such as dynamic web applications.

Acknowledgments

This work was partially supported by NSF awards CCR-0205422 and CCR-0306372 to Georgia Tech and by the Department of Homeland Security and US Air Force under Contract No. FA8750-05-C-0179.

References

[1] T. Ball and J. R. Larus. Efficient Path Profiling. In *Proc. of Micro 96*, pages 46–57, Dec. 1996.

[2] M. Chan and S. Cheung. Testing Database Applications with SQL Semantics. In *CODAS'99: Proc. of 2nd International Symposium on Cooperative Database Systems for Advanced Applications*, pages 363–374, Dec. 1999.

[3] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weyuker. A Framework for Testing Database Applications. In *ISSTA '00: Proc. of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 147–157, Aug. 2000.

[4] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker An AGENDA for Testing Relational Database Applications. *Journal of Software Testing, Verification and Reliability*, Mar. 2004.

[5] J. Chilenski and S. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*, 9(5):193–200, Sep. 1994.

[6] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, pages 1–18, Jun. 2003.

[7] W. R. Cook and S. Rai. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. In *Proc. of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 97–106, May 2005.

[8] B. Daou, R. A. Haraty, and N. Mansour. Regression Testing of Database Applications. In *SAC '01: Proc. of the 2001 ACM Symposium on Applied Computing*, pages 285–289, 2001.

[9] Y. Deng, P. Frankl, and D. Chays. Testing Database Transactions with AGENDA. In *ICSE '05: Proc. of the 27th International Conference on Software Engineering*, pages 78–87, 2005.

[10] P. G. Frankl and E. J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, Oct. 1988.

[11] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *Proc. of the 26th International Conference on Software Engineering (ICSE 04)*, pages 645–654, May 2004.

[12] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proc. of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 174–183, Nov. 2005.

[13] G. M. Kapfhammer and M. L. Soffa. A Family of Test Adequacy Criteria for Database-Driven Applications. In *ESEC/FSE-11: Proc. of the 9th European Software Engineering Conference, held jointly with, 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 98–107, Sep. 2003.

[14] R. McClure and I. Krüger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In *Proc. of the 27th International Conference on Software Engineering (ICSE 05)*, pages 88–96, May 2005.

[15] T. J. Ostrand and M. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, 31(6), Jun. 1988.

[16] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *The 33rd Annual Symposium on Principles of Programming Languages*, pages 372–382, Jan. 2006.

[17] M. J. Suárez-Cabal and J. Tuya. Using an SQL Coverage Measurement for Testing Database Applications. In *Proc. of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 253–262, Oct. 2004.

[18] F. Tip. A Survey of Program Slicing Techniques. *Journal Of Programming Languages*, 31(5):32–40, May 1998.

[19] A. H. Watson and T. J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. Technical Report 500-235, NIST Special Publication, Aug. 1996.

[20] E. J. Weyuker. The Applicability of Program Schema Results to Programs. *International Journal of Parallel Programming*, 8(5):387–403, Oct. 1979.

[21] D. Willmor and S. M. Embury. A safe regression test selection technique for database-driven applications. In *Proc. of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 421–430, Sep. 2005.

[22] D. Willmor and S. M. Embury. Exploring test adequacy for database systems. In *Proceedings of the 3rd UK Software Testing Research Workshop (UKTest 2005)*, pages 123–133, Sep. 2005.

[23] D. Willmor and S. M. Embury. An Intensional Approach to the Specification of Test Cases for Database Systems. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, pages 102–111, May 2006.

[24] J. Zhang, C. Xu, and S. C. Cheung. Automatic Generation of Database Instances for White-box Testing. In *COMPSAC '01: Proc. of the 25th International Computer Software and Applications Conference*, pages 161–165, Oct. 2001.

IEEE COMPUTER SOCIETY