MODA: Automated Test Generation for Database Applications via Mock Objects

Kunal Taneja[1], Yi Zhang[2], Tao Xie[1]
[1]Department of Computer Science, North Carolina State University, Raleigh, NC, 27695, USA
[2]Center for Devices and Radiological Health, US Food and Drug Administration, Silver Spring, MD, 20993, USA
[1]{ktaneja, txie}@ncsu.edu, [2]Yi.Zhang2@fda.hhs.gov

ABSTRACT

Software testing has been commonly used in assuring the quality of database applications. It is often prohibitively expensive to manually write quality tests for complex database applications. Automated test generation techniques, such as Dynamic Symbolic Execution (DSE), have been proposed to reduce human efforts in testing database applications. However, such techniques have two major limitations: (1) they assume that the database that the application under test interacts with is accessible, which may not always be true; and (2) they usually cannot create necessary database states as a part of the generated tests.

To address the preceding limitations, we propose an approach that applies DSE to generate tests for a database application. Instead of using the actual database that the application interacts with, our approach produces and uses a mock database in test generation. A mock database mimics the behavior of an actual database by performing identical database operations on itself. We conducted two empirical evaluations on both a medical device and an open source software system to demonstrate that our approach can generate, without producing false warnings, tests with higher code coverage than conventional DSE-based techniques.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Languages, Experimentation

Keywords

Test Generation, Database Application, Mock Object

1. INTRODUCTION

Database applications, i.e., programs interacting with database back-ends, play an increasingly important role in mission-critical systems that require reliable data storage and correct data access. Since a failure of mission-critical systems can cause disastrous consequences, it becomes critical to guarantee the deployment of correct and sound database applications in these systems.

Software (unit) testing has been widely used by developers to assess the quality of database applications. Given that manual test generation can be labor-intensive, developers usually rely upon automatic techniques [5, 7] to create quality tests, i.e., tests that exercise comprehensive behaviors of the program under test. Unfortunately, such automatic techniques face two significant challenges when applied to database applications.

First, automatic test generation usually requires executing database applications to collect necessary information. Executing a database application, however, may alter the data stored in its database back-end, which may not always be desirable for data privacy and preservation reasons. Using a local test database, instead of the original one, may address this issue. However, it is not always an easy task to set up such a local database and clone configurations of the original database onto it.

Second, no matter what database (original or test) is used, a test generation tool has to bridge the semantic gap between database applications and their database back-ends. That is, to comprehensively cover a database application, tests produced for it should not only provide inputs to the application itself, but also prepare necessary states in its database back-end.

Mocking techniques [6], and their generalized variant called Parameterized Mock Objects (PMO) [10], can be used to alleviate the first challenge. Such techniques replace an actual database with a mock object, called mock database, which responds to actual database queries with either default values (as in mocking techniques) or values customized to explore program paths in the application under test (as in PMO).

However, using mock databases is not a silver bullet. A mock database usually does not reflect the genuine behavior of the actual database that it simulates. Instead, it simply responds to a database query with customized values, even if the query is not satisfiable or the values are unrealistic. Moreover, a mock database is incapable of simulating how an actual database changes its states during system execution. Consequently, running tests with a mock database may result in unexpected results or false warnings being generated (i.e., tests fail in test execution but should not in reality; see Section 3 for an example).

In this paper, we address the preceding challenges and present an automatic test generation approach for database applications. Given a database application and the schema defining its database back-end, our approach produces a *parameterized* mock database that complies with the schema and, at the same time, captures the behavior of the database back-end. In particular, such a mock database replicates the effect of operations over an actual database by performing the same operations on itself, so that the state of the

ASE'10, September 20–24, 2010, Antwerp, Belgium.

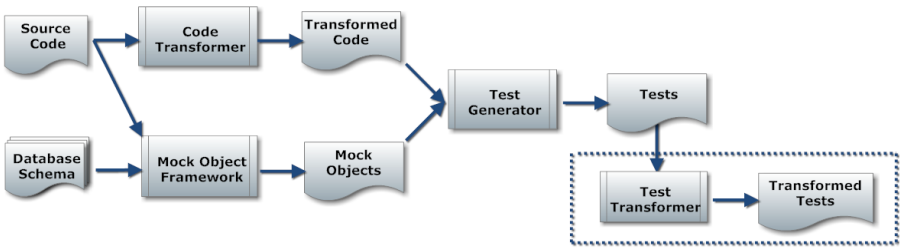

Figure 1: Architectural Overview of MODA

actual database is faithfully mirrored and only feasible values are returned for database queries.

Having a mock database produced, our approach uses Dynamic Symbolic Execution (DSE) [5, 7] to generate inputs for testing the target database application. DSE runs the application, using the mock database, with both random and symbolic inputs. The execution of the application with symbolic inputs, as well as the subsequent solving of path constraints collected during symbolic execution, helps to explore program paths in the application that are not covered by random inputs.

We have implemented a prototype tool, called MODA (Mock Object Based Test Generation for Database Applications), for our approach. In its current implementation, MODA incorporates Pex [9], a tool from Microsoft Research, as its DSE engine. To make Pex work with a mock database, our tool equips it with two capabilities: (1) a code transformation feature that redirects the application under test to interact with the mock database; and (2) a feature to inject records into the mock database, so that preparing database states as a part of tests becomes possible.

We have conducted two empirical evaluations to assess the effectiveness of MODA: one on a real-world medical device and the other on an open source software system. Results of the evaluations demonstrate that our approach can achieve higher code coverage, with no false warnings, than conventional DSE-based techniques.

2. APPROACH

Given a database application and the schema of its database back-end, MODA coordinates four of its components, namely *Mock Object Framework*, *Code Transformer*, *Test Generator*, and *Test Transformer*, to produce a test suite that systematically exercises the target application.

Figure 1 presents an architectural view of MODA, where the *Mock Object Framework* component establishes a mock database that simulates the database back-end and the *Code Transformer* component pre-processes the target application such that all of its database interactions are replaced by interactions with the mock database. The *Test Generator* component of MODA serves as the engine for generating tests for any user-selected method in the target application, while the *Test Transformer* component post-processes the generated tests to make them runnable with the target application.

2.1 Mock Object Framework

From a database application's point of view, its database back-end accomplishes the following functions: organizes and maintains data in a specified structure; provides interfaces through which the application can interact with it; and responds to queries issued from the application by performing appropriate operations over the stored data. A mock database intending to faithfully mimic an actual database should be able to replicate all these functions. This requirement motivates the design of the *Mock Object Framework* component in our approach.

Mocking Database. The component first parses the input schema[1] in order to obtain the information about how the original database is structured. The input schema typically prescribes the set of tables T defined in the original database, every column C_i of each table $t_i \in T$, and constraints over C_i. Based on the parsed information, the component creates and initializes tables in a mock database DS, so that DS shares the same data structure as the original database.

Mocking Database Interfaces. Database applications typically interact with database back-ends through standardized interfaces. To make an application under test *talk* to a mock database, it is essential to equip the mock database with the same set of interfaces as a standard database offers. As currently implemented, the *Mock Object Framework* component creates a mocking counterpart for each API class and method defined in the MySQL library [8], especially query execution APIs that forward SQL queries to a database for execution. Given a query execution API, MODA produces its mocking version to do two things: intercept the type and parameter information of the query forwarded by the API, and execute the corresponding mock operations on the mock database (see below on mocking database operations) with the same parameters. For other APIs in the MySQL library, such as those that facilitate the establishment of database connections or the creation of SQL commands, MODA either trivially mocks them as empty methods or replicates their functionalities according to their specifications.

Mocking Database Operations. When mocking actual databases, conventional mocking or PMO techniques suffer from the fact that they can neither interpret SQL queries nor simulate how a database changes its states over program execution. As a result, these techniques may produce infeasible tests that are typically difficult for developers to filter out.

MODA addresses the preceding issue by performing the same operations over the mock database as the application under test instructs its database back-end to do, so that the effects of such operations over the original database are faithfully reflected in the mock database. Typically, a database application requests its database to manipulate the stored data by issuing four types of queries, namely $SELECT$, $INSERT$, $UPDATE$, and $DELETE$. For each type of query, the *Mock Object Framework* component includes a mocking algorithm to simulate its effects.

As an example, consider mocking the $INSERT$ query. Formally, MODA translates an $INSERT$ query into the format of $M_I(T, C, V)$, where T defines the table that the query targets, C is a subset of columns in T, V=$\{v_1, v_2, ..., v_n\}$ are the list of values defined in the $VALUE$ clause of the query, and $|C| = |V|$. The component mocks an $INSERT$ operation in a straightforward manner: the table with the name T is first fetched from DS; a new record ρ is then created with values $v_1, v_2, ..., v_n \in V$ for columns $c_1, c_2, ..., c_n \in C$; and finally ρ is added into T.

[1]In this paper, we consider only a subset of the standard relational database schemas [13].

2.2 Code Transformer

Given a target database application P, the *Code Transformer* component redirects it to interact with the mock database created by the *Mock Object Framework*. Specifically, the component transforms P into P_t, such that: (1) if P refers to an API class A, then this reference is replaced in P_t by the reference to its corresponding mock class M_A; and (2) if P invokes a database API method O, then this invocation is replaced in P_t with an invocation of the mock method M_O that the *Mock Object Framework* component creates for O.

As a result, whenever P issues an SQL query by calling a certain database API, P_t calls the corresponding mock API and makes the same query to the mock database.

2.3 Test Generator

MODA currently incorporates Pex to generate tests for the transformed database application P_t. Pex performs both concrete and symbolic execution on P_t simultaneously, where concrete execution is used to obtain an instance program path p in P_t and symbolic execution over p collects path constraints along p. By altering the collected constraints, Pex is able to acquire a condition that represents a new path p' deviated from p. Generating tests that cover p' is then accomplished by solving the acquired condition. Pex iterates this process until all paths within P_t have been explored or a predefined threshold of cost is reached.

Based upon Pex, the *Test Generator* component generates tests, each of which is a combination of program inputs and an initial state of the mock database, to cover program paths in the database application. Specifically, the component first turns records[2] in the mock database and their attributes into symbolic values. These symbolic values, together with the symbolic values representing program inputs, are then passed to Pex for collecting and solving path constraints. Solutions to all path constraints collected by Pex constitute the final test suite for the database application. In order to synthesize initial states of the mock database, the *Test Generator* component also takes the responsibility of inserting records generated by Pex into the mock database.

2.4 Test Transformer

It is sometimes desirable to execute the generated tests on the target database application in its original format (i.e., before code transformation). For example, when third-party reviewers generate tests exposing defects in a database application, these tests may be shipped back to the developers of the application for fixing defects. In such circumstances, the developers may expect the tests to be executable on the application in its original format. To handle this situation, MODA includes the *Test Transformer* component to transform the generated tests, making them executable directly on the target database application.

Consider that each test generated by MODA essentially inserts a set of records into the mock database to synthesize an initial database state. For each generated test, the $Test\ Transformer$ replaces each insertion in it with the execution of a corresponding $INSERT$ query on the original database. Formally, the component substitutes each insertion $M_I(T, C, V)$ of the test with an execution of "$INSERT\ INTO\ T(c_1, c_2, ...c_n)\ VALUES\ V$" in the original database, where $C = \{c_1, c_2, ...c_n\}$.

[2]The number of records in each table is also turned into a symbolic value.

3. EMPIRICAL EVALUATIONS

We conducted two empirical evaluations to evaluate the usefulness and effectiveness of MODA in both regulatory and ordinary contexts. In our first evaluation, the test subject was a point-of-care assistant system, which communicated patient health data and treatment plans with a remote database. Because of confidentiality, we conceal the identity of this system, but simply refer to it as *the device*. The evaluation focused on the GUI component of the device, as the component contained intensive interaction with the remote database. Within the GUI component, 62 SQL queries were found, among which 54 were $SELECT$ queries. However, only 39 of such queries were testable. The remaining 15 $SELECT$ queries either contained illegal characters or were affected by illegal queries. Hence, the evaluation was conducted on the 39 testable queries, which spread over 12 classes that consist of 2,994 lines of code (LOCs).

The subject of the second evaluation was an open source library $Odyssey$[3], out of which two classes, `DAL` and `BizContext`, were selected for test generation. Class `DAL` was found to have 33 methods involving SQL queries, while 20 methods in class `BizContext` contained no SQL queries but invoked one or more query-involving methods in `DAL`. The evaluation focused on generating tests for these 53 methods, which in total include 1227 LOCs.

Evaluation Settings. In both evaluations, MODA and Pex were applied to the evaluation subjects, under the assumption that the related databases were initially empty. When applying Pex in both evaluations, we replaced the query execution statements with invocations to stub methods that returned empty result sets to simulate an empty database. We also equipped the code under test with necessary factory methods, so that Pex could generate values for objects with non-primitive types.

In addition, we also applied the PMO approach to the evaluation subjects, and measured the number of false warnings produced in comparison with MODA. To implement the PMO approach, we bypassed the execution of each encountered SQL query, and instructed Pex to inject a symbolic value to represent the result of executing this query. We manually inspected each test generated by MODA and the PMO approach to determine whether the test represented a false warning.

Evaluation Results. Table 1 summarizes the results of our evaluations, where columns LOC and C list the sizes of test subjects and the number of classes tested in these subjects, respectively. As illustrated in column C_M, tests that MODA generated in both evaluations achieved approximately 90% branch coverage (91.9% in the first evaluation and 89.7% in the second). In contrast, Pex produced tests that covered only 64.4% and 77.2% of branches of the two subjects, respectively (as shown in column C_P).

The evaluations also showed that MODA improved Pex's capability to detect defects in SQL queries, because MODA calculated the runtime content of SQL queries during test generation. In the first evaluation, MODA reported 3 defects related to SQL queries, none of which was found by Pex or PMO approach. These defects are as follows: (1) illegal characters were found in a query, (2) a query used an incorrect name to refer to a column in the database, and (3) the WHERE condition of a SELECT query might potentially assemble empty conjunctions.

The PMO approach achieved the same branch coverage as MODA in both the evaluations. In the first evaluation, the PMO approach produced no false warnings since there were no interactions among database queries. However, in the second evaluation, running the tests generated by the PMO approach resulted in false

[3]`http://odyssey.codeplex.com/`

Table 1: Results of Two Empirical Evaluations

Subject	LOC	C	M	C_M	C_P	C_M-C_P
Medical Device	2994	12	46	91.9%	64.4%	27.5%
Odyssey	1227	2	53	89.7%	77.2%	12.5%

```
void SaveCategory(Category category){
    ...
    if (category.Id == 0) Dal.CreateCategory(category);
    Dal.UpdateCategory(category);
    ...
}

public void UpdateCategory(Category category){
   int n = Execute("update Category ...", category.Name,
    category.Order, category.Id);
   if (n != 1) throw new DBEntityNotUpdatedException();
}
```

Figure 2: An example segment of *Odyssey* code

warnings for 8 methods of class `BizContext`, as compared to no false warnings when running tests generated by MODA.

Figure 2 shows a snippet of code that the PMO approach produced a false warning for. The method `SaveCategory` in Figure 2 invokes method `CreateCategory` if the input object `category` has its `Id` member equal to 0. The mere functionality of `CreateCategory` is to insert into the `Category` table a blank record, which is then updated by the `UpdateCategory` method. An exception will be thrown out if either the insertion or update operation fails.

The PMO approach did generate a test with `category.Id` $= 0$ for the code shown in Figure 2. Running this test, however, resulted in a false warning: a record was first inserted into table `Category`, as evidenced by the fact that `CreateCategory` did not throw an exception. However, since the PMO approach could not memorize what records were maintained in table `Category` after the insertion, method `UpdateCategory` threw an exception, indicating that no record was found in the table.

Threats to Validity. A threat to validity includes faults in our prototype, such as the potentially incorrect or incomplete translation from constraints underlying SQL queries into constraints that Pex can handle. One way of reducing this threat is to avoid this translation. In fact, this translation is no longer necessary if our approach uses a constraint solver that can universally solve constraints from both imperative programs and database queries.

Another threat to validity is the degree to which the test subjects represent true practices, considering that these subjects used only relatively simple SQL queries to realize database interaction. Moreover, since we had no access to the actual databases in both evaluations, we could not compare MODA with other test generation techniques that required executing actual databases. The effectiveness of MODA needs to be further evaluated in circumstances where the actual databases are available and the target applications interact with their databases in a more complex manner.

4. RELATED WORK

The research work closest to ours was proposed by Emmi et al. [4]. Their work also relies on concrete execution to explore program paths and on symbolic execution to collect constraints representing unexplored program paths. Compared to Pex, their approach integrates a constraint solver that universally provides solutions for constraints from both SQL queries and the application under test, and hence crosses the semantic boundary between the application and its database back-end. However, for the purpose of concrete execution, this approach requires the presence of the actual database back-end, which is not always available as discussed earlier. In fact, this approach can be used in our approach to replace Pex, so that its application can be extended to circumstances where database back-ends are not available.

Techniques such as the one proposed by Willmor and Embury [12] rely on human input to synthesize initial database states as a part of tests for database applications. It is challenging for one to specify a set of database states that are comprehensive enough to exercise the code under test. Our approach, in contrast, requires no specification effort from developers in terms of generating initial (mock) database states.

Several techniques [1, 3] use either predefined or random tests for database applications. Such techniques do not use control- and data-flow information of data applications, and hence usually generate abundant unnecessary tests before hitting a new program path. In contrast, our approach considers both program structures and SQL queries that a database application has, and guarantees that each generated test covers a program path not yet covered.

Techniques [2, 11] are also available for generating tests for only SQL queries, where each query is considered independently. Our approach, in contrast, takes into account the dependency among SQL queries, and hence produces no infeasible database states.

5. CONCLUSION

In this paper, we presented an approach that combines mock databases and DSE to generate quality tests for database applications. Our approach does not require database back-ends as input, making it suitable to be used in circumstances where it is difficult to access database back-ends. Empirical results show that our approach is capable of generating tests with higher code coverage than existing DSE-based techniques.

Although MODA is currently implemented on Pex, the key ideas of MODA are general for testing database applications. Other test generation techniques can also be used by MODA to replace Pex, as long as these techniques are extended to manipulate mock databases.

Acknowledgment. This work is supported in part by NSF grants CCF-0725190, CCF-0845272, CCF-0915400, CNS-0958235, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU SOSI.

6. REFERENCES

[1] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna. A Genetic Approach for Random Testing of Database Systems. In *Proc. VLDB*, pages 1243–1251, 2007.

[2] C. Binnig, D. Kossmann, and E. Lo. Reverse Query Processing. In *Proc. ICDE*, pages 506–515, 2007.

[3] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker. An AGENDA for Testing Relational Database Applications. *STVR*, 14:17–44, 2004.

[4] M. Emmi, R. Majumdar, and K. Sen. Dynamic Test Input Generation for Database Applications. In *Proc. ISSTA*, pages 151–162, 2007.

[5] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. *Proc. PLDI*, pages 213–223, 2005.

[6] T. Mackinnon, S. Freeman, and P. Craig. Endo-Testing: Unit Testing with Mock Objects. In *Extreme Programming Examined*, pages 287–301. Addison-Wesley Longman, 2001.

[7] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proc. FSE*, pages 263–272, 2005.

[8] `http://msdn.microsoft.com/en-us/library/system.data.sqlclient.aspx`.

[9] N. Tillmann and W. Schulte. Parameterized Unit Tests. In *Proc. ESEC/FSE*, pages 253–262, 2005.

[10] N. Tillmann and W. Schulte. Mock-Object Generation with Behavior. In *Proc. ASE*, pages 365–368, 2006.

[11] M. Veanes, P. Grigorenko, P. de Halleux, and N. Tillmann. Symbolic Query Exploration. In *Proc. ICFEM*, pages 49–68, 2009.

[12] D. Willmor and S. M. Embury. An Intensional Approach to the Specification of Test Cases for Database Applications. In *Proc. ICSE*, pages 102–111, 2006.

[13] C. Zaniolo and M. A. Meklanoff. On The Design of Relational Database Schemata. *ACM Trans. Database Syst.*, 6:1–47, 1981.