Symbolic Execution of Stored Procedures in Database Management Systems

Muhammad Suleman Mahmood Maryam Abdul Ghafoor Junaid Haroon Siddiqui

Department of Computer Science
LUMS School of Science and Engineering
Lahore, Pakistan

{13030004, 15030036, junaid.siddiqui}@lums.edu.pk

ABSTRACT

Stored procedures in database management systems are often used to implement complex business logic. Correctness of these procedures is critical for correct working of the system. However, testing them remains difficult due to many possible states of data and database constraints. This leads to mostly manual testing. Newer tools offer automated execution for unit testing of stored procedures but the test cases are still written manually.

In this paper, we propose a novel approach of using dynamic symbolic execution to automatically generate test cases and corresponding database states for stored procedures. We treat values in database tables as symbolic, model the constraints on data imposed by the schema and by the SQL statements executed by the stored procedure. We use an SMT solver to find values that will drive the stored procedure on a particular execution path.

We instrument the internal execution plans generated by PostgreSQL database management system to extract constraints and use the Z3 SMT solver to generate test cases consisting of table data and procedure inputs. Our evaluation using stored procedures from a large business application shows that this technique can uncover bugs that lead to schema constraint violations and user defined exceptions.

CCS Concepts

•**Software and its engineering** → **Software verification and validation;** *Software testing and debugging;*

Keywords

Symbolic Execution, Stored Procedures, SQL

1. INTRODUCTION

Symbolic execution is a powerful program analysis technique based on systematic exploration of (bounded) program paths, which was developed over three decades ago [7, 19]. A key idea in symbolic execution is to build *path conditions*—given a path, a path condition represents a constraint on the input variables, which is a conjunction of the branching conditions on the path. Thus, a solution to a (feasible) path condition is an input that executes the corresponding path. A common application of symbolic execution is indeed to generate test inputs, say to increase code coverage. Automation of symbolic execution requires constraint solvers or decision procedures [1, 10] that can handle the classes of constraints in the ensuing path conditions.

A lot of progress has been made during the last decade in constraint solving technology, in particular SAT [31] and SMT [1, 10] solving. These technological advances have fuelled the research interest in symbolic execution, which today not only handles constructs of modern programming languages and enables traditional analyses, such as test input generation [18, 15, 26, 4], but also has non-conventional applications, for example in checking program equivalence [25], in repairing data structures for error recovery [12], and in estimating power consumption [27].

Despite the progress, applying symbolic execution in new domains remains a challenging problem. One such domain is stored procedures in Relational Database Management Systems. Relational databases are widely used for storing and managing data for applications. Many applications using databases interact with multiple users simultaneously. In order to process user requests, the applications mostly need information from database tables (relations), often requiring access to multiple database tables and sometimes involving decisions in the form of conditional statements as well. If database server and application server are on separate machines then the network overhead can be significant due to multiple requests to the database server. In order to avoid this overhead and to place business logic close to data for integrity purpose, application programmers often move business logic for common tasks from the application server to the database server in stored procedures.

Correctness of stored procedures is crucial for correct working of the application as well as for maintaining data integrity. However testing them is a difficult task. Any test case written for stored procedures needs to provide data for database tables in addition to the usual program inputs in order to have reproducible results. Symbolic execution has been used with database driven applications [13, 20, 23, 21, 16, 22]. However, the techniques for testing applications are not directly applicable to testing stored procedures because they run inside a database server. Furthermore, the techniques are limited in

nature due to the challenges posed by multi-lingual nature of these application (SQL and some imperative language) and because of considering the database as an external system. Often, a very limited SQL grammar has been supported for these applications for automated test generation.

This paper presents a novel technique to adapt symbolic execution for automated testing of stored procedures. Our key insight is that instrumenting the internal query nodes of the database query processing engine enables us to gather internal database constraints while being more precise and efficient in producing non-redundant test cases. We adapted symbolic execution technique for stored procedures written in the PL/pgSQL language for PostgreSQL[1]. PL/pgSQL runs directly on the database so it is not an external system. A major advantage of our technique of instrumenting query nodes for PL/pgSQL is that it has the same type system as the database tables. This makes it simpler to establish a relation between program variables and data stored in the database. This is a simplification that is not available to logic written in other languages. Because of these simplification we are able to support a larger subset of the SQL grammar.

In contrast to dynamic symbolic execution of simple programs where program conditions are the only source of constraints, we identified three sources of constraints in symbolic execution of stored procedures. These constraints imposed on program variables are gathered during execution of the program through instrumentation of PostgreSQL. These sources of constraints include

(i)***Language constructs***, e.g. IF statement, are first source of constraints imposed on program variable.

(ii)***SQL constructs*** are the second source of constraints that are indirectly imposed by SQL statements. Based on a key insight, we used the query plans generated for execution of the SQL statements to extract such constraints. The important difference between language and SQL construct is, that the number of paths (choices) depends on the number of rows being processed at the SQL node, e.g. a sequential scan plan node has four choices when the table being scanned has two rows and eight choices with three rows in a table and

(iii) ***Database integrity constraints*** are imposed on the database tables to ensure that the data always satisfies certain properties.These constraints may be violated in some cases during the execution of the application, causing exceptions in the procedures. A typical example would be a primary key constraint violation that occurs when the user requests insertion of a record which already exists in the system.

We make the following contributions:

- **Symbolic execution of stored procedures:** We demonstrate an end-to-end technique for symbolic execution of stored procedures in PostgreSQL.
- **Instrumentation at query node level:** Our instrumentation at the level of query nodes allows more precision and efficiency than was possible using any technique at the abstraction of a high-level query language like SQL.
- **Constraint collection from query nodes:** We identified and collected constraints from three sources using query node instrumentation i.e. conditions on program variables, indirect constraints due to SQL statements, and database integrity constraints.
- **Implementation:** We implemented symbolic execution of stored procedures on PostgreSQL—an open-source database with support for stored procedures and using the Z3 SMT solver [10]. The algorithms are adaptable to other database systems but the implementation is specific for the query node types and processing in PostgreSQL.
- **Evaluation:** We evaluated our technique on 150 procedures that are part of an open source Accounting and CRM ERP, PostBooks[2]. Our symbolic executor has generated around 90 cases that trigger constraint violations or hit user defined exceptions showing the effectiveness of our technique and the limitations of writing manual test cases for stored procedures.

[1]http://www.postgresql.org

2. BACKGROUND

Symbolic execution is a technique for executing a program on symbolic values [8, 19]. There are two fundamental aspects of symbolic execution: (1) defining semantics of operations that are originally defined for concrete values and (2) maintaining a *path condition* for the current program path being executed – a path condition specifies necessary constraints on input variables that must be satisfied to execute the corresponding path.

As an example, consider the following program that returns the absolute value of its input:

```
1   static int abs(int x) {
2     int result;
3     if (x < 0)
4       result = 0 - x ;
5     else
6       result = x;
7     return result;
8     }
```

To symbolically execute this program, we consider its behavior on integer inputs, say `x`. We make no assumptions about the value of `x` (except what can be deduced from the type declaration). So, when we encounter a conditional statement, we consider both possible outcomes of the condition. To perform operations on symbols, we treat them simply as variables, e.g., the statement on L4 updates the value of result to be 0 - x. Of course, a tool for symbolic execution needs to modify the type of result to note updates involving symbols and to provide support for manipulating expressions, such as 0 - x. We perform operations on symbols algebraically.

Symbolic execution of the program `abs` explores two paths:

```
path 1:  [X < 0] L2 -> L3 -> L4 -> L7
path 2:  [X > 0] L2 -> L3 -> L6 -> L7
```

Note that for each path that is explored, there is a corresponding path condition (shown in square brackets). While execution on a concrete input would have followed exactly one of these two paths, symbolic execution explores both.

[2]http://www.xtuple.com/postbooks

```
1  CREATE OR REPLACE FUNCTION updateEmpSalary(id
       integer)
2  RETURNS integer AS
3  $BODY$
4  DECLARE
5     sal integer;
6     experience integer;
7  BEGIN
8     SELECT A.salary, A.experience INTO sal,
          experience from emp A WHERE empno = id;
9     IF NOT FOUND THEN
10      RETURN -1;
11    END IF;
12    IF experience >= 5 THEN
13      sal := sal + 500;
14    ELSE
15      sal := sal + 200;
16    ENDIF;
17    UPDATE emp SET salary = sal WHERE empno = id;
18    RETURN 1;
19 END;
20 $BODY$
```

Figure 1: Example code of stored procedure for test case generation through symbolic execution.

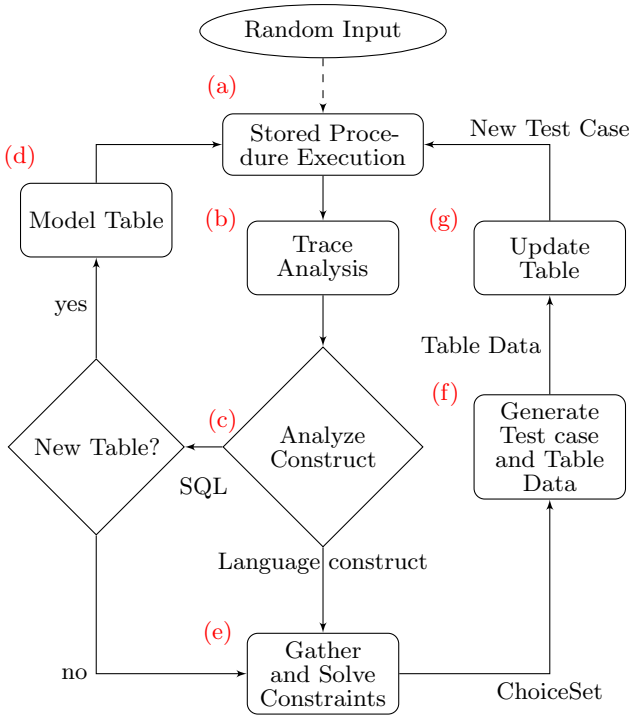

Figure 2: Flow of Symbolic Execution of Stored Procedure

3. ILLUSTRATIVE EXAMPLE

In order to explain symbolic execution of stored procedure, consider procedure 'updateEmpSalary' given in Figure 1. Our database schema for this example is a single table `emp` with table structure given in Table 1, which contains records of the employees of the company. We have primary key and not null constraints on the `empno` and `salary` columns respectively. Before initiation of symbolic execution of 'updateEmpSalary', our program connects to the database to extract signature of

```
1  T_Result TargetList TARGET_ENTRY FunctionCall
       7129501 ARGUMENT_START 23 1 ARGUMENT_END AS
       updateempsalary
2  START_FUNCTION 7129501
3  PLPGSQL_STMT_EXECSQL
4  T_SeqScan 7129489 TargetList TARGET_ENTRY Col 23
       salary AS salary TARGET_ENTRY Col 23
       experience AS experience Conditions 65 Col 23
       empno Param 23 $1
5      Into 23 sal 23 experience
6  PLPGSQL_STMT_EXECSQL_END
7  PLPGSQL_STMT_IF Not Param 0 found
8  PLPGSQL_STMT_IF_END
9  PLPGSQL_STMT_IF 150 Param 0 experience 23 5
10 ...
```

Figure 3: 'Trace Log ' for Example Code

Table 1: EMP Table Structure

No	Attributes	Constraints	Type
1	empno	Primary Key	integer
2	name	-	string
3	salary	Not Null	integer
4	experiance	-	integer

the procedure, i.e. number of input parameters along with their data types. We generate first test case with random value e.g., id = 1. Symbolic execution with concrete input is also called 'Concolic Execution'. Flow of execution of stored procedure is given in Figure 2.

(a) **Stored Procedure Execution** We execute stored procedure with first test case. As procedure runs on the database, we record its execution through instrumentation of query nodes in PostgreSQL in 'Trace Log', which includes every step that database performs as shown in Figure 3. It also lists conditions enforced at a particular point during execution of the procedure.

(b) **Trace Analysis** We analyze collected trace line by line to extract conditions imposed on variables and tables and to store information in a corresponding state. For example in case of a variable declaration in a procedure we create symbolic object and add constraints to it. For language constructs, e.g., IF statement, we add constraint to the respective symbolic objects. For SQL constructs, we model tables. Here in this example, line 1 in 'Trace Log' corresponds to procedure call. Line 4 corresponds to sequential scan due to occurrence of SQL construct in stored procedure on line 8 of Figure 1.

(c) **Analyse Constructs** In this step, we check if we already have model for the table we move directly to step (e), otherwise we identify new table and prepare model for it in step (d). Line 4 in 'Trace Log' tells that sequential scan is done on the table `emp` where output of scan is given as TargetList which comprises of `salary` and `experience`. 'Condition' specifies constraint in SQL construct and is given in preorder notation, i.e, operatorId firstOperand secondOperand. Here condition is '$emp\_id = Param\$1$', where Param$1 is first input argument of the procedure. For language constructs, as given on line 12 of example code given in Figure 1, line 9 in 'Trace Log' given in Figure 3 is recorded. It tells us about IF condition.On encountering such constructs, we move to step (e).

Table 2: Symbolic Rows for Table EMP

empno	name	salary	experience
r11	r12	r13	r14
r21	r22	r23	r24

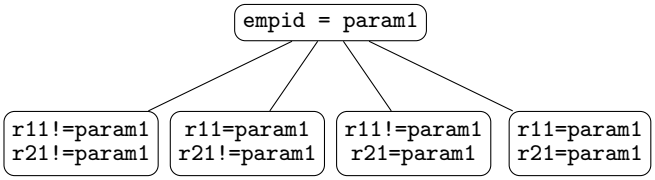

Figure 4: Possible Cases for Sequential Scan

Table 3: Data Set Against Condition for Case1
Input Value for Procedure: id = 1

empno	name	salary	experience
2	Ali	3500	7
3	Amna	3000	4

(d) **Model Table** We model table based on SQL constructs analyzed above. We query database to extract structure and integrity constraints on the table. We add two symbolic rows to our table `emp` as shown in Table 2.

(e) **Gather Constraints** For SQL constructs, we define and gather already defined constraints imposed on a table. Adding integrity constraints ensures that data generated for tables will not violate constraints imposed on the structure of the table. We consider three cases for processing of sequential scan. (i) *None of the rows match*, (ii) *Only one row matches* and (iii) *All rows match* scan condition. Case (i) and (iii) corresponds to one condition whereas case (ii) corresponds to two conditions. All four conditions in Figure 4 are constructed and added to structure 'ChoiceSet' as choices, shown in Figure 5. In addition to the conditions, we also generate and store corresponding result which satisfy conditions. This result set is a representation of output of SQL statement, e.g., for Case 1 it is empty and for Case 4 it returns both rows of table `emp` as shown in Figure 5. For language constructs, we add constraints imposed by these constructs to 'ChoiceSet'. ChoiceSets are generated lazily, i.e. choices are only available for the path under exploration at any point in time. In our example, when we reach line 3 of 'Trace Log', we define symbolic variable for 'experience' and add to the 'ChoiceSet'.

(f) **Generate Test Case and Table Data** In order to generate test data for table `emp`, our symbolic executor does not know which of the possible four paths shown in Figure 4 is taken by the procedure during execution. But we now know that value of input argument id can satisfy or dissatisfy the condition in where clause. We append this constraint to each of the choices in our 'ChoiceSet'. We pick each choice, one by one, from 'ChoiceSet' and append integrity constraints of the table as well. We use Z3 to solve these constraints to generate value of id and four different datasets for table `emp` where each dataset corresponds to one choice. In our example, dataset for Case 1 and Case 2 are given in Table 3 and Table 4 respectively. While executing line 3 of 'Trace Log', for each of the path generated by SQL construct, we append constraints of language construct to lead the execution of store procedure to within IF statement.

```
1 ChoiceSet
2 {
3 {Condition 1: Not(r_{11}==param1),
      Not(r_{21}==param1) Result: [ ]}
4 {Condition 2: (r_{11}==param1),
      Not(r_{21}==param1) Result: [r11 ]}
5 {Condition 3: Not (r_{11}==param1),
      (r_{21}==param1) Result: [r12 ]}
6 {Condition 4: (r_{11}==param1), (r_{21}==param1)
      Result: [r11, r12 ]}
7 }
```

Figure 5: ChoiceSet for Sequential Scan

Table 4: Data Set Against Condition for Case2
Input Value for Procedure: id = 1

empno	name	salary	experience
1	Ali	3500	7
3	Amna	3000	4

(g) **Update Table** We setup our tables with data generated in previous step and re-execute stored procedure with new set of inputs generated as test cases in previous step. For above example, we populate `emp` table and move to step (a) for the execution of stored procedure with new test case.

We explore remaining choices in depth first search manner by repeating steps described above until test cases for testing complete procedure are generated.

4. TECHNIQUE

Our technique for modeling of database tables is described in Section 4.1, instrumentation and processing of query plan nodes is in Section 4.2, and symbolic execution of the collected trace is in Section 4.3. Handling of data types and query nodes is discussed in Section 5 in detail.

4.1 Modeling of Database Tables

Tables in database are two dimensional objects. Each table can have multiple columns with different data types. However, the number of columns in a table is fixed whereas the number of rows is varied which makes it difficult to properly initialize the table model. Our model for the tables is a 2-D array of symbolic variables with number of rows as a configurable constant in initial model. Due to exponential increase in number of cases with increase in rows of the table, we setup initial table with two rows only. This, in our experience, covers most of the cases that can arise in execution of stored procedures. Insertions and deletions in the tables are modeled and they can change the number of symbolic rows in the table as procedure executes.

When a table is modeled, we also have to consider the constraints imposed on table by the schema. The constraints can be primary key, foreign key, unique and check constraints. The primary key constraint can be decomposed into a unique and a set of not null check constraints. We have basic underlying models for unique, foreign key and check constraints in the system. The modeling of a foreign key constraint requires addition of another table model. The foreign key constraint

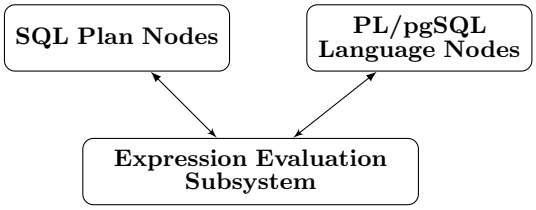

Figure 6: Structure of PostgreSQL

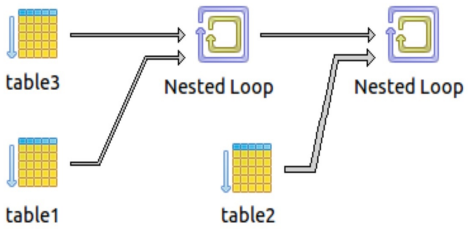

Figure 7: Plan for Join Query

can be recursive, for such cases we model chains of foreign key relations. We disable foreign key relations only if we detect a cyclic relation during the processing of foreign key constraints. We generate data by solving these constraints applied to all rows in a table model using Z3 SMT solver and populate tables during setup for procedure execution.

4.2 Processing of SQL

Execution of PL/pgSQL procedure can be divided into execution of 1) PL/pgSQL Language constructs and 2) SQL statements. Because of same type system, both rely on the same expression processing subsystem as shown in Figure 6 to process conditions and expressions. Expressions can also contain function calls allowing recursive procedure calls.

Everytime PL/pgSQL encounters an SQL statement it calls the SQL processing system to get the results. PostgreSQL prepares multiple possible execution plans with estimate of cost of executing each plan. An optimal plan represented as tree is selected for execution.

For example, *Select * from table1 t1, table2 t2, table3 t3 where t1.col1 = t3.col1 and t1.col2 = t2.num1 and t2.num2 > 2* joins three tables and places extra conditions on column of *table2*. The plan for this query is shown in Figure 7. During execution, system scans *table1* and *table3* discarding the rows that don't meet condition. Results of the two scans are joined using a nested loop based join. Latter these results are joined with output of scan of *table2*.

We treat the nodes in the plan tree as basic building blocks of the program and modeled their execution in our symbolic executor. When a sequential scan executes, we extract the table identifier, the columns that appear in the output and condition(s). Extracted conditions decide the results of the scan. In the absence of condition, all rows are selected as result. Whereas, in presence of 'WHERE' clause, we impose the conditions on the table model such that each row in a table can either satisfy or dissatisfy the scan condition.

Here we draw inspiration from numerous works on symbolic execution that treat a simple IF condition as a choice point where system can take any of the two paths depending on whether the condition or its negation is true. Here we have a larger number of conditions that are not related to each other by a simple negation. Each of the conditions, if true, has a corresponding result model containing the rows selected by that condition. Therefore, we define choice as a set of conditions with corresponding result model. 'ChoiceSet' is the set of all choices at the node. During exploration of paths, at each node, we select a choice from ChoiceSet one by one, append table integrity constraints as well as conditions from already processed nodes along this path. We then solve it to generate corresponding result. However, if solver gives no solution for the condition, it means that choice corresponding to condition is not possible. This is a possible limitation but in our evaluation we did not encounter any case where solver timed out. If solver times out, we consider path to be unreachable. When we reach a node which needs the results of the earlier node, we can easily provide symbolic models of the results processed earlier. In example above, second NestedLoop requires results from the first NestedLoop and a Sequential Scan. Since output model of first NestedLoop is compatible with general result model, plan can be symbolically modeled. This means our grammar coverage of SQL is not based on syntax, rather it is based on the underlying execution plan of SQL.

4.3 Symbolic Execution

We have instrumented PostgreSQL to record execution of PL/pgSQL, SQL and Expression Processing in a file 'Trace Log'. In our symbolic executor, State object is used to keep track of the explored and unexplored paths and it tells us about execution of the program. The algorithm for generating test cases is outlined in Algorithm 1. This algorithm explores states in a depth first search manner. The 'Trace Log' is processed line by line. Information extracted from each trace line from 'Trace Log' is stored in our data structure, called StackFrame. Each state maintains its StackFrame, which allows processing of nested queries/functions. When a trace line comes in for processing, the State object can tell if it has already processed that line before. If the trace line is not processed, then a new stack frame is prepared in State to store information about the current trace line. The function ProcessTraceLine models all kinds of operations and generate ChoiceSets and add them in State. If trace line generates only one choice which means only one path out of state, then we are certain about program flow. At this point we simply add condition to the solver and get new traceline for processing. If the ChoiceSet of State has more than one condition then we are not certain about the path taken by the current trace, so we stop analyzing trace and solve constraints to generate test cases.

On line 20, a new condition is fetched from the State. This tells us if the State has advanced (moved to next state) since the last call to NextChoice. If StateAdvanced is True, then we have a new stack frame in State and condition should be added to solver stack. Otherwise, we will pick next condition from stack frame. We explore all conditions in the stack frame before advancing to next state. This happens when Solver says that the condition is not satisfiable. We check if we are still on same trace line then it means that State has

```
Solver = Create new solver object
  // Contains the path condition stack
State = Create new state object
  // Keeps a stack of ChoiceSets for trace
T = Create a test case with random values
// Main Symbolic Execution Code
while (T is not NULL)
  Execute test case T on database
  For Line in ExecutionTrace:
    if (Line is already processed)
      continue with next loop iteration
    State.Advance() // prepare new stack frame
    ChoiceSet = ProcessTraceLine(Line)
    State.AddChoiceSet(ChoiceSet)
    if (ChoiceSet.size == 1)
      Add the condition from the only choice in
           Solver
      continue with next loop iteration
    break
  while true
    Condition, StateAdvanced = State.NextChoice()
    if Condition == NULL
      Terminate = Backtrack()
      if Terminate
        T = NULL
        break
      else
        continue with next iteration
    else if StateAdvanced
      Add condition in Solver Stack

    else
      Replace condition at Solver Top of Stack
    Solve the path condition stack in solver
    if (path condition is satisfied)
      T = Make test case from solver result
      break
```

Algorithm 1: Test Case Generation Algorithm

not advanced.In such case NextChoice function will return the second condition from the same ChoiceSet. As the new condition comes from the same ChoiceSet we need to replace the condition in the Solver stack instead of adding a new frame. The NextChoice function returns a NULL, if all choices are explored. This means that end of path is reached and we need to move back. Function BackTrack removes the StackFrame from top of stack for both State and Solver objects and continues with next loop iteration. The next iteration then gets a choice from the new stack top, leading to exploration of another path. Whenever backtracking ends up emptying the State stack, then we know that the complete tree has been explored. Backtrack function returns true for terminating the search.

From the algorithm, it is clear that we are progressing towards a full path condition by generating many cases that are based on incomplete paths through the program. We skip the processing of the trace lines already processed based on the assumption that the execution of those statements will be exactly the same. This is a reasonable assumption for PL/pgSQL language nodes but it is not completely true for SQL. SQL, being a declarative language, leaves it for the database to decide how to execute the statement through a planner instead of simply executing programmer instructions. Hence it is possible for the SQL statement to join two tables using NestedLoop algorithm in one execution and use sorting with MergeJoin algorithm in the next execution. We refer to this problem as plan instability for SQL statements. Both the plans would give the same final output but it will throw off our symbolic executor which was expecting the exact same trace till the last processed trace line. We addressed this issue by turning off the use of many algorithms that planner can use to optimize the queries, such as BitmapScan, HashAgg, HashJoin, IndexScan, IndexonlyScan and MergeJoin. This just deactivates possible planner optimizations without restricting language grammar in any case.

Table 5: Data Type Models

Data Type	Solver Type	Model Summary
Integer	Integer	Direct Mapping
Numeric	Real	Direct Mapping
Boolean	Boolean	Direct Mapping
Character	Integer	Integer represents ASCII value Restricted to A-Z, a-z, 0-9
Text	Integer	Integer is dictionary lookup value
Date	Integer	Integer is offset from base date

5. IMPLEMENTATION DETAILS

5.1 Data Type Models

Data elements in the tables and variables have specific data types. We represented these data elements as symbolic variables. Most data types don't translate directly to Z3 solver data types. Only three database types integer, numeric and Boolean map directly to corresponding solver types. We provided added support for character, text and date types by modeling them as integers because these are some of the most common data types in databases.

Characters naturally translate to integers through their ASCII values. In order to ensure that the solver does not assign invalid values to integers representing characters, a constraint is added to restrict the value of the such integers between 'a' and 'z' or'A' and 'Z' or '0' and '9'. This restricts special characters but using special characters in character type columns is not a common practice. Strings are mapped to integers. This allows for exact string matches but does not support partial matches with LIKE operation in SQL. In order to model date as an integer, we have maintained a reference date in the symbolic executor. The integer modeling of the date is an offset from this reference date. The timestamp data types are compatible with date type and we use the same model for them. Any data element in the database can be initialized to NULL. So NULL is modeled as a special integer with value -101 as it is not a common hard coded value. Data type models are summarized in Table 5.

5.2 Expression Processing Models

While processing constraints we come across a variety of expressions. The expressions that we have modeled include binary operators, Boolean operators, testing for NULL values, Coalesce expressions and Functions calls. The arguments for these operators can be table columns, variables, constants or expressions. PostgreSQL represents expressions as a tree before processing it which is printed out as a pre-order traversal in 'Trace Log'. The results of processing each type of expression is shown in the Table 6. For many cases in the

Table 6: Expressions and Table Constraint Models

Expression Type	**Expression Model**
Boolean Operator And	Result: And (Arg1 , Arg2)
Boolean Operator Or	Result: Or (Arg1 , Arg2)
Boolean Operator Not	Result: Not(Arg)
NULLTEST	Result: Arg == NULL OR Arg != NULL
Conditional Binary Operator op	Result: (Arg1 op Arg2) Condition: And(Arg1 != NULL, Arg2 != NULL)
Arithmetic Binary Operator op	Result: ExprResult Condition:Or(And(Arg1 != NULL, Arg2 != NULL, ExprResult == Arg1 op Arg2) , And(Or(Arg1 == NULL, Arg2 == NULL) , ExprResult == NULL))
Coalesce Expression	Result: ExprResult Condition: Or And(Arg1 != NULL, ExprResult == Arg1), And(Arg1 == NULL, Arg2 != NULL, ExprResult == Arg2) , ... , And(All Args == NULL, ExprResult == NULL)
PL/pgSQL Function Call	Result: FunctionResult Condition on Function Start: And(Param1 == Arg1, Param2 == Arg2, ...) Condition on Function End: FunctionResult == ActualReturnExpresion
Non PL/pgSQL Function Call	Result: FunctionResultFromModel (FunctionID, ArgumentList)
Constraint Type	**Constraint Model**
Unique Constraint (Assuming 3 rows in table Composite constraint [col1, col2])	Condition: And(Not(And(row1.col1 == row2.col1, row1.col2 == row2.col2)) , Not(And(row1.col1 == row3.col1, row1.col2 == row3.col2)) , Not(And(row2.col1 == row3.col1, row2.col2 == row3.col2)))
Foreign Key Constraint (Assuming table1 = 1 row, table2 = 2 rows, table1.col2 referencing table2.col1)	Condition: Or(table1.row1.col2 == table2.row1.col1 , table1.row1.col2 == table2.row2.col1 , table1.row1.col2 == NULL)

table, we have specified processing result expression with a condition. Result is the expression that is sent back to the main trace processor. Main trace processor use the result according to its needs, e.g, IF condition trace line processing would use it as the decision condition and generate two choices from it; the condition itself and its negation. Whereas an assignment operation trace line processing will use it to assert that the target variable is equal to this expression value. The condition expression represents the conditions that are automatically added with every choice by the trace processor using the result expression.

5.2.1 Boolean Operators, Binary Operations and Coalesce Expression

All three Boolean operations and binary conditional operators map directly to the Z3 solver API. We have two types of NULLTESTs. ISNULL and ISNOTNULL test. Both return true when the value is null and not NULL respectively. Result expressions for ISNULL and ISNOTNULL as in Table 6 are returned. In binary arithmetic operators, for NULL arguments, result of the expression is also NULL. For the arithmetic operation, a new symbolic variable is created and returned as a result. Conditions are imposed on the new symbolic variable. We have two conditions inside the OR function. Both the conditions are mutually exclusive because of the NULL checks in each condition i.e. if both arguments are not NULL then the OR condition inside the second AND can never be true. Therefore the solver has to impose *ExprResult == Arg1 op Arg2* in order to get a satisfying assignment. Similarly, if any of the arguments is NULL then the OR condition will be true but the first AND can never be true so in order to get a satisfying assignment *ExprResult == NULL* must be imposed. A coalesce expression takes multiple inputs and returns its first NOT NULL result. Like arithmetic operators the result is a new symbolic variable whose value comes from a set of mutually exclusive constraints.

5.2.2 Function Calls

We classify the function calls according to the function language. Execution of functions written in PL/pgSQL is printed in our trace and we can follow the path through these functions. For function calls appearing in an expression we simply return a new symbolic variable representing its result value with no condition imposed on it. We also generate a condition that assigns the values to the function input variables. We call this condition Call Condition for the function and it links the variables in the current function with the variables in the new context. Reaching trace line of the expression containing function calls, we get the trace lines for execution of the functions listed in that expression. Call condition is loaded into solver when we get trace for the start of the function. At the end of the function, the value returned by the function is stored in result variable created earlier. Implementing this functionality requires tracking the call stack in the symbolic executor. The maximum depth of the call stack for the symbolic executor is a configurable constant.

5.2.3 Special Functions and Sequences

Functions not written in PL/pgSQL don't show execution details in the trace. These functions includes SQL built-in functions like the "nextval" for sequences, date and time and type conversion. Model for the "nextval" function relies on our model for sequences to output a symbolic expression. We model the sequence object as symbolic variable of type integer with starting value as 0. We return the expression

start_value + integer, whenever nextval function is called for this object. The model for current date and time function simply returns 0. That means we are treating the base date of our symbolic executor as current date. Another function that we have modeled takes in the sequence name and returns the sequence id in database. To model this function we query database for the relevant information and return it as model output.

5.3 Constraint Models

We model check, unique, foreign key and primary key(NOT NULL check and unique) constraints. Check constraints are translated into pre-order traversed tree expressions and processed in same way to get conditions. The conditions are added to the solver for each row in the table(s). The models for foreign key constraints and unique constraints are shown in Table 6.

For unique constraint in Table 6, the condition is generated for a composite constraint on *col1* and *col2*. Composite constraint means that value of both columns taken together must be unique. In general the condition needs to be asserted for all row combinations. Table 6 also shows foreign key constraint model. Column value has to be one of the values in the referenced column or it needs to be NULL.

5.4 SQL Plan Node Processing

We support SequentialScan, NestedLoop, ResultNode and ModifyTable nodes. In previous section we discussed first two types of nodes. ResultNode is used to generate a single row of results. The values in the row being generated can be variables, constants or any expression containing variables and constants.We treat results of the expressions as symbolic element in a new symbolic row that ResultNode is supposed to produce. The conditions from processing of multiple expressions are appended to generate a single condition. The row produced by the expression results by solving conditions is the result for this node.

ModifyTable is top plan node and supports inserts, updates and deletes. Thus it relies on its child node to provide the data it needs to perform its job. For DELETE operation, this node needs a list of row identifiers of the rows that need to be deleted. For UPDATE operation it need the new rows along with the row identifiers of the rows which the new rows are supposed to replace. For INSERT operations, the new row must be the output of the child plan. The simple case, where we specify the list of values to be inserted, is covered by the ResultNode, which makes the row out of the values. In more complicated cases, where we are inserting the result of a query into a table, we simply have the whole query plan as the child plan for ModifyTable. In all three cases, we check that the table modifications do not violate the table constraints by creating constraint conditions on the modeled table and adding the result as a choice. We get our second choice as negation of the first choice which is responsible for generating cases which violate constraints.

5.5 PL/pgSQL Construct Processing

The PL/pgSQL language constructs such as IF condition, FOR loops over SQL query results, assignment statement, variable initialization from SQL results, function start, function return statement are supported. IF statement can be a simple expression or it can have a complicated condition with an SQL statement embedded in it. For the simple case, the condition for the choice can be obtained from the expression processor where as in later case, the result comes from the SQL processing. Consequently, generating cases for the SQL is the way to explore the possible directions the code can take from the IF condition.

Assignment statements usually generate only one choice i.e *target variable == expression result*. Another kind of assignment occurs with the result of the SQL queries. The keyword *into* allows the SQL statements in PL/pgSQL procedures to directly assign their result values to variables. This is handled similarly except for the fact that multiple variables can be simultaneously assigned new values. In both cases these nodes have ChoiceSets of size 1.

Function start and function return statements are part of the support for the PL/pgSQL function calls. The ChoiceSets for these are described in Table 6. We have also added support for FOR loop over SELECT statements. The statement at the start of the FOR loop acts like an assignment statement and assigns a row from the FOR loop SELECT query results to the variables on which the loop runs. FOR loop works with currently modeled data types.

Table 7: Exception Cases

Total Exception Cases	93
No data found in SELECT	Over 50
Exceptions due to sequence reset	23
Constraint violation due to unchecked inputs	10
User defined exceptions on input validation	4

6. EVALUATION

We evaluated our technique on an open source Accounting and Enterprise Resource Planning (ERP) system, PostBooks which has a significant amount of its business logic written in functions in the database. It has a schema consisting of 251 tables. Functionality of 151 procedures is fully supported by our models. Manual inspection of some of these procedures indicated that 100% branch coverage was achieved. The symbolic execution was configured with the stack depth of 5, result size of 2 excluding unconditional scans and cross joins, and an initial table size of 2 rows each.

All experiments were performed on a 1.9Ghz Dell i7 machine. Many of the procedures in PostBooks have user defined exceptions to give a user friendly response to the client. During exploration of the procedures, we automatically generate cases that drive the procedure execution towards such exceptions, e.g. schema constraint violations. We found 93 test cases that trigger user defined exceptions or constraints violations.

The scalability of our technique is shown by the number of modeled tables for many stored procedures. PostBooks uses a very large number of constraints to ensure data integrity. In particular, the schema has over 400 foreign key constraints. This means that long chains of tables related by foreign keys are common. Even if a procedure directly uses a few tables, we had to model the tables it references to be able to set up data properly for execution of the procedure. In 71 procedures our symbolic executor ended up modeling over 30 tables and the constraints associated with them. Usually time taken for symbolic execution of stored procedure depends on number of tables being modeled.

Now we will analyze the nature of the exception cases found. Breakup of 93 exception cases found in the fully

```
1  CREATE OR REPLACE FUNCTION attachcontact(integer,
       integer)
2  DECLARE
3      pcntctId ALIAS FOR $1;
4      pcrmacctId ALIAS FOR $2;
5    BEGIN
6      UPDATE cntct SET cntct_crmacct_id = pcrmacctId
7      WHERE cntct_id = pcntctId;
8    ...
9  END;
```

Figure 8: Code for NOT NULL Constraint Violation with Test Case `attachcontact(2, NULL)`

```
1  CREATE OR REPLACE FUNCTION createcrmacct(integer,
       integer,...)
2  DECLARE
3  _crmacctid ALIAS FOR $1;
4   _custid      ALIAS FOR $2;
5  ...
6  BEGIN
7   INSERT INTO crmacct (crmacct_id,...,
        crmacct_cust_id,..., crmacct_prospect_id,
        crmacct_taxauth_id,...)
8    VALUES (_crmacctid,..., _custid,...,_prospectid,
         _taxauthid,...);
9  ...
```

Figure 9: Code for Foreign Key Constraint Violation with Test Case `createcrmacct(2,...4,...,8,6,...)`

explored procedures is listed in Table 7. Of these 93 cases, over 50 are user defined exceptions. Manual inspection of many procedures and cases involved in these exceptions indicates that majority of them correspond to no data found cases for SELECT statements. The typical scenario for these exceptions is the case where a select statement tries to fetch configuration data. For these cases, we observed that the programmer has raised exceptions in the code to give the client a meaningful message about the missing data. Another class of user defined exceptions is based on validation of input values. Exceptions are raised to notify the user exactly which input is incorrect.

We have 23 cases of primary key constraint violations. This is because most of the tables in PostgreSQL have sequences linked to the primary key columns as default values. In this situation, the only way a primary key violation can occur is when the programmer specifies the value of the primary key column himself or when the sequence is reset. From manual inspection of the procedure source we know that there is no mention of any value overriding the primary key column's default value in the procedure source. So the constraint violation is triggered by resetting of the sequence. This is something that we have allowed in our sequence model because accidental sequence reset is a common problem during an implementation phase of ERP systems in our experience resulting in a buggy behavior. In 10 other cases, the symbolic executor found cases that violate NOT NULL constraint during UPDATE statements. Inspection of the code indicated that the UPDATE statements in these functions are directly using some of the input values of the procedure allowing the SMT solver to set them to any value to trigger a constraint violation.

Just like assertions provide an Oracle for test generation of normal programs, user defined exceptions and constraint violations provide an oracle against which our technique can be used to automatically generate valid test cases. Our technique will also generate other valid and important test cases whose results should be verified by the application programmer to find if they are correct or they identify a mistake or missing exception handling in the stored procedure itself. Our test cases are useful as they all cover unique scenarios in stored procedure execution using symbolic execution.

In order to evaluate effectiveness of constraint models, we injects faults in code and execute our symbolic executor. Sample code for `attachcontact` is given in Figure 8 . There is NOT NULL check constraint on field 'cntct_crmacct_id' of table cntct. Our symbolic executor generates a test case `attachcontact(2, NULL)` for check constraint violation. On reaching line 6 of the code, we will get exception. In other stored procedure `createcrmacct` in Figure 9 value of crmacct_cust_id is a reference field, i.e., foreign key from customer table. Test case `createcrmacct(2,...4,...,8,6,...)` generated by our symbolic executor will throw an exception for violation of foreign key constraint.

7. RELATED WORK

Symbolic Execution. Clarke [8] and King [19] pioneered traditional symbolic execution for imperative programs with primitive types. Much progress has been made on symbolic execution during the last decade. PREfix [3] is among the first systems to show the bug finding ability of symbolic execution on real code. Generalized symbolic execution [18] defines symbolic execution for object-oriented code and uses *lazy initialization* to handle pointer aliasing.

DART [15] combines concrete and symbolic execution to collect the branch conditions along the execution path. DART negates the last branch condition to construct a new path condition that can drive the function to execute on another path. DART focuses only on path conditions involving integers. To overcome the path explosion in large programs, SMART [14] introduced inter-procedural static analysis techniques to compute procedure summaries and reduce the paths to be explored by DART. CUTE [26] extends DART to handle constraints on references.

EGT [5] and EXE [6] also use the negation of branch predicates and symbolic execution to generate test cases. They increase the precision of symbolic pointer analysis to handle pointer arithmetic and bit-level memory locations. KLEE [4] is the most recent tool from the EGT/EXE family. KLEE has been shown to work for many off the shelf programs written in C/C++. Many recent research projects have proposed techniques for scaling symbolic execution by parallel and incremental execution [28, 32, 30, 29, 24, 36].

Testing Database Applications. The testing of stored procedures is closely related to the testing of database driven applications written in imperative languages. While we have not found any previous work on automated test case generation for stored procedures, the testing of database driven applications has received significant attention in the past decade.

Binnig et al. [2] introduced reverse relational algebra for generating a test case given an SQL SELECT statement and a desired output. In general, test oracles like this are not available. Further, a given output exercises one particular behavior of an SQL statement whereas our work is concerned

with exhausting many possible behaviors under some bounds. Veanes et al. [34] modeled SQL queries as constraints and used SMT solvers to generate database table data. They also need some specifications of the required output like the result should be empty or result should have a specified number of rows. While using SMT solvers to improve the analysis, they are also concerned with finding one particular database state for one particular query. Tuya et al. [33] introduced a new coverage criterion for testing of SQL statements considering semantics of multiple SQL constructs. They later [9] worked on a constraint based approach to generate test cases for SQL queries that satisfy their proposed criteria. The criteria was written in Alloy [16] i.e. required additional specifications and was focused on single SQL statements. In contrast to the above approaches, our approach is focused on complete stored procedures analysis and does not need any additional specifications.

One of the first tools for automated test case generation of database driven application was AGENDA [11]. AGENDA generates test cases for transactions in applications by considering the database schema constraints. To model the conditions imposed by the transaction logic, it relies on user supplied constraints and is focused on specific kinds of tests. As far as we know, Emmi et al. [13] were the first ones to apply the idea of symbolic execution to database driven applications. They used concolic execution and used two constraint solvers to obtain the test cases. First constraint solver was used to solve arithmetic constraints while other was specialized to solve string constraints. They were able to support partial string matches that are expressed in SQL with LIKE keyword. Although they supported a wide variety of constraints that appear in the WHERE clause, their supported SQL grammar was limited to queries using a single table unlike our work which supports joins with any number of tables. Emmi et al. designed their symbolic executor to maximize branch coverage in the code. Li et al. [20] and Pan et al. [23] extended their approach for different coverage criterion. Our work is different from the above symbolic approaches as it is hosted inside the database and is able to apply symbolic analysis at a finer granularity and is therefore able to generate test cases for complete stored procedures.

Marcozzi et al. [21] proposed an algorithm for testing control flow graph of Java code interacting with the database. It generates Alloy [16] relational model constraints for a given database schema, a finite set of paths from the control flow graph, and variables along those paths (both method variables and those used in SQL queries). It generates a symbolic variable for each value taken by the method variables or database tables during path exploration. The Alloy model generated ensures the execution of the path that involves these symbolic variables. Alloy Analyzer solves these constrains to generate test cases. In later work [22], they described how an SMT solver can be potentially used to model the constraints and generate test cases. This would make analyzing larger applications possible. However, this is an idea paper and they have not implemented or evaluated it. Potentially such a technique would face the same hurdles as other approaches implemented outside the database management system. In our work, because of implementing it at the query engine level, we are able to implement and evaluate complex queries like multi-table joins which previous techniques are unable to handle.

Khalek and Khurshid [17] presented a framework that uses Alloy [16] to model a subset of SQL queries by automatically generating SQL queries, database state and expected results of queries when executed on a database management system. They have modeled SQL queries and database schema using Alloy which used SAT solver to populate tables. The focus of their work is testing the correctness of database management system itself and not of the applications using them.

A common approach in many of the above techniques is using declarative specifications in Alloy [16] and using the Alloy Analyzer to solve them. The solutions are often converted back automatically to INSERT queries that can populate a database. While Alloy is a powerful language, converting imperative constraints like those which are mixed with queries in a stored procedure are difficult to model, resulting in a substantially reduced SQL subset being modeled. In contrast, our technique of instrumenting the query nodes in the database query execution engine results in both declarative queries and imperative constraints converted into a series of imperative sequential tasks on which standard symbolic execution techniques can be applied. While we do not support the entire SQL grammar, our limitations are not fundamental in nature and the technique can be easily extended to other SQL statements.

Commercial IDEs like Visual Studio[3] have support for unit testing stored procedures. However, this support is limited to automatically filling databases, executing the procedure, and comparing output. The database values and procedure inputs are not automatically generated. There is also work on preventing SQL injection attacks in stored procedures [35]. They combine static analysis to instrument SQL statements in stored procedures and a dynamic part to compare the statements to what was observed statically. However, this technique is specific to SQL injection and cannot be extended to generic test case generation.

8. CONCLUSIONS AND FUTURE WORK

We presented a novel approach of applying symbolic execution to automatically generate test cases for stored procedures. We instrumented the internal execution plans generated by PostgreSQL database management system to extract constraints and used the Z3 SAT solver to generate test cases consisting of table data and procedure inputs. We treated values in database tables as symbolic, modeled the constraints on data imposed by the schema and by the SQL statements executed by the stored procedure, and used a SMT solver to find values that will drive the stored procedure on a particular execution path.

We showed in our evaluation on more than a hundred stored procedures from a large business application that this technique can generate many useful test cases and also uncover bugs that lead to schema constraint violations or user defined exceptions.

In future, we plan to extend our technique to handle the remaining query node types and release our tool for end-to-end automated testing of PostgreSQL procedures. In addition, we intend to make seamless symbolic execution from Java applications to PostgreSQL stored procedures that can share a symbolic map. We are also working on a technique to automatically generate larger tables if it enables a particular path exploration in code to be executed.

[3]https://www.visualstudio.com

9. REFERENCES

[1] C. Barrett and C. Tinelli. CVC3. In *Proc. 19th International Conference on Computer Aided Verification (CAV)*, pages 298–302, 2007.

[2] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *IEEE 23rd International Conference on Data Engineering (ICDE)*, pages 506–515, 2007.

[3] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software Practice Experience*, 30(7):775–802, June 2000.

[4] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[5] C. Cadar and D. Engler. Execution Generated Test Cases: How to make systems code crash itself. In *Proc. International SPIN Workshop on Model Checking of Software*, pages 2–23, 2005.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *Proc. 13th Conference on Computer and Communications Security (CCS)*, pages 322–335, 2006.

[7] L. A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering (TSE)*, 2(3):215–222, May 1976.

[8] L. A. Clarke. *Test Data Generation and Symbolic Execution of Programs as an aid to Program Validation.* PhD thesis, University of Colorado at Boulder, 1976.

[9] C. De La Riva, M. J. Suárez-Cabal, and J. Tuya. Constraint-based test database generation for SQL queries. In *Proceedings of the 5th Workshop on Automation of Software Testing*, pages 67–74, 2010.

[10] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[11] Y. Deng, P. Frankl, and D. Chays. Testing database transactions with AGENDA. In *Proceedings of the 27th international conference on Software engineering*, pages 78–87, 2005.

[12] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based Repair of Complex Data Structures. In *Proc. 22nd International Conference on Automated Software Engineering (ASE)*, pages 64–73, 2007.

[13] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 151–162, 2007.

[14] P. Godefroid. Compositional Dynamic Test Generation. In *Proc. 34th Symposium on Principles of Programming Languages (POPL)*, pages 47–54, 2007.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proc. 2005 Conference on Programming Languages Design and Implementation (PLDI)*, pages 213–223, 2005.

[16] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[17] S. A. Khalek and S. Khurshid. Systematic testing of database engines using a relational constraint solver. In *Proceedings of the Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 50–59, 2011.

[18] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 553–568, 2003.

[19] J. C. King. Symbolic Execution and Program Testing. *Communications ACM*, 19(7):385–394, July 1976.

[20] C. Li and C. Csallner. Dynamic symbolic database application testing. In *Proceedings of the Third International Workshop on Testing Database Systems (DBTest)*, 2010.

[21] M. Marcozzi, W. Vanhoof, and J.-L. Hainaut. A relational symbolic execution algorithm for constraint-based testing of database programs. In *IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 179–188, 2013.

[22] M. Marcozzi, W. Vanhoof, and J.-L. Hainaut. Towards testing of full-scale SQL applications using relational symbolic execution. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, pages 12–17, 2014.

[23] K. Pan, X. Wu, and T. Xie. Database state generation via dynamic symbolic execution for coverage criteria. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, page 4, 2011.

[24] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed Incremental Symbolic Execution. In *Proc. 2011 Conference on Programming Languages Design and Implementation (PLDI)*, pages 504–515, 2011.

[25] D. A. Ramos and D. R. Engler. Practical, Low-Effort Equivalence Verification of Real Code. In *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, pages 669–685, 2011.

[26] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proc. 5th joint meeting of the European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005.

[27] C. Seo, S. Malek, and N. Medvidovic. Component-Level Energy Consumption Estimation for Distributed Java-Based Software Systems. In *Proc. 11th International Symposium on Component-Based Software Engineering*, pages 97–113, 2008.

[28] J. H. Siddiqui and S. Khurshid. ParSym: Parallel Symbolic Execution. In *Proc. 2nd International Conference on Software Technology and Engineering (ICSTE)*, pages V1: 405–409, 2010.

[29] J. H. Siddiqui and S. Khurshid. Scaling Symbolic Execution using Ranged Analysis. In *Proc. 27th Annual Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2012.

[30] J. H. Siddiqui and S. Khurshid. Staged Symbolic Execution. In *Proc. 27th Symposium on Applied Computing (SAC): Software Verification and Testing Track (SVT)*, 2012.

[31] N. Sörensson and N. Een. An Extensible SAT-solver. In *Proc. 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, 2003.

[32] M. Staats and C. Păsăreanu. Parallel Symbolic Execution for Structural Test Generation. In *Proc. 19th International Symposium on Software Testing and Analysis (ISSTA)*, pages 183–194, 2010.

[33] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva. Full predicate coverage for testing SQL database queries. *Journal of Software Testing, Verification and Reliability*, 20(3):237–288, 2010.

[34] M. Veanes, P. Grigorenko, P. De Halleux, and N. Tillmann. Symbolic query exploration. In *Formal Methods and Software Engineering*, pages 49–68. Springer, 2009.

[35] K. Wei, M. Muthuprasanna, and S. Kothari. Preventing sql injection attacks in stored procedures. In *Proceedings of the Australian Software Engineering Conference (ASWEC)*, pages 191–198, 2006.

[36] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized Symbolic Execution. In *Proc. 2012 International Symposium on Software Testing and Analysis (ISSTA)*, ISSTA 2012, pages 144–154, 2012.