Dynamic Test Input Generation for Database Applications*

Michael Emmi
UC Los Angeles
mje@cs.ucla.edu

Rupak Majumdar
UC Los Angeles
rupak@cs.ucla.edu

Koushik Sen
UC Berkeley
ksen@cs.berkeley.edu

ABSTRACT

We describe an algorithm for automatic test input generation for database applications. Given a program in an imperative language that interacts with a database through API calls, our algorithm generates both input data for the program as well as suitable database records to systematically explore all paths of the program, including those paths whose execution depend on data returned by database queries. Our algorithm is based on concolic execution, where the program is run with concrete inputs and simultaneously also with symbolic inputs for both program variables as well as the database state. The symbolic constraints generated along a path enable us to derive new input values and new database records that can cause execution to hit uncovered paths. Simultaneously, the concrete execution helps to retain precision in the symbolic computations by allowing dynamic values to be used in the symbolic executor. This allows our algorithm, for example, to identify concrete SQL queries made by the program, even if these queries are built dynamically.

The contributions of this paper are the following. We develop an algorithm that can track symbolic constraints across language boundaries and use those constraints in conjunction with a novel constraint solver to generate both program inputs and database state. We propose a constraint solver that can solve symbolic constraints consisting of both linear arithmetic constraints over variables as well as string constraints (string equality, disequality, as well as membership in regular languages). Finally, we provide an evaluation of the algorithm on a Java implementation of MediaWiki, a popular wiki package that interacts with a database backend.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and debugging. D.2.4 [Software Engineering]: Software/Program Verification.
General Terms: Verification, Reliability.

*This research was funded in part by the NSF grants NSF-CCF-0427202 and NSF-CCF-0546170.

ISSTA'07, July 9–12, 2007, London, England, United Kingdom.

Keywords: directed random testing, database applications, automatic test generation, concolic testing.

1. INTRODUCTION

Programs that interact with database back-ends play a central role in many software systems applications that require persistent data storage and high-performance data access. Such programs include the business logic layer of most middleware systems. The database management system (DBMS) —which is usually bought off-the-shelf— ensures atomic and durable access to large amounts of data, while relieving the applications programmer of the low-level details of storage and retrieval. The correctness of database systems have been the focus of extensive research. The correctness of business applications, though, depend as much on the database management system implementation as it does on the business logic of the application that queries and manipulates the database. While DBMS systems are usually developed by major vendors with large software quality assurance processes, and can be assumed to operate correctly, one would like to achieve the same level of quality and reliability to the business critical applications that use them.

The usual technique of quality assurance is testing: run the program on many test inputs and check if the results conform to the program specifications (or pass programmer-written assertions). The success of testing highly depends on the quality of the test inputs. A high quality test suite (that exercises most behaviors of the application under test) may be generated manually, by considering the specifications as well as the implementation, and directing test cases to exercise different program behaviors. Unfortunately, for many applications, manual and directed test generation is prohibitively expensive, and manual tests must be augmented with automatically generated tests. Automatic test generation has received a lot of research attention, and there are several algorithms and implementations that generate test suites. For example, white-box testing methods such as symbolic execution may be used to generate good quality test inputs. However, such test input generation techniques run into certain problems when dealing with database-driven programs. First, the test input generation algorithm has to treat the database as an external environment. This is because the behavior of the program depends not just on the inputs provided to the current run, but also on the set of records stored in the database. Therefore, if the test inputs do not provide suitable values for both the program inputs and the database state, the amount of test coverage obtained may be low. Second, database applications are *multi-lingual:* usually, an imperative program implements the application logic, and makes declarative SQL queries to the database. Therefore, the test input generation algorithm must faithfully model the semantics of both languages and analyze the mixed code under that model to generate tests inputs. Such an analysis must cross the boundaries between the application and the database.

We describe an algorithm and a tool for the automatic generation of test input data for database applications. Given a program which makes calls to a database through an API, we automatically generate test inputs for the program as well as database states that will attempt to systematically exercise all executions of the application program, including those paths whose execution depend on values returned by database queries. In particular, given a coverage objective such as branch coverage, our algorithm will attempt to find test inputs as well as database states such that each branch of the application is covered.

Our algorithm is based on *concolic execution* [11, 24], that runs a program under test simultaneously on random concrete inputs as well as symbolic inputs. The execution of the program on symbolic inputs, or the symbolic execution, is used in conjunction with a constraint solver to generate concrete inputs for subsequent executions. Our main insight is that during symbolic execution, the database state can be maintained symbolically by tracking the SQL queries made along the program execution path, by translating constraints in an WHERE clause to appropriate constraints in linear arithmetic and over strings. At the end of the execution we get, in addition to a symbolic state giving a path constraint, a constraint on the database state whose satisfying assignments are records that, if inserted to the database, will return positive results for queries along the path.

In more detail, our testing algorithm performs *concolic testing* of the source code [11, 24]. This involves running the program simultaneously on random inputs and some initial database state as well as on symbolic inputs and a symbolic database. The symbolic execution generates constraints, called path constraints, over the symbolic program inputs along the execution path as in [11, 24]. In addition, the algorithm generates constraints over the symbolic database, called database constraints, by *symbolically* tracking the *concrete* SQL queries executed along the execution path. Observe that to track a SQL query symbolically, we need the concrete string representing the SQL query. Often such a string cannot be inferred precisely by statically looking at the program [7, 12, 13] or by observing a symbolic execution. However, since we have the side-by-side concrete execution, we can get the exact strings representing dynamic queries made to the database, without requiring static analysis of strings.

At the end of one execution we consider, for each branch hit on the path, the path constraints and database constraints up to that branch, negate the last path constraint, and find satisfying assignments to these constraints. The satisfying assignment either gives new values to the program inputs, or suggests records that must be inserted in the database in order for the new branch direction to be executed. The program is then run concolically (i.e., both concretely and symbolically) on these new inputs (or run after the records have been inserted in the database) to generate further coverage. This continues until coverage goals are met.

Technically, satisfying assignments are obtained using a constraint solver for linear arithmetic together with a constraint solver for string constraints. While our constraint solver is approximate, in that it assumes that arithmetic and string constraints do not interact, and therefore may fail to find satisfying assignments, we have found it adequate for a large number of SQL query examples.

The problem of finding appropriate test inputs for database applications has been studied before, and semi-automatic user driven techniques to add records to the database have been proposed [5, 6]. Most of these techniques ask the user to suggest appropriate categories for the attributes, and then fill up the database using pseudo-random records chosen from the user-specified attribute value ranges. In contrast, our test generation technique considers the actual execution of the program, and adds records to the database as direct responses to actual queries made by the program on the database. The two techniques are complementary: user-driven record generation can be used to initialize the database to some state, and our technique can be applied on top to target coverage goals that have not been met by pseudo-random testing.

Our work is orthogonal to techniques that look for well-formedness errors in SQL querying programs [12, 13], or for security vulnerabilities in database applications, especially vulnerabilities exploiting SQL injection attacks [15, 26]. We assume the queries are well-formed, and aim to generate maximal coverage of the program paths by symbolically tracking the database state, and automatically generating appropriate records to be included in the database that (by being returned as results of database queries) will exercise specific program paths.

In summary, our contributions are the following.

- A test input generation algorithm for applications that interact with database management systems, that extends concolic testing with simultaneous symbolic tracking of application state as well as database state;

- A constraint solver that can solve symbolic constraints consisting of both linear arithmetic constraints over variables as well as string constraints (string equality, disequality, as well as membership in regular languages); and

- An implementation of the test input generation algorithm for Java programs using the JDBC interface, and an evaluation of the algorithm on a Java implementation of MediaWiki, a popular wiki package.

2. OVERVIEW: AN SQL-QUERYING APPLICATION

We provide an overview of our approach using a small Java method making SQL queries. The code, shown in Figure 1, contains both Java code interfacing with the database through JDBC and queries written in SQL. The goal of our test generation approach is to generate sets of both program inputs and suitable database records to direct execution through each feasible syntactic code path. To enable the generation of such a complete set of test inputs, we need to address the following challenges:

```
void query(int preferred) {
    int inv;
1:  DriverManager.registerDriver(...);
2:  Statement stmt =
        DriverManager.getConnection(...)
          .createStatement();
3:  if (preferred == 1)
4:      inv = 0;
5:   else
6:      inv = 100;
7:  String query =
        "SELECT * FROM books WHERE inventory > "
        + inv + " AND subject LIKE 'CS%'";
8:  ResultSet results = stmt.executeQuery(query);
9:  while (results.next()) {
10:     String val = results.getString("publisher");
11:     Long isbn = results.getLong("isbn");
12:     if (val.equals("ACM"))
13:           this.discountSet.add(isbn, 20);
14:     else
15:           this.discountSet.add(isbn, 10);
    }
}
```

Figure 1: A database-querying Java method

- In addition to the Java code, the dynamically constructed SQL queries must be identified and symbolically executed, and symbolic state must be transferred across the language boundaries from Java to the database and back.

- The set of constraints generated during symbolic execution must be solved so that we can generate both program inputs and database records.

In our algorithm, we address both these challenges and show that we can generate test inputs for systematic testing of database applications. In the example, we consider *branch coverage* as the testing target, as opposed to full path coverage. Our techniques can be extended to (bounded depth) path coverage in a standard way [11].

The code in Figure 1 queries a database of books to figure out a set of books that will be sold at a discount. Books will be sold at a discount if they are on CS, and if their inventory is high. However, preferred customers get the discount irrespective of the inventory. The discount is different for different publishers: ACM books are discounted 20%, all others are discounted 10%. The method takes a parameter `preferred` signifying whether the computation is for a preferred customer.

The first two lines of the code (lines 1 and 2) open a database connection and set up a statement. Lines 3 to 6 conditionally set the variable `inv` to 0 or 100, based on the input flag `preferred` which identifies preferred customers. Line 7 sets up the query as a string: the query looks for all books whose inventory is more than `inv` copies, and whose subject is a string that starts with “CS”. Notice that the value of `inv` (0 or 100) depends on the input `preferred`. Line 8 executes this query on the database, and constructs a `ResultSet`, i.e., a set of records that satisfy the query. The `while` loop on lines 9-15 iterates over the records returned by the query, adding all books satisfying the query to a set `discountSet` representing the books that would be sold at a discount. If the publisher is “ACM” (the test on line 8), the discount is 20%, otherwise it is 10%. We omit some error handling code for readability.

In order to obtain full branch coverage for this code, we have to execute the code for values of the input set to 1 (i.e., the user is preferred) or not 1 (the user is not preferred), and in database contexts that (1) do not contain books with inventory more than `inv` or books on CS, (2) contain books with more than `inv` copies and on CS, (3) contain books on CS with more than `inv` copies with publisher “ACM” as well as books whose publisher is not “ACM.” An usual symbolic execution based test generator [11, 24, 30, 31] that ignores the database environment in which the program is run, or which fixes the database with concrete records and only executes queries concretely, may not be able to obtain full coverage if all the different database states are not considered while testing. For example, if the testing naively starts with a freshly installed copy of the program and the database, the query will not return any results, and the body of the `while` loop will not be executed. What we need is an algorithm that treats database queries symbolically, and is able to modify the database state (by inserting or deleting records) so that the program is exercised along all the different paths, based on the outcome of database queries made along the execution.

Our test generation algorithm works as follows. It starts by executing the program on random inputs and with an initial database state. We shall assume for simplicity that the database is empty to begin with. While executing the program, our analysis simultaneously constructs a *path constraint* consisting of symbolic constraints on program variables that must hold in order to execute the path, as well as a *database constraint* consisting of both database metadata and the actual SQL queries executed.

For the first execution, we choose a random value for `preferred` and run the program with an empty database. In this run, the value of `preferred`, having been set randomly, is very likely to be unequal to 1, so the `else` branch on line 6 will be executed. Since the database is empty, the result set returned on line 8 is also empty and the `while` loop is not entered. The path constraint for this path sets a constraint $\texttt{preferred} \neq 1$, reflecting the else branch of the conditional executed on line 3. Moreover, it treats the variable `results` as a symbolic variable, and states that $\texttt{results} = \emptyset$ (which is the abstraction for the predicate $\texttt{results.next}()$ being false). The database constraint contains the set of attributes of the table `books`, and moreover records that any record v in the relation `results` must satisfy the constraint (obtained from the concrete SQL query):

$$v.\texttt{inventory} > 0 \wedge v.\texttt{subject}\ \mathsf{LIKE}\ '\texttt{CS}\%' \qquad (1)$$

The first constraint in the above expression is a constraint in linear arithmetic, and the second one is a *string constraint* that stipulates that in any satisfying assignment, the string variable `subject` must be assigned a string from the regular expression $\texttt{CS}\Sigma^*$ of all strings starting with the letters `CS`, followed by any sequence of 0 or more letters from the alphabet Σ. In particular, the branch on line 5 that enters the body of the loop can be taken only if `results` is not empty, i.e., `results` contains at least one entry satisfying the above constraint.

At this point, our algorithm looks for touched but uncovered branch statements. These are branches such that some test execution has executed the `then` or the `else` branch, but no test has executed the `else`, respectively, the `then`, branch. In our example, the `then` branches at lines 3 and 9 are touched but uncovered. In order to cover the branch at line 3, we negate the path constraint $\texttt{preferred} \neq 1$ to derive a constraint $\texttt{preferred} = 1$ on the input. A satisfying assignment for this constraint sets `preferred` to 1, and we use this new input to execute the program. This time, the `then` branch is taken on line 3, but the while loop is still not entered.

Now we consider the uncovered branch entering the while loop. We negate the constraint $\texttt{results.next}() = \emptyset$, and find a satisfying assignment for this negated constraint together with the database constraint. This entails finding records that satisfy the database constraint from Equation (1) subject to the database metadata that defines the structure of the table `books`. While we assume here that the constraint only consists of the WHERE clause, we can conjoin additional database consistency constraints here as well.

We find a satisfying assignment to the query by using a constraint solver for strings together with a constraint solver for linear arithmetic [8,18]. For our example, our constraint solver can automatically produce a record

`isbn`	1	`publisher`	'ABS@E'
`inventory`	101	`subject`	'CS'

Notice that the attributes `inventory` and `subject` satisfy the constraint, and the other attributes are given arbitrary values.

We add this record to the database and run the test again. This time, the `while` loop is entered, as the database query on line 4 yields a result (namely, the record we added to the database). Since the publisher attribute is not "ACM", the else branch of the conditional is taken. The path constraint records this by storing the constraint

$$\texttt{results.get}(\text{“}\texttt{publisher}\text{”}) \neq \text{'}\texttt{ACM}\text{'} \quad (2)$$

The algorithm now considers the remaining touched but uncovered branch. To cover this branch, we consider the negation of the constraint in Equation (2) and add that to Equation (1). The resulting constraint is solved for satisfying assignments. This time, we get a satisfying assignment

`isbn`	776	`publisher`	'ACM'
`inventory`	122	`subject`	'CS'

Notice that the constraint from Equation (2) forces the publisher attribute to take the value "ACM". This new record is added to the database, and the program is run again. This covers the then branch of the conditional.

At this point, we have achieved full branch coverage, and our algorithm terminates.

As demonstrated, our algorithm performs symbolic execution together with symbolic handling of the database queries. By explicitly tracking the database state, it is able to provide inputs that ensure higher coverage than standard test generation techniques that ignore the database. Moreover, since symbolic execution is performed simultaneously with concrete inputs, we can dynamically generate and trap concrete queries sent to the database.

3. DATABASE-DRIVEN APPLICATIONS

SELECT $A_1, A_2, \ldots, A_n$
FROM $\mathbf{r}$
WHERE $bcond$

DELETE FROM $\mathbf{r}$
WHERE $bcond$

INSERT INTO $\mathbf{r}$
VALUES $(v_1, \ldots, v_n)$

UPDATE $\mathbf{r}$
SET $A = F(A)$
WHERE $bcond$

Figure 2: Syntax of SQL data manipulation operations

bcond	::=	*bcond* OR *bterm*
	|	*bterm*
bterm	::=	*bterm* AND *bfactor*
	|	*bfactor*
bfactor	::=	NOT*bcond*
	|	*id* IS NULL
	|	*arithterm*
	|	*stringterm*
arithterm	::=	*value* θ *value*
value	::=	*id* | *num*
stringterm	::=	*id* LIKE *string*
	|	*id* η *string*
	|	*id* η *id*
θ	::=	$= \mid < \mid > \mid <= \mid >= \mid !=$
η	::=	$= \mid !=$

Figure 3: Simplified grammar for ***bcond***

3.1 Databases

Relational Databases We illustrate our algorithm on programs written in a simple imperative language that interact with a set of relational databases. Let **int** and **string** denote the sorts of integers and finite strings (over some fixed alphabet), and let $\perp$ be a special "null" symbol. We write $\mathbf{int}_\perp$ (respectively, $\mathbf{string}_\perp$) for the set $\mathbf{int} \cup \{\perp\}$ (respectively, $\mathbf{string} \cup \{\perp\}$). A *relational schema* is a finite set of relation symbols with associated arities. The sorts in the arities are either $\mathbf{int}_\perp$ or $\mathbf{string}_\perp$. Each position in an arity is called an *attribute* and given an identifying name. A record is an ordered list of attribute values (the value of attribute A has position $\mathsf{pos}(A)$), and each value is of type **int** or **string** or the null element $\perp$. Thus, a finite relation is a finite set of records.

A *relational database* R over a relational schema S consists of a mapping associating to each relation symbol $\mathbf{r} \in \mathsf{S}$ a finite relation r of the same arity. For a relation r, and a record v with the same arity as r, we write $r \cup \{v\}$ for the relation with all the records of r together with the additional record v. For a relational database R over S, relation symbol $\mathbf{r} \in \mathsf{S}$, and a relation r, we write $R[\mathbf{r} \leftarrow r]$ for the database which maps $\mathbf{r}$ to the finite relation r but agrees with R on every other relation symbol in S.

Data Definition and Manipulation Relations define the abstract data model. The definition of relational schemas and the manipulation (insertion, deletion, and querying) of data are performed by a structured query language (SQL). We focus here on (a simplified fragment of) the data manipulation language (DML) provided by SQL. Figure 2 describes simplified syntax for the DML operations SELECT, INSERT, DELETE, and UPDATE. The SELECT statement queries a database and returns all records that satisfy some constraints (for simplicity of exposition, we omit the "join" operation and restrict the select statement to just one relation). The INSERT statement inserts a record into a relation. The DELETE statement removes a set of records satisfying a constraint from a relation. The UPDATE statement modifies a set of records in a relation.

The data manipulation statements include a WHERE clause which is used to define a predicate that restricts a relation to a subset of its records that satisfy the predicate. Figure 3 shows a simplified grammar for the predicate used in the SQL WHERE clause. A predicate is a boolean combination of atomic conditions. An atomic condition is either a nullary constraint of the form *id* IS NULL, or an arithmetic or string condition. An arithmetic condition constrains the values of attributes of type **int** by performing an arithmetic comparison between variables and integer constants. String conditions come in two flavors. The first compares the value of one string variable to a constant or the value of another variable by equality. The second checks containment of the value of a string variable within a regular language, specified by a regular expression. (For simplicity, our grammar ignores some aspects that can be considered syntactic sugar; for example, the IN clause, indicating containment within a numeric range, can be rewritten using disjunctions.) The semantics of predicates is three-valued: arithmetic or string comparisons involving $\perp$ return UNKNOWN rather than true or false, otherwise, predicates have the expected semantics. Moreover, the result of any arithmetic or string operation where any argument is $\perp$ returns $\perp$.

The semantics of the data manipulation statements are defined using the following helper functions. Fix a schema S, a relational database R over S, an n-ary relation symbol $\mathbf{r} \in \mathsf{S}$, a relation $r = R(\mathbf{r})$, an attribute A of $\mathbf{r}$, and a boolean condition ψ with free variables over the attributes of $\mathbf{r}$. We define the selection $\sigma_\psi(r)$ as the relation

$$\{v \mid v \in r \wedge v \models \psi\}$$

consisting of records in r that satisfy the condition ψ, and the projection $\pi_A(r)$ as the set of values indexed by A, or

$$\{v_{\mathsf{pos}(A)} \mid \langle v_1 \ldots, v_n\rangle \in r\}.$$

Furthermore, for a function F over the sorts of values indexed by A, we define the substitution $r[A \leftarrow F(A)]$ by

$$\{\langle v_1, \ldots, v_{i-1}, F(v_i), v_{i+1}, \ldots, v_n\rangle \mid \langle v_1, \ldots, v_n\rangle \in r \text{ and } i = \mathsf{pos}(A)\}.$$

The data manipulation statements define a transformer from relational databases to relational databases: from an input relational database R, they return a pair $\langle R', r\rangle$ of an updated database R' and a result relation r [28].

The statement DELETE FROM $\mathbf{r}$ WHERE ψ returns the database and result pair $\langle R[\mathbf{r} \leftarrow R(\mathbf{r}) \setminus \sigma_\psi(R(\mathbf{r}))], \emptyset\rangle$. That is, the new database maps $\mathbf{r}$ to the records in $R(\mathbf{r})$ that do not satisfy ψ, and the result relation is ignored.

Given a tuple $\langle v_1, \ldots, v_n\rangle$ of the same arity as $\mathbf{r}$, the statement INSERT INTO $\mathbf{r}$ VALUES $(v_1, \ldots, v_n)$ returns the database and result pair $\langle R[\mathbf{r} \leftarrow R(\mathbf{r}) \cup \{\langle v_1, \ldots, v_n\rangle\}], \emptyset\rangle$. That is, the new database maps $\mathbf{r}$ to the relation that contains all the records from $R(\mathbf{r})$ and in addition contains the record $\langle v_1, \ldots, v_n\rangle$. The returned result is ignored.

The statement UPDATE $\mathbf{r}$ SET $A = F(A)$ WHERE ψ returns the database and result pair $\langle R[\mathbf{r} \leftarrow R(\mathbf{r}) \setminus \sigma_\psi(R(\mathbf{r})) \cup \sigma_\psi(R(\mathbf{r}))[A \leftarrow F(A)]], \emptyset\rangle$. That is, the relation symbol $\mathbf{r}$ is mapped to a new relation where all records in $R(\mathbf{r})$ that do not satisfy ψ are retained (the first term in the union), while each record in $R(\mathbf{r})$ satisfying ψ have their attribute A updated to $F(A)$. Again, the result relation is ignored.

For a relation symbol $\mathbf{r}$ and a set of attributes $\{A_j \mid j \in \{1, \ldots, n\}\}$ of the relation symbol, the statement SELECT $A_1, A_2, \ldots, A_n$ FROM $\mathbf{r}$ WHERE ψ returns the database and result pair

$$\left\langle R, \sigma_\psi\left(\prod_{i=1}^{n} \pi_{A_i}\left(R(\mathbf{r})\right)\right)\right\rangle,$$

That is, the database state is unchanged, and the returned result relation is a mapping from attributes $\langle A_1, \ldots, A_n\rangle$ to the tuples that satisfy ψ.

3.2 Database Application Program

Syntax We shall focus on imperative programming languages that embed DML operations in the program syntax. Let S be a relational database schema. We define an imperative programming language that interacts with a relational database over the schema S. Our programming language has integer, string, record, and relation-valued variables and references to memory. Relation-valued variables are used to transfer data to and from the database. A relation is logically a set of records. For the purposes of our analysis, we assume strings, records, and relations are opaque immutable types that are manipulated using an abstract data type (ADT). The ADT for strings allows creation, comparison, and concatenation of strings. The ADT for relations has a method size to find the number of records in the relation, and an accessor get(int `n`, string `attr`) to get the value of the attribute `attr` of the `n`th record of the relation.

The operations of the language consist of labeled instructions $\ell : s$. Intuitively, the label corresponds to an instruction address. A statement is either (1) the halt statement halt denoting normal program termination, (2) an *input statement* $m := \mathsf{input}()$ that updates the lvalue m with a non-deterministically chosen external input, (3) an assignment $m := e$ where m is an lvalue and e is a side-effect free expression, (4) a conditional statement $\mathsf{if}(e)\mathsf{goto}\ \ell$ where e is a side-effect free expression and ℓ is a program label, (5) a database manipulation statement $m := \mathsf{query}(s)$ over S that updates the lvalue m with the result relation of a DML statement s, and (6) an abort statement signifying program error. Execution begins at the program label ℓ_0. For a labeled assignment statement $\ell : m := e$ or input statement $\ell : m := \mathsf{input}()$ we assume $\ell + 1$ is a valid label, and for a labeled conditional $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$ we assume both ℓ' and $\ell + 1$ are valid program labels. Furthermore, we assume that the relation methods get and size only appear as the primary expression e of an assignment statement $m := e$.

A database application consists of a relational schema S together with an imperative program P containing database manipulation statement over S.

Semantics The set of *data values* consists of program memory addresses (for pointer values) and data values chosen from integers, strings, record, and relation types. We assume the program is type safe. The semantics of the program are given w.r.t. a *memory* consisting of a mapping from lvalue addresses to values, a relational database providing the state of the database, and an input map which is an infinite sequence of values from which inputs are read.

Execution starts from the initial memory M_0 which maps all addresses to some default value in their domain. Given a memory M, we write $M[m \mapsto v]$ for the memory that maps the address m to the value v and maps all other addresses m' to $M(m')$.

The input map represents a sequence of values that provide input for the input statements. Whenever the program reaches an $m := \mathsf{input}()$ statement, the next value is read off the input map and assigned to the address m. We omit type issues in the description of the input map, assuming implicitly that every element of the input map is the correct type. This can be ensured, e.g., by lazily generating the sequence, by looking at the type of the receiver at run time, and generating a value of the correct type as the next member of the sequence. For every finite sequence $\bar{s}$ of values, we associate an input map by appending a random sequence of values to $\bar{s}$. In particular, if $\bar{s}$ is the empty sequence, this is equivalent to running the program with random inputs.

Statements update the memory, the database state, and the input map. The concrete semantics of the program is given as a relation from program location, memory, database, and input map to an updated program location (corresponding to the next instruction to be executed), updated memory which evaluates program expressions in the context of the current memory, updated relational database reflecting changes if any to the database, and an updated input map.

For an assignment statement $\ell : m := e$, this relation calculates, possibly involving address arithmetic, the address m of the left-hand side, where the result is to be stored. The expression e is evaluated to a concrete value v in the context of the current memory M, the memory is updated to $M[m \mapsto v]$, and the new program location is $\ell + 1$. The database and the input map do not change.

For an input statement $\ell : m := \mathsf{input}()$, the lvalue m is evaluated to its address and the transition relation updates the memory M to the memory $M[m \mapsto v]$ where v is the head of the input map, and the new input map is the tail of the old input map. At the same time, the new location is $\ell + 1$. However, the database does not change.

For a conditional $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$, the expression e is evaluated in the current memory M, and if the evaluated value is zero, the new program location is ℓ' while if the value is non-zero, the new location is $\ell + 1$. In either case, the new memory, the database, and the input map are identical to the old ones.

The relational database is updated by the database manipulation statements. For a database manipulation statement $\ell : m := \mathsf{query}(s)$, the expression s is evaluated in the memory M to yield a string that represents a DML statement. The new database state is obtained from the old database state by executing this DML statement. Moreover, the memory M is updated to $M[m \mapsto r]$ where r is the return relation of the DML statement, and m is assumed to be a relation-typed lvalue. Notice that the return relation is empty except for a SELECT statement. The input map remains unaffected by these statements.

Execution terminates normally (resp. abnormally) if the current statement is halt (resp. abort).

Constraint	φ	$::=$	$\alpha \mid \beta \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi$
Arithmetic	α	$::=$	$\sum_i c_i \delta_i \leq c$
String	β	$:=$	$\delta = \delta \mid \delta \neq \delta \mid \delta = s \mid \delta \neq s \mid \delta\ \mathsf{LIKE}\ s$
Atom	δ	$::=$	$x \mid r.\mathsf{get}(c, s) \mid r.\mathsf{size}()$

Figure 4: Constraint language. The variables x, y ranges over integer or string-valued symbolic variables, r ranges over relation-valued symbolic variables, c over integer constants, and s over string (or regular expression) constants.

4. CONCOLIC TESTING OF DATABASE APPLICATIONS

Concolic Execution Concolic execution [3,11,24] extends the concrete semantics of the program by carrying along a symbolic state and simultaneously performing symbolic execution of the path that is concretely being executed. In addition to the memory, database, and input map, it maintains a *symbolic memory map* μ, a *symbolic path constraint* ξ, and a *symbolic database state* Γ. These are filled in during the course of execution. The symbolic memory map is a mapping from concrete memory addresses to symbolic expressions, while the symbolic database state is a mapping from symbolic expressions, of type relation, to logical formulas over symbolic values. The symbolic path constraint is a logical formula over symbolic values. At the beginning of a symbolic execution, μ and Γ are initialized to empty maps and ξ is initialized to *true*.

We use the constraint language shown in Figure 4 to represent path and database constraints. A constraint φ is a boolean combination of arithmetic or string constraints. An arithmetic constraint α is a linear inequality on symbolic atoms, and a string constraint β is either equality (or disequality) comparison or an inclusion constraint δ LIKE ρ for a symbolic atom δ and regular expression ρ. A symbolic atom δ is either a symbolic value x or a function get (with constant arguments c and s for a number c and a string s) or size applied to a symbolic value r. In the latter case, we assume r is of type relation.

The details of the construction and update of the symbolic memory and path constraint is standard [11,24,30]. At every statement $\ell : m := \mathsf{input}()$, the symbolic memory map μ introduces a mapping $m \mapsto x$ from the concrete address m to a fresh symbolic value x, and at every assignment $\ell : m := e$, the symbolic memory map updates the mapping of m to $\mu(e)$, the symbolic expression obtained by evaluating e in the current symbolic memory. The concrete values of the variables (available from the memory map M) are used to simplify $\mu(e)$ by substituting concrete values for symbolic ones whenever the symbolic expressions go beyond the constraint language.

At every conditional statement $\ell :\ \mathsf{if}(e)\ \mathsf{goto}\ \ell'$, if the execution takes the then branch, the symbolic path constraint ξ is updated to $\xi \wedge (\mu(e) \neq 0)$ and if the execution takes the else branch, the symbolic path constraint ξ is updated to $\xi \wedge (\mu(e) = 0)$. Thus, ξ denotes a logical formula over the symbolic input values that the concrete inputs are required to satisfy to execute the path executed so far.

Both the symbolic memory μ and the symbolic database state Γ are updated by the execution of a statement of the form $\ell : m := \mathsf{query}(s)$. For example, if s is of the form

SELECT $A_1, A_2, \ldots, A_n$FROM $\mathbf{r}_1$ WHERE $bcond$, then we create a fresh symbolic value r, which denotes the relation returned by the query, update μ with $\mu[m \mapsto r]$, and update Γ with $\Gamma[r \mapsto (\forall x) bcond']$, where $bcond'$ is obtained from $bcond$ by replacing each occurrence of id by $r.\mathsf{get}(x, id)$. Thus, the map $\Gamma(r)$ gives constraints on each record in the relation r.

The symbolic execution of a statement of the form $\ell : m := m'.\mathsf{get}(m'', m''')$ updates μ to $\mu[m \mapsto \mu(m').\mathsf{get}(M(m''), M(m'''))]$. Notice that the arguments to get are concrete values. Similarly, the symbolic execution of a statement of the form $\ell : m := m'.\mathsf{size}()$ updates μ to $\mu[m \mapsto \mu(m').\mathsf{size}()]$.

Testing Algorithm Given a concolic program execution, concolic testing generates a new test input in the following way. It selects a conditional $\ell : \mathsf{if}(e)\mathsf{goto}\ \ell'$ along the path that was executed such that (1) the current execution took the "then" (respectively, "else") branch of the conditional, and (2) the "else" (respectively, "then") branch of this conditional is uncovered. Let ξ_ℓ be the path constraint corresponding to the current program path up to the location ℓ just before executing the conditional and let ξ_e be the constraint generated by the execution of the conditional (i.e., ξ_e is either $\mu(e) \neq 0$ if the then branch was executed or $\mu(e) = 0$ if the else branch was executed). Using a decision procedure (described in the next section), our algorithm finds a satisfying assignment for the constraint

$$\xi_\ell \wedge \neg\xi_e \wedge \bigwedge_{r \in \mathrm{dom}(\Gamma)} \Gamma(r)$$

A satisfying assignment λ for the constraint is a map from symbolic atoms to concrete values. The assignments to symbolic atoms of the form x that were created during the execution of $\ell : m := \mathsf{input}()$ statements are used to populate the input map. The assignments to symbolic atoms of the form $r.\mathsf{get}(c, s)$ are used to create $\lambda(r.\mathsf{size}())$ many new database records and the records are inserted into the database.

The newly created input map and the database are used for the next concolic execution. Because we create the input map and the database by solving symbolic constraints, the next execution will follow the old execution up to the location ℓ, but then take the conditional branch opposite to the one taken by the old execution, thus ensuring that the other branch gets covered. We iterate this process of concolic execution along with new input and database generation until the required coverage is achieved.

5. CONSTRAINT SOLVING

Given the constraints generated by a concolic execution, the constraint solving algorithm generates satisfying assignments to the constraints, which are used to update both the input map and the database for a subsequent run. Thus, in addition to generating new program inputs (as in symbolic execution based test generation), we generate records that get inserted into the database. Together, the inputs ensure that coverage goals are satisfied.

Our constraint satisfaction algorithm takes as input a formula φ in the constraint language and returns either a satisfying assignment to φ, or failure. Our procedure is sound, in that satisfying assignments are guaranteed to satisfy φ, but approximate, in that it may fail to find an assignment even if one exists.

5.1 Constraint Satisfaction Algorithm

We show the algorithm in the case that φ is a conjunction of atomic formulas or their negations. If it is a general boolean formula, we can either write it out in disjunctive normal form or search for cubes using a propositional SAT solver [8, 10, 25]. First, for each atomic formula of the form x IS NULL we set the variable x to NULL, and then propagate NULL values through the formula. If the formula evaluates to UNKNOWN, we treat the evaluation as unsatisfiable to be consistent with the SQL semantics (which says no results are returned on UNKNOWN). At this point, any negated atomic formula $\neg(x$ IS NULL$)$ is dropped.

Second, for each r, we instantiate the universally quantified predicates $\Gamma(r)$ arising from the symbolic database state for each constant i for which there is some s with $r.\mathsf{get}(i, s)$ occurring in $\Gamma(r)$. Then, we partition the resulting formula with the instantiations into φ_1 and φ_2, where φ_1 is a string formula and φ_2 is an arithmetic formula. Third, we use a decision procedure for linear arithmetic to find a satisfying assignment for φ_2. Finally, we use the automaton based procedure in Subsection 5.2 below to find a satisfying assignment for φ_1. Together, these give a satisfying assignment for φ.

Given a satisfying assignment λ mapping atomic variables to values, we create an input map and database state as follows. For each δ in the domain of λ, if δ is of the form x where x is a symbolic value created during the execution of a $\ell : m := \mathsf{input}()$ statement, then $\lambda(\delta)$ is used to update the input map. In particular, if x is the ith-read symbolic input value, then the ith position in the input map is set to $\lambda(\delta)$.

Similarly for each symbolic relation value r, we create $\lambda(r.\mathsf{size}())$ records that are inserted into the database. For $1 \le c \le \lambda(r.\mathsf{size}())$ we construct the set R_r^c of all δ in the domain of λ such that δ is of the form $r.\mathsf{get}(c, s)$. Then, we use the constraints $\{\delta = \lambda(\delta) \mid \delta \in R_r^c\} \cup \{bcond[c/x] \mid \Gamma(r) = \forall x.bcond\}$ to generate a record and insert it in the database. We fill the unconstrained attributes of the record with arbitrary (random) data. The above procedure ensures that the inserted records satisfy the constraints imposed by the symbolic database state as well as any additional constraints imposed by the path constraint. For example, the constraint $r.\mathsf{size}() = 2 \wedge r.\mathsf{get}(x, \texttt{'dept'}) = \texttt{'CS'} \wedge r.\mathsf{get}(1, \texttt{'name'}) = \texttt{'Bush'}$ could lead to the insertion of $\langle\texttt{'Bush'}, \texttt{'CS'}\rangle$ and $\langle\texttt{'SHDNK4S'}, \texttt{'CS'}\rangle$ into a database table with fields "name" and "dept".

In order to find out multiple satisfying assignments to a constraint, e.g., to generate multiple tuples satisfying a database constraint, we use the following standard trick. Let φ be a constraint with free variables $x_1, \ldots, x_n$. For a satisfying assignment $\bar{s}$ mapping variables x_i to constants s_i for $i \in \{1, \ldots, n\}$, we define the constraint $[\bar{s}]$ as:

$$[\bar{s}] \equiv \bigwedge_{i=1}^{n} x_i = s_i$$

Given the formula φ, we iteratively ask the constraint solver for a satisfying assignment $\bar{s}_1$ of φ, then a satisfying assignment $\bar{s}_2$ for $\varphi \wedge \neg[\bar{s}_1]$, then for $\varphi \wedge \neg[\bar{s}_1] \wedge \neg[\bar{s}_2]$, etc. Iterating this procedure k times gives k distinct satisfying assignments to φ (as long as k distinct satisfying assignments exist).

We note that our algorithm is *approximate* for the entire SQL language, in that our satisfiability procedure is not guaranteed to find a satisfying assignment in all cases even if one exists. For example, it may not be possible to partition a boolean condition into a pure linear arithmetic formula and a pure string formula (since one can take the length of strings). Even if such a partition is possible, the string constraints may not fall into our constraint language (since we do not handle concatenation) and in the presence of operators such as AVG (the SQL average operator) or exponentiation, the arithmetic constraints need not be linear. In general, the satisfiability problem for the theory of strings together with a length function is a long standing open problem [1, 9]. However, since we have the concrete values, we can specialize non-linear constraints to linear ones by substituting the concrete constants for the variables (again, this may lose generality). In practice, we have found our approximate routine adequate for most SQL-querying applications.

5.2 Satisfiability Procedure for Strings

We now outline a decision procedure to check satisfiability of string constraints. Again, we assume we are given a conjunction of atomic string constraints. We begin by normalizing our constraints to be of the form $\delta_1 = \delta_2$, $\delta_1 \neq \delta_2$, or δ LIKE ρ, for atoms δ_1 and δ_2, and regular expressions ρ. To do this normalization, we replace constraints of the form $\delta = s$ (respectively $\delta \neq s$), for variable δ and string constant s, with δ LIKE s (resp., δ LIKE $\bar{s}$), where $\bar{s}$ is a regular expression matching all strings except s.

Let C denote the set of normalized constraints and X the set of variables appearing among those constraints. We then define an equivalence relation $\equiv$ over X by $\delta_1 \equiv \delta_2$ if and only if either (1) δ_1 and δ_2 are syntactically identical, or (2) there is a $\delta \in X$ such that either $\delta_1 = \delta$ or $\delta = \delta_1$ is a constraint in C and $\delta \equiv \delta_2$. The equivalence relation $\equiv$ is the reflexive transitive closure of the equality relation, and can be computed efficiently using a union find algorithm [27].

Once we create the set of equivalence classes of $\equiv$, we check for trivial unsatisfiability by disequality constraints between equivalent variables, that is, we return unsatisfiable if there are two variables δ_1 and δ_2 such that $\delta_1 \equiv \delta_2$, but there is a disequality constraint $\delta_1 \neq \delta_2$ in C.

Let $P = \{p_1, \ldots, p_k\}$ be the set of equivalence classes of the relation $\equiv$. In the second step, we build regular languages L_p, one for each partition $p \in P$. The regular language L_p is obtained by conjoining the regular languages ρ, such that some $\delta \in p$ has a constraint δ LIKE ρ in C. Formally,

$$L_p = \bigcap_{\delta \in p, \delta \text{ LIKE } \rho \in \mathsf{C}} \rho$$

Finally, the constraints are satisfiable iff there exist words $w_1 \in L_{p_1}$, $w_2 \in L_{p_2}$, ..., $w_k \in L_{p_k}$ such that for every constraint $\delta_1 \neq \delta_2$ in C with $\delta_1 \in p_i$ and $\delta_2 \in p_j$, we have $w_i \neq w_j$. These words can be chosen from the k shortest words in each language by a systematic enumeration and search.

The above procedure shows that the decision problem for string constraints is in PSPACE. The satisfiability problem is also PSPACE-hard, since the problem of checking if the intersection of k regular expressions is empty is PSPACE-hard [17], and this can be encoded as the satisfiability question x_1 LIKE $r_1 \wedge \ldots \wedge x_k$ LIKE $r_k \wedge x_1 = x_2 \wedge x_2 = x_3 \wedge \ldots \wedge x_{k-1} = x_k$. While we do not use it here, the PSPACE upper bound also follows from a much deeper decision procedure for the theory of word equations with regular constraints [9].

```
Article getRandomArticle() {
    Article art = new Article();
    String sql;
    int noa = getNumberOfArticles() - 1;
    results.content.clear();

    do {
        int x = (int) ((double) noa * Math.random());
        sql = "SELECT * FROM cur WHERE "
            + "cur_namespace=0 LIMIT 1 OFFSET ";
        sql += x;
        query( sql );

    } while ( results.content.size() == 1
            && results.get(0,"cur_is_redirect")
               .equals("1") );

    if ( results.content.size() == 1 ) {
        String s = results.get(0,"cur_text");
        s = filterBackslashes(s);
        art.setSource(s);
        art.setTitle(
            new Title(results.get(0,"cur_title")) );
    }
    return art;
}
```

Figure 5: The method getRandomArticle—a Java adaptation of code from MediaWiki

Theorem 1. *The satisfiability problem for string constraints is PSPACE-complete.*

In practice, we have found that the string constraints arising in SQL queries are very simple and the procedure is fast.

6. CASE STUDY

We have implemented the test generation algorithm for Java code interacting with databases. Our implementation is built on top of the JCute testing framework [23], which uses the Soot [29] Java optimization framework, and the lp-solve [18] linear program solver. The primary modifications that were necessary included discovering database meta-data, parsing SQL query strings, augmenting JCute's symbolic state space for SQL data, tracking input values originating from a database, and modifying database tables for directed testing. Our implementation parses SQL SELECT statements by using the ANTLR [21] parser generator with a derivative grammar of [22]. Our target programs are written using Java's databases API (package `java.sql`). We use a MySQL database, accessed through a JDBC/MySQL driver, though our implementation is, in theory, portable across differing database and driver configurations.

We ran our program on a Java reimplementation of MediaWiki [19], a popular wiki package. Figures 5 and 6 show the `getRandomArticle` method and support method `query` of a Java package adapted from MediaWiki. The relation `cur` used in the queries above contains two integer fields `cur_namespace` and `cur_is_redirect`, and two string fields `cur_title` and `cur_text`. We label the branch predicates `results.content.size() == 1` and `results.get(0,"cur_is_redirect").equals("1")` with p_1 and p_2, respectively. The field `results` is an instance of the

```
void query(String q) {

1:  results.clean();
2:  try {
3:      DriverManager.registerDriver(...);
4:      Statement stmt =
          DriverManager.getConnection(...)
            .createStatement();

5:      stmt.execute(q);
6:      ResultSet rs = stmt.getResultSet();

7:      if (rs != null) {
8:        ResultSetMetaData md = rs.getMetaData();

9:        for (i=1; i<=md.getColumnCount(); i++)
10:         results.field.add(md.getColumnName(i));

11:       while (rs.next()) {
12:         Vector<String> row =
              new Vector<String>(md.getColumnCount());
13:         for (i=1; i<=md.getColumnCount(); i++)
14:           row.add(rs.getString(i));
15:         results.content.add(row);
          }
        }
    } catch (Exception e) {
       ...
    }
}
```

Figure 6: The method query—an auxiliary method to getRandomArticle

class `SQLResult` of Figure 7. Existing directed testing tools will be unable to direct execution through this code, since they cannot reason about the interaction with the database. Assuming, for example, that the table `cur` is initially empty, p_1 will never be satisfied, and p_2 will never be tested.

On the other hand, a run of our tool under the same initial conditions proceeds as follows. The method calls to `getConnection` and `createStatement` on line 4 of `query` create symbolic values which store some structure of the table `cur`. The call to `execute` on line 5 then adds the concrete query (string) to the symbolic value of `stmt`. When `getResultSet` on line 6 is called, a symbolic value is created for `rs` which keeps a cursor which is modified by subsequent calls to `next` and `prev`. When a value is actually read (e.g., during `getString` on line 14) from `rs`, that cursor is used to index the result set—the column is given by the row name or number passed as an argument to the accessor method (e.g., `getString`). At this point a symbolic input value is created for a particular position in the result set, which will be propagated through the program's execution as a usual symbolic input value.

Ideally, the calls to `results.content.size` and `results.get` in lines 10 and 11 of the method `getRandomArticle` would have symbolic values, but this relies on symbolically reasoning about `Vector`s, the data-structure backing `results`. A symbolic procedure for `Vector`s could propagate the symbolic values read from the database through to predicates p_1 and p_2, and directed testing would be able to accurately predict the trajectory through `getRandomArticle`. Unfortunately, our version of JCute did not have a built-in symbolic interpretation for `Vector`s, and any symbolic information

```
class SQLResult {
    Vector<String> field =
            new Vector<String>();
    Vector<Vector<String>> content =
            new Vector<Vector<String>>();

    void clean() {
        field.clear();
        content.clear();
    }
    String get(int row, int col) {
        return content.get(row).get(col);
    }
    String get(int row, String s)
            throws IndexOutOfBoundsException {

        for (int col=0; col<field.size(); col++)
            if (field.get(col).equalsIgnoreCase(s))
                return get(row,col);
        throw new IndexOutOfBoundsException();
    }
}
```

Figure 7: The class SQLResult of the field results

associated with data stored in a vector would be replaced with concrete data.

However, it turns out, even in the absence of symbolic reasoning about vectors, our algorithm still provides more thorough testing through `getRandomArticle` (we cover 75% of branches reached from `getRandomArticle`, as opposed to the 50% which JCute-alone covers—the remaining 25% is accounted for by the absence of symbolic execution for vectors, and branches which are infeasible). Line 11 of the method `query` tests the predicate `rs.next()` (which we'll label p_3) in order to fill the `result` data with query results. Since the symbolic state for `rs` keeps a cursor into the result set, we can reason about p_3. For example, if the execution of this method fails a test of p_3, then the relevant result set must contain an additional row in order to satisfy p_3 on the next execution. Thus, given an initially empty table `cur`, the symbolic expression for this predicate effectively becomes `num_rows(rs) > 0`, which our solver must consider as a constraint to direct execution though the corresponding loop. However, this constraint combined with the query constraint `LIMIT 1` of line 7 of `getRandomArticle` will also trigger success of the predicate p_1, effectively exploring the previously unexplored paths through `getRandomArticle`. We believe that this situation is not a peculiarity of our particular program, but a common idiom in these programs. Moreover, symbolic reasoning about container data structures will significantly improve the precision and coverage of our implementation in these cases.

7. RELATED WORK

Despite their importance in many business-critical applications, research on testing application programs interacting with databases has been somewhat limited. The Agenda framework for testing database-driven applications [5, 6] uses user-provided data ranges to randomly populate the database with typles that satisfy the schema constraints. Similar user-directed random generation subject to schema constraints have been considered in [20, 32]. While in many cases, the test data and database state can be comprehensive, by ignoring the data and control flow through the program, these techniques may achieve lower coverage. Instead, our techniques are likely to provide better coverage by explicitly considering the program structure and the actual queries made to the database. Of course, the techniques are complementary and can be used together for better testing of database-driven applications, e.g., by starting with an initial database that has been filled with records using the above techniques, and then incrementally adding or deleting records as dictated by the concolic execution.

Orthogonal to our work, the problem of defining appropriate coverage criteria for database driven applications that tracks flow of data through the database has been considered before [2, 14, 16]. Our test generation algorithm is parameterized by the desired coverage goals and can directly use these more refined coverage criteria without change in the test generation algorithm.

Previous attempts [4] have embedded SQL statements into the imperative program, and applied white box testing techniques on the resulting program where the database has been abstracted away in the translation. This provides an alternate approach. We provide an explicit symbolic-execution based automatic test generation strategy as well as a constraint solving algorithm that can handle common constraints on strings and integers arising in the queries. In contrast, the onus of finding tests in [4] was on the user. More importantly, their compilation algorithm assumes that all SQL queries are available statically. This is seldom true in practice, as query strings are dynamically constructed based on control flow. Since concolic execution generates queries dynamically, we do not face this problem.

As mentioned before, our testing algorithm is not geared towards checking syntactic soundness of queries, or for security vulnerabilities in the presence of dynamic queries. Both these aspects have received a lot of attention [13, 15, 26]. In contrast, we aim to test for functional requirements of the software.

8. CONCLUSION

Enterprise applications that interact with database systems are ubiquitous, and there is a need for better validation techniques for these systems. We have presented a novel test input generation algorithm that tracks not only the program state but also the environment (the state of the database). While our early results are encouraging, we have identified some limitations of the presented approach.

First, our implementation assumes a view of the world with two participants: Java programs and the database. In reality, most enterprise applications are built in several different layers, including JavaScript code, browser forms, and a server such as tomcat that mediates data flow. While conceptually the algorithm remains the same, we admit that scaling our implementation to a real enterprise system is a significant engineering effort.

Second, symbolic execution based test generation is ultimately limited by the expressibility of the constraint language and the capacity of the constraint solver. We believe our constraint solver presents a compromise between fast constraint solving and the ability to capture many constraints of practical interest.

Despite these limitations of our current implementation, we believe that context-aware concolic execution presents a powerful tool for automatic test generation and validation of database-driven applications.

9. REFERENCES

[1] J. R. Büchi and S. Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. *Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik*, 22, 1987.

[2] M. J. S. Cabal and J. Tuya. Using an SQL coverage measurement for testing database applications. In *SIGSOFT FSE*, 2004.

[3] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, 2005.

[4] M. Chan and S.-C. Cheung. Testing database applications with SQL semantics. In *CODAS*, 1999.

[5] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weber. A framework for testing database applications. In *ISSTA*, 2000.

[6] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker. AGENDA: a test generator for relational database applications. Technical Report TR-CIS-2002-04, Polytechnic University, 2002. `http://cis.poly.edu/tr/tr-cis-2002-04.shtml`.

[7] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS 03: Static Analysis Symposium*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, 2003.

[8] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3), 2005.

[9] V. Diekert. Makanin's algorithm. In *Algebraic Combinatorics on Words*, volume 90 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2002.

[10] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated canonizer and solver. In *CAV*, 2001.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

[12] C. Gould, Z. Su, and P. T. Devanbu. JDBC checker: A static analysis tool for SQL/JDBC applications. In *ICSE*, 2004.

[13] C. Gould, Z. Su, and P. T. Devanbu. Static checking of dynamically generated queries in database applications. In *ICSE*, 2004.

[14] W. G. J. Halfond and A. Orso. Command-form coverage for testing database applications. In *ASE*, 2006.

[15] W. G. J. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *SIGSOFT FSE*, 2006.

[16] G. M. Kapfhammer and M. L. Soffa. A family of test adequacy criteria for database-driven applications. In *ESEC / SIGSOFT FSE*, 2003.

[17] D. Kozen. Lower bounds for natural proof systems. In *FOCS*, 1977.

[18] lp_solve. `http://groups.yahoo.com/group/lp_solve/`.

[19] MediaWiki. `http://www.mediawiki.org/wiki/MediaWiki`.

[20] A. Neufeld, G. Moerkotte, and P. C. Lockemann. Generating consistent test data for a variable set of general consistency constraints. *VLDB J.*, 2(2), 1993.

[21] T. J. Parr and R. W. Quong. ANTLR: A predicated-$LL(k)$ parser generator. *Softw., Pract. Exper.*, 25(7), 1995.

[22] MS SQL Server 2000 SELECT statement grammar. `http://www.antlr.org/grammar/1062280680642/MS_SQL_SELECT.html`.

[23] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, 2006.

[24] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, 2005.

[25] A. Stump, C. W. Barrett, and D. L. Dill. CVC: A cooperating validity checker. In *CAV*, 2002.

[26] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *POPL*, 2006.

[27] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2), 1975.

[28] J. D. Ullman, H. Garcia-Molina, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[29] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, 1999.

[30] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA*, 2004.

[31] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, 2005.

[32] J. Zhang, C. Xu, and S.-C. Cheung. Automatic generation of database instances for white-box testing. In *COMPSAC*, 2001.