Dynamic Symbolic Database Application Testing

Chengkai Li, Christoph Csallner
Department of Computer Science and Engineering
University of Texas at Arlington
Arlington, TX 76019, USA
{cli,csallner}@uta.edu

ABSTRACT

A database application differs form regular applications in that some of its inputs may be database queries. The program will execute the queries on a database and may use any result values in its subsequent program logic. This means that a user-supplied query may determine the values that the application will use in subsequent branching conditions. At the same time, a new database application is often required to work well on a body of existing data stored in some large database. For systematic testing of database applications, recent techniques replace the existing database with carefully crafted mock databases. Mock databases return values that will trigger as many execution paths in the application as possible and thereby maximize overall code coverage of the database application.

In this paper we offer an alternative approach to database application testing. Our goal is to support software engineers in focusing testing on the existing body of data the application is required to work well on. For that, we propose to side-step mock database generation and instead generate queries for the existing database. Our key insight is that we can use the information collected during previous program executions to systematically generate new queries that will maximize the coverage of the application under test, while guaranteeing that the generated test cases focus on the existing data.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution, testing tools*; H.2.8 [**Database Management**]: Database Applications; D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*

General Terms

Algorithms, Design, Reliability, Verification

Keywords

dynamic symbolic execution, test case generation

1. INTRODUCTION

Maximizing code coverage is an important goal in database application testing. The more application code our test cases cover during testing, the higher is our confidence in the correctness of the application. How to generate test cases to increase code coverage is a well-known software engineering problem. If the program under test is a database application, this issue is compounded by several problems that are not addressed by traditional testing techniques. One of these additional problems stems from the fact that a database application may expect a query as input, evaluate it on a database, and use the values in the result set as normal program values in subsequent program logic. This means that a user-supplied query may determine the values that the application will use in subsequent branching conditions. In order to maximize coverage of database application, we therefore propose to generate database queries such that the result sets returned from the database will lead to different branching decisions in the subsequent program logic and thereby maximize code coverage.

In this paper, we present a novel technique to maximize code coverage in database applications. Our key insight is that we can observe the outcome of the branching decisions taken during program execution, selectively invert some conditions, and convert the resulting constraints to database queries. In this way, our technique systematically generates new database queries, such that the values of their result sets will trigger different branching decisions and thereby maximize code coverage.

Our technique offers two main advantages over recent approaches [2, 7, 17] to database application testing. (1) By generating database queries that the application will issue against real data, we side-step the entire hard problem of generating mock databases. This means that (2), with our technique, we do not have to worry if a generated mock database is representative of an actual production database. Instead, our technique focuses on systematic application testing with actual business data. This is important for several software engineering tasks. For example, for an initial round of testing, a software engineer may want to ensure that a new application is able to process existing data. In such a scenario, we want a test case generator to only generate test cases that correspond to real data.

2. MOTIVATION AND EXAMPLE

New database applications are often required to work well with the data that is already stored in the database. For example, a new insurance claims application should work correctly for the existing customer data, a new book search application should process existing book data, etc. In many cases, existing data is not just representative of but identical to the data that the application is expected to handle. This is due to the fact that large real-world databases are often relatively static—new data is added, but existing data is preserved. Preserving data makes sense, as they are often very valuable to the data owner. Given this "mostly append-only" character of many real-world databases, software engineers need an effective way to test a new application on existing data. Our technique supports software engineers in generating test cases that systematically test the application on the existing data.

```
public void dbfoo(String q) { // "db app"
  query = "Select * From r Where "+q;
  tuples = db.execute(query);
  for (Tuple t: tuples) {
    int x = t.getValue(1);
    bar(x);
  }
}
public void foo(int[] arr) {  // "app"
  for (int x: arr)
    bar(x);
}
public void bar(int x) {
  int z = -x;
  if (z > 0) {              // c1
    if (z < 100)            // c2
      // ..
}
```

Listing 1: Application foo and database application dbfoo. Maximizing code coverage is harder for dbfoo—besides reasoning about program logic, we also need to reason about the input-output relation of the database.

At a high level, a database application differs from a traditional application in that the database application may expect a query as a program input. Since we want to test the application on real data, we refrain from redirecting the query to a mock database that returns arbitrary synthetic data. Instead, we want to execute the query on actual data. This poses the challenge to generate queries such that the result of the program executing the query will trigger many execution paths in the application. For example, in the code of listing 1, method dbfoo is a minimal database application that expects an arbitrary query condition from the user. To test this simple program, we have to generate meaningful query conditions. The challenge is that the number of possible, legal program inputs is infinite. Clearly, we cannot try all of them.

On the other hand, it is also impractical to plainly retrieve all data from the database. In the dbfoo example, this would correspond to a query without WHERE clause. There are several reasons why processing all data is undesirable. First, real-world databases contain so much data that we simply may not have enough resources in testing to run a new application on the entire database. Second, even if we had the resources, we may not get access to the entire data, due to security or other constraints, which essentially place a quota on the amount of data we can retrieve from the database. For example, a book search service may allow users to search a book for arbitrary search terms and return the first n book pages. But to protect the book's copyrights, the service may limit the total number of book pages returned to any given user. Third, the complete access may be impractical due to the limited query capability allowed for the database. For example, deep-Web or hidden-Web sources [1, 4] provide access to the underlying databases through limited query interfaces. An application can only get the data through queries allowed by the interfaces. Although various techniques are proposed to crawl the deep-Web source by querying[14, 11], it is often impossible to enumerate all the queries.

Database sampling is the process of selecting a random sample of database tuples. The data statistics collected from database sampling are useful in many places including data summarization, query optimization, and data mining. Many database sampling methods have been proposed, e.g., [18, 13]. These approaches are optimized for obtaining a sample that is representative of the entire database. A good sample according to that definition however may be a bad sample for our purposes. For us, a good sample triggers a set of program execution paths that is representative of the program execution paths encountered in production use. In other words, existing sampling techniques are not aware of the structure of a given application program. Our technique can be seen as a database sampling technique that is aware of the structure of a database application program.

Recent advances in software testing, i.e., using dynamic symbolic (or "concolic") execution, allow a systematic exploration of a program's execution paths. In a traditional, database-less program (such as method foo of listing 1), assuming program constraints are simple enough, we can use a concolic execution engine (e.g., Dart [8] or Pex [16]) to systematically enumerate the different paths through the program and thereby maximize coverage. The high-level idea is to treat the program input as a symbolic variable, observe the execution path taken, and encode any branching decisions as constraints on the program input variables. Our key idea is that we can leverage dynamic symbolic execution to build database query constraints.

We use a dynamic program analysis technique to build database queries as dynamic techniques are more precise than static analysis techniques. Although typically being faster in deriving constraints from the code, a static analysis may produce results that do not correspond to any actual program execution.

3. ASSUMPTIONS

We focus on such programs that follow tuple-wise semantics. The program takes one user query at a time and issues the query to the database, which returns a set of tuples as the query result. The program iterates through the tuples one by one and applies program logic over each tuple.

For the database query, we assume it is a single-relation conjunctive selection query, where each conjunct is a simple comparison condition between an attribute value and a constant. More formally, the query can be represented in relational algebra as follows:

$$\pi_* \sigma_{c_1\ AND ... AND\ c_k}(R).$$

Each c_i has the form $a \odot v$, where a is an attribute in the schema of R, v is a constant value, and $\odot$ can be $<$, $\leq$, $>$, $\geq$, $=$, or $\neq$. Note that we simply assume the projection (π) always returns all the attributes, since it is trivial and independent of our work. The above query does not consider grouping, aggregation, and join. Database modification queries, such as insertion, deletion, and updates, are not considered either. More complex queries is one subject of our future work.

A program contains a set of methods. One method is designated as the program's entry point or main method. Each method contains a list of statements. A statement reads or writes a local variable or heap location (class field, instance field, or array field), computes an expression, conditionally transfers control (jumps or "branches") to another statement in the method, calls a method, or returns control back to the calling method. A variable is either a bit-vector (boolean, short, int, long, etc.) or a heap reference. We currently ignore floating-point types.

The control-flow graph of a method is a directed graph that depicts the may-execute-next relation on the method's statements. There is an edge from statement s to statement t if t may be executed directly after s. Given program loops, a control-flow graph may contain cycles.

In addition to explicit branching (if-statements and loops) a piece of code may contain several implicit branching statements. These statements, for certain values, throw a runtime exception, which transfers control to the next suitable exception handler. (For example, several language constructs perform an operation on an object or array reference, such as reading or writing an array or instance field or calling an instance method. When attempting to perform such an operation on a null reference, the runtime system will throw a null pointer exception.) The exception handler may be in the same method or a calling method or absent. In the last case, the program terminates. For example, when dividing by zero, the runtime system will throw a runtime exception, which will transfer control to the closest handler or terminate the program. Control-flow graphs traditionally depict only explicit branching statements. One reason for omitting implicit branching is that many language constructs are implicitly branching, which would lead to very dense or cluttered control-flow graphs.

If during execution a branching decision depends on a value returned from the database, we assume that the branching condition can be be rewritten to a form that is permitted by the database query language, e.g., $a \odot v$. Rewriting (normalizing) symbolic expression during dynamic symbolic execution has the important side-effect of keeping the symbolic state representation of the program compact.

4. PROBLEM STATEMENT AND METHOD

4.1 Problem Statement

Given program $\mathcal{P}$, suppose there are s possible program paths. A program path can be represented as a sequence of branching conditions (and their negations). That is, for i from 1 to s, $p_i{=}c_i^1\wedge ...\wedge c_i^{l_i}$, where l_i is the length of the sequence, and each c_i has the form $a \odot v$ or $\sim (a \odot v)$. The set of s possible program paths is $Path(\mathcal{P}){=}\{p_i | 1 \le i \le s\}$.

Our dynamic symbolic engine maintains a tree to represent execution paths. Each node represents the outcome of a branch condition. Therefore each execution path is a path from the root of the tree to some leaf node. For instance, Figure 1 is a tree of execution paths for method *bar* in Listing 1. Note that all the branches of *bar* have been executed, due to the multiple executions of *bar* with different inputs x that are from different database tuples.

The paths in a dynamic execution tree form only a subset

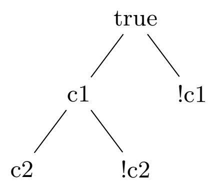

Figure 1: Execution tree for method bar, after executing all branches shown in Listing 1.

Algorithm 1 Iterative Testing Method

```
 1: $q \leftarrow$ get the first test query; $\mathcal{Q} \leftarrow \{q\}$
 2: repeat
 3:     $\mathcal{T} \leftarrow$ run $q$ and get the first $n_q$ result tuples
 4:     for each tuple $t$ in $\mathcal{T}$ do
 5:         run the program over $t$ and update the execution
            tree $tree_{\mathcal{Q}}$ with encountered new execution paths
 6:     $\overline{tree_{\mathcal{Q}}} \leftarrow$ the complement tree of $tree_{\mathcal{Q}}$
 7:     $\mathcal{Q}_c \leftarrow$ get the candidate queries based on $\overline{tree_{\mathcal{Q}}}$
 8:     $q \leftarrow$ select a query from $\mathcal{Q}_c$
 9:     $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{q\}$
10: until stopping criteria satisfied
```

of $Path(\mathcal{P})$, because some program paths may not be reachable given the input to the program. Consider a database instance $\mathcal{R}$ and a program $\mathcal{P}$ that follows the aforementioned assumptions. Given a tuple $t \in \mathcal{R}$, the execution path for t is one of the program paths, i.e., there exists a $Path_t \in Path(\mathcal{P})$. The *reachable paths* of $\mathcal{P}$ under $\mathcal{R}$ is $Path(\mathcal{P},\mathcal{R}) = \{Path_t | t \in \mathcal{R}\}$. Our test method $\mathcal{M}$ iteratively sends a set of queries $\mathcal{Q}{=}\{q_i\}$ to $\mathcal{R}$. We assume the database, upon receiving a query q_i, will return the set of result tuples $\mathcal{R}_i \subseteq \mathcal{R}$ sequentially to the application. Our testing method can choose to only fetch the first n_i tuples $\mathcal{T}_i \subseteq \mathcal{R}_i$. Therefore the *covered paths* of our test method is $Path(\mathcal{P},\mathcal{R},\mathcal{M}) = \{Path_t | t \in \bigcup_{\mathcal{T}_i}\}$. By definition $Path(\mathcal{P},\mathcal{R},\mathcal{M}) \subseteq Path(\mathcal{P},\mathcal{R}) \subseteq Path(\mathcal{P})$.

Our problem is to design a test method $\mathcal{M}$ that chooses test queries $\mathcal{Q}{=}\{q_i\}$ such that $|Path(\mathcal{P},\mathcal{R},\mathcal{M})|$ is as large as possible and at the same time, $\sum_i cost(q_i)$ is as small as possible, where $cost(q_i)$ is the cost of query q_i. These exists a tradeoff between these two goals, therefore it is challenging to achieve a $\mathcal{M}$ that strives for a good balance of them.

4.2 Overview of the Method

We propose a novel iterative approach that employs dynamic symbolic analysis for automatic query generation. The outline of our method is in Algorithm 1. We start with an initial query. The application executes upon the query result tuples, one by one. We analyze the program execution trace and identify which program paths are covered. The uncovered paths are due to unsatisfied branching conditions. The testing method decides how many tuples to fetch for each tested query q_i, based on the collected information. By analyzing the query, the result tuples, the covered and uncovered paths, and the satisfied and unsatisfied branching conditions, the method derives a new query, which potentially can bring new tuples that satisfy the failed branching conditions. Therefore more program paths can be tested. This process continues iteratively, until no more paths could be covered, due to either the lack of data in the underlying database or the full coverage of all the paths.

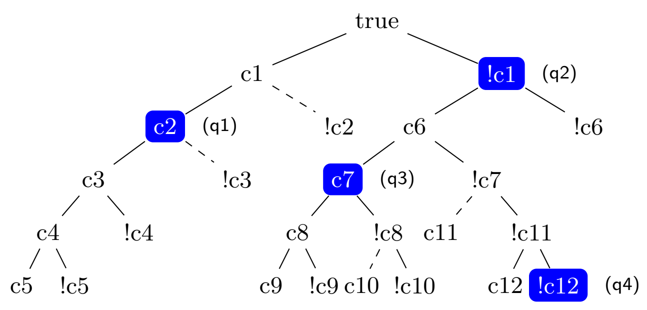

Figure 2: $tree_{\mathcal{Q}}$, the execution tree after the k queries are tested.

Given a test query q_i, our method will not always exhaust the result tuples. Instead, it may decide that it has reached the point of diminishing returns such that testing more tuples may not be as effective as using a new test query. To intuitively understand this, imagine the result of q_i as a sequence of tuples. Each tuple results in an execution path. Multiple tuples may result in the same path. In other words, from the viewpoint of covering program paths, they are duplicate tuples. After a certain number of initial tuples, most or all distinct paths may have been encountered, therefore we should stop using more tuples from q_i. The benefits are twofold: less tested tuples means less testing cost, considering that the program may take a substantial amount of time to run/test for even just one tuple; less tested tuples also means less query execution cost, especially for "non-blocking" query plans that produce results in a pipeline.

Suppose our testing method has issued k queries, $\mathcal{Q}$=$\{q_1$, ..., $q_k\}$. Each query q_i yields covered paths $\{Path_t | t \in \mathcal{T}_i\}$, therefore the covered paths till q_k is $Path(\mathcal{P},\mathcal{R},\mathcal{M},\mathcal{Q})$ = $\{Path_t | t \in \bigcup_{i=1}^{k} \mathcal{T}_i\}$. The key task of the testing method is thus to derive the next test query q_{k+1}. Given the goal of maximizing $|Path(\mathcal{P},\mathcal{R},\mathcal{M})|$ and minimizing $\sum_i cost(q_i)$, our method is a greedy algorithm that aims at local optimum, i.e., maximizing $|Path(\mathcal{P},\mathcal{R},\mathcal{M},\mathcal{Q}\cup\{q_{k+1}\})|-|Path(\mathcal{P},\mathcal{R},\mathcal{M},\mathcal{Q})|$ while minimizing *cost* (q_{k+1}).

4.3 Details of the Method

Execution Tree

The dynamic symbolic engine maintains and updates an execution tree that represents the execution paths encountered for the tuples retrieved for the tested queries. We use $tree_{\mathcal{Q}}$ to denote the execution tree after the k queries are issued. One example is illustrated in Figure 2. Each node in the tree represents a branching condition. Therefore each node can have at most two child nodes, corresponding to two opposite conditions c_i and $!c_i$, respectively. Each path from the root to a leaf corresponds to the executed program path for the tuples that satisfy the conjunctive conditions in the path. For instance, the path from the root to $!c4$ is the execution path for tuples satisfying $c_1 \wedge c_2 \wedge c_3 \wedge !c4$. Among the nodes in the tree, k nodes correspond to the k issued queries, thus are called *queried nodes*. We highlight the queried nodes and list the corresponding query beside each *queried node*. For example, in Figure 2, the 4 issued queries are q_1=$c_1 \wedge c_2$, q_2=$!c_1$, q_3=$!c_1 \wedge c_6 \wedge c_7$, and q_4=$!c_1 \wedge c_6 \wedge !c_7 \wedge !c_{11} \wedge !c_{12}$, respectively.

Complement Tree

For each internal node that has only one child node, we add the other child (which corresponds to the inverse condition of the existing child), to form a *complement tree*. The added nodes are thus called *complement nodes*. We use dashed lines to represent the edges to the complement nodes. For example, in Figure 2, the complement nodes are $!c_2$, $!c_3$, c_{10}, and c_{11}. The path from the root to each complement node, called a *complement path*, represents an execution path that has not been encountered during testing. That path is only different from (the prefix of) some encountered path by the last condition. For example, the path for $!c_3$ is $c_1 \wedge c_2 \wedge !c_3$. It is different from $c_1 \wedge c_2 \wedge c_3$ by the last condition, and $c_1 \wedge c_2 \wedge c_3$ is the prefix of several executed paths, including the ones for c_5, $!c_5$, and $!c_4$.

Candidate Queries

Our task is to derive the next query q_{k+1} that can potentially cover some complement paths. Note that the goal of our testing method is to cover as many *reachable paths* as possible and avoid wasting efforts in trying to cover a path that cannot be satisfied by any tuple in the database. A complement path is only slightly different from some existing path, therefore the chance for the database to have some tuples satisfying that path is higher than an arbitrary path. Moreover, the query for a complement path is straightforwardly generated by only inverting the last condition of some encountered path. On the contrary, using static analysis to generate other uncovered paths may produce results not corresponding to any actual program execution.

Based on the above intuition, a node in the complement tree represents a candidate for the next query q_{k+1}, if itself or at least one descendant node is a complement node. In Figure 2, the candidates are *true* (the root), c_1, c_2, $!c_3$, $!c_2$, $!c_1$, c_6, c_7, $!c_8$, c_{10}, $!c_7$, c_{11}. [1] We further elaborate on this.

The following types of nodes are candidates:

- *a leaf complement node*: e.g., $!c_2$ corresponds to path $c_1 \wedge !c_2$, which has not been encountered before;
- *a queried node* that has at least one descendant that is a complement node: e.g., c_7 is a queried node and its descendant c_{10} is newly added. Using query $!c_1 \wedge c_6 \wedge c_7$ we can potentially get tuples that cover $!c_1 \wedge c_6 \wedge c_7 \wedge !c_8 \wedge c_{10}$, which is not covered yet. Note that, as mentioned in Section 4.1, although the query has been tested before, only the first n_i result tuples were fetched. Therefore if the query is to be issued again, we should get the tuples beyond the first n_i;
- *a non-queried internal node* that has at least one complement descendant: e.g., c_6 is a candidate because by querying $!c_1 \wedge c_6$ we can potentially cover $!c_1 \wedge c_6 \wedge c_7 \wedge !c_8 \wedge c_{10}$ and $!c_1 \wedge c_6 \wedge !c_7 \wedge c_{11}$.

The following types of nodes are not candidates:

- *an existing leaf node*: e.g., $!c_6$ corresponds to path $!c_1 \wedge !c_6$, which has been encountered for some tuples. Along that path, there are no more branching conditions, therefore query $!c_1 \wedge !c_6$ will not cover any new path. For the same reason, $!c_{12}$ is not a candidate even though it is a queried node;
- *an internal node* that does not have any complement descendant: e.g., c_3 is not a candidate because all its descendants are already covered.

Choosing the Next Query

Given a real-world application, there can be a large number of candidate queries. The key challenge is to rank the candidate queries and the most highly ranked candidate becomes the next query q_{k+1}. The gist of our method is to

[1]Note that the root corresponds to the query $\pi_*(R)$, which selects all the tuples without constraints.

rank the candidates based on the average cost per new execution path. That is, given tested queries $\mathcal{Q}$ and a candidate query q, the score of q is

$$score(q) = \frac{cost(q)}{|Path'(\mathcal{P}, \mathcal{R}, \mathcal{M}, \mathcal{Q} \cup \{q\})| - |Path(\mathcal{P}, \mathcal{R}, \mathcal{M}, \mathcal{Q})|},$$

where $|Path'(\mathcal{P}, \mathcal{R}, \mathcal{M}, \mathcal{Q} \cup \{q\})|$ is an estimate of $|Path(\mathcal{P}, \mathcal{R}, \mathcal{M}, \mathcal{Q} \cup \{q\})|$, and $cost(q)$ is the estimated cost of q.

The cost of a query q is estimated by the following formula, which has two components. The query cost $q\_cost(q)$ is the cost of evaluating q in the database, and the testing cost $t\_cost(q)$ is the cost of executing the program over the result tuples and maintaining the execution trace by the dynamic symbolic engine. In $t\_cost(q)$, t is the average time for testing each tuple. In $q\_cost(q)$, w is the waiting time before the first result tuple is available and c is the average time for getting each tuple. w and c are constants but can also be query-dependent. Note that more complex and realistic cost formula from a DBMS can be used to estimate $q\_cost(q)$.

$$cost(q) = q\_cost(q) + t\_cost(q) = w + c \times |\mathcal{T}| + t \times |\mathcal{T}|$$

Given the next query q, $|Path'(\mathcal{P}, \mathcal{R}, \mathcal{M}, \mathcal{Q} \cup \{q\})|$ and $cost(q)$ compete with each other, as both are monotonic in the number of retrieved tuples $\mathcal{T}$ for the query. Intuitively the more tuples retrieved, the higher the cost and, at the same time, the more covered program paths. To trade off between $|Path'(\mathcal{P}, \mathcal{R}, \mathcal{M}, \mathcal{Q} \cup \{q\})|$ and $cost(q)$, we estimate $|Path'(\mathcal{P}, \mathcal{R}, \mathcal{M}, \mathcal{Q} \cup \{q\})|$, as a function of the size of $\mathcal{T}$, based on the statistics collected from the tested queried and from query result cardinality estimation. We then find the optimal size of $\mathcal{T}$ that minimizes $score(q)$. Below we give a sketch of the method.

Each node in the complement tree corresponds to a query. A DBMS engine can estimate the size of the query result. For example, in Figure 1, suppose the queries corresponding to $!c_8$, c_{10}, and $!c_{10}$ are q_1', q_2', q_3', respectively. (They are not necessarily tested.) If the estimated result sizes for q_2' and q_3' are 10 and 40, respectively, we can estimate the result size of q_1' to be 10+40=50. In this way we can recursively estimate the size for each node in the tree. Note that we only use DBMS to estimate the size for leaf nodes, otherwise there will be awkward inconsistencies such as the size of q_1' being smaller than the total sizes of q_2' and q_3'. Based on these numbers, assuming the uniform distribution of tuples, we estimate that there is one tuple satisfying $!c_1 \wedge c_6 \wedge c_7 \wedge !c_8 \wedge c_{10}$ in every 5 result tuples of q_1'. Thus if we use q_1' to cover the complement path to c_{10}, we would need 5 tuples. In Figure 1, it is clear that query q_2' is a better choice than q_1'. However, the query corresponding to c_6 may be an even better choice, because it may cover two complement paths by smaller number of tuples per path.

Note that in principle we can fine-tune the above estimation using the real result size of tested queries. We plan to investigate it in the future.

Stopping Condition for Testing

In addition to deciding when to stop for each individual query (i.e., deciding the number of result tuples to fetch), our method also needs to decide when to stop the whole testing procedure. One straightforward case is when there is no candidate query, meaning all possible reachable paths in the program given the database are encountered. Another scenario is when we have exhausted all the tuples in the database, although there are still candidate queries (i.e., uncovered program paths). Thus the uncovered paths can never be encountered by the tuples in the given database. However, it is not straightforward to know if we have exhausted the whole table, since the tuples are obtained through different queries. One solution is to count the number of tuples that reach each program path. For a program path that has been covered by multiple queries, we use the maximum number obtained among these queries. If the total number of tuples across all different paths is larger than the table size, we know for sure that we have exhausted the table. Such size information can be obtained from database catalog.

It also makes sense to stop the testing even when there are still candidate queries and the table is not exhausted, e.g., when the limited recourses available for testing are used up.

5. IMPLEMENTATION

In this section we discuss the implementation of our technique for database application testing.

5.1 Dynamic Symbolic Engine

Our dynamic symbolic execution engine automatically inserts instrumentation code into a given method, yielding an instrumented version of that method. The execution of the instrumented version behaves just like the original, except that it also creates a symbolic representation of the program execution state. In that our engine is similar to previous ones such as Dart, jCute, and Pex [8, 15, 7, 16].

Whenever execution encounters a call to a database (i.e., via Jdbc), we track the corresponding result set. We also track each tuple that the program reads from the result set. This allows us to represent each value read from a tuple with a unique symbolic variable. When we continue dynamic symbolic execution of the method, we thereby obtain a complete symbolic representation of the path taken, in terms of the symbolic variables we introduced to track the database result set.

A common challenge in the implementation of a dynamic symbolic execution engine is minimizing its overhead. For example, with a naive implementation, we can quickly exhaust the main memory of the analysis machine. Several techniques have been described to keep the symbolic state as compact as possible, e.g., in the Pex dynamic symbolic execution system [16].

Compared to other program analysis techniques, an advantage of dynamic symbolic execution is that it makes it relatively easy to model all relevant features of a program precisely. I.e., we can track the exact outcome of a branch condition encountered during execution. Similarly, we can track any other implicit or explicit control flow, including loops, method calls, and recursion. The key observation is that in dynamic symbolic execution we follow one execution path at a time. We merely record in symbolic terms both the state of the runtime system and the executed path. E.g., if a given execution path passes through a loop n times, we record the outcome of $n+1$ branching decisions, one for each iteration of the loop.

5.2 Query and Non-Query Parameters

Up to now we have described a scenario in which a program expects only parameters that are database queries. However, many real-world applications take both user queries and non-query values as parameters. We now extend our scheme of handling query parameters by also handling regular parameters. Like in standard dynamic symbolic execution [8, 15, 7, 16], we treat each regular (non-query) parameter as a symbolic variable. When combined with our query parameters, this means that a resulting path condition ranges over both database attributes and input parameters.

There are two cases. In the first case, the parts of the path condition that deal with database result values are independent from the parts that deal with the non-query parameters. This means we can solve the non-query parts using standard constraint solving techniques and treat the query-related parts as described in Section 4.2. In the second case, the resulting path condition has one or more conjuncts c_b that depend on both the query and non-query parameters. We currently simplify this case using a heuristic that removes all such c_b from the path condition, which results in a path condition that matches case 1 and we proceed as with any other case 1 path condition.

5.3 Prototype Implementation

In our ongoing prototype implementation, we use the code instrumentation facilities of Java 5 to rewrite the user program at load-time [6], via the third-party open source bytecode instrumentation framework ASM [3]. Rewriting the program at the bytecode level allows our analysis to extend to third-party libraries that are not in Java source code.

For programs that expect both query and non-query parameter values, we solve the non-query portion of the path condition using the third-party satisfiability modulo theories (SMT) solver Z3 [12].

6. RELATED WORK

Despite the widespread use of databases in enterprise computing, historically, there has been relatively little research interest in database application testing. Only in recent years, there has been more work on database application testing. Kapfhammer's dissertation [10] provides a good review of the field of database application testing. The following focuses on work closely related to ours.

The most closely related work, by Emmi et al., applies a similar dynamic symbolic execution engine to the problem of maximizing code coverage in database application testing [7]. Reverse query processing [2] takes a query and a result set and generates a corresponding input database. Veanes et al. [17] also generate database state and use the same constraint solver that we use in our implementation, Z3 [12]. However, all of these approaches aim at overall code coverage. They do not distinguish between actual business data and contrived (synthetic, generated) data. Our technique can distinguish between generated and actual data, which is important for several software engineering tasks.

Prior work on generating database queries by Chays et al. [5] uses a black-box technique, which is unaware of source code details such as control or dataflow. It is unclear if and how this technique can scale up to very large databases and complex query conditions.

A recent paper by Haller discusses several challenges of white-box testing of database applications [9]. In this paper, we go one step further and propose a novel technique for database application white-box testing.

7. ONGOING AND FUTURE WORK

We are currently implementing our technique as an application of our new dynamic symbolic execution engine for Java, Dsc. Our next step will be to evaluate our prototype on existing database applications and their respective databases. The goal of our evaluation will be twofold. First, we want to measure if, when allotted the same amount of time as a mock database generation technique, our tool can achieve higher coverage of application code that is reachable with the existing body of data. Second, we plan to conduct a user study in which we will ask users how they rank bug reports by our tool when compared with other tools. We assume that for some tasks users will prefer reports from a tool that only uses actual data in its test cases.

Beyond evaluating our current technique, we plan to extend our technique to also take into account limits that may be imposed on the number of queries we can issue to the database. Such limits are especially common in the context of software services and deep-web sources, which often throttle the number of queries a single user can pose.

8. REFERENCES

[1] M. K. Bergman. The deep web: Surfacing hidden value. *Journal of Electronic Publishing*, 7(1), Aug. 2001.

[2] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *ICDE*, pages 506–515. IEEE, Apr. 2007.

[3] É. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Proc. ACM SIGOPS France Journées Composants 2002: Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, Nov. 2002.

[4] K. C.-C. Chang, B. He, C. Li, M. Patel, and Z. Zhang. Structured databases on the web: observations and implications. *SIGMOD Rec.*, 33(3):61–70, 2004.

[5] D. Chays, J. Shahid, and P. G. Frankl. Query-based test generation for database applications. In *DBTest*, pages 1–6. ACM, 2008.

[6] G. A. Cohen, J. S. Chase, and D. L. Kaminsky. Automatic program transformation with Joie. In *Proc. USENIX Annual Technical Symposium*, pages 167–178, June 1998.

[7] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 151–162. ACM, July 2007.

[8] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI*, pages 213–223, 2005.

[9] K. Haller. White-box testing for database-driven applications: a requirements analysis. In *DBTest*, pages 1–6. ACM, June 2009.

[10] G. M. Kapfhammer. *A comprehensive framework for testing database-centric software applications.* PhD thesis, University of Pittsburgh, 2007.

[11] J. Madhavan, D. Ko, L. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy. Google's deep web crawl. *Proc. VLDB Endow.*, 1(2):1241–1252, 2008.

[12] L. d. Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer, Apr. 2008.

[13] F. Olken. *Random Sampling from Databases.* PhD thesis, University of California, Berkeley, 1993.

[14] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *VLDB '01*, pages 129–138, 2001.

[15] K. Sen and G. Agha. Cute and jCute: Concolic unit testing and explicit path model-checking tools. In *CAV*, 2006.

[16] N. Tillmann and J. de Halleux. Pex - white box test generation for .Net. In *Proc. 2nd International Conference on Tests And Proofs (TAP)*, pages 134–153. Springer, 2008.

[17] M. Veanes, P. Grigorenko, P. de Halleux, and N. Tillmann. Symbolic query exploration. In *ICFEM*, pages 49–68, 2009.

[18] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.