Testing Database Programs using Relational Symbolic Execution

Marcozzi, Michaël; Vanhoof, Wim; Hainaut, Jean-Luc

Publication date:
2014

Document Version
Early version, also known as pre-print

Testing Database Programs using Relational Symbolic Execution

Michaël Marcozzi[1], Wim Vanhoof, Jean-Luc Hainaut

Faculty of Computer Science
University of Namur
Rue Grandgagnage, 21
5000 Namur, Belgium

Abstract

Symbolic execution is a technique which allows to automatically generate test inputs (and outputs) exercising a set of execution paths within a program to be tested. If the paths cover a sufficient part of the code under test, the test data offer a representative view of the program's actual behaviour, allowing to detect failures and correct faults. Relational databases are ubiquitous in software, but symbolic execution of the programs that manipulate them remains a non-trivial problem, because of the complex structure of such databases and the complex behaviour of the SQL statements. In this work, we define a symbolic execution for SQL code and integrate it with a more traditional symbolic execution of normal program code. The database tables are represented by relational symbols and the SQL statements by relational constraints over these symbols and the symbols representing the normal variables of the program. An algorithm based on these principles is presented for the symbolic execution of simple Java methods, reading and writing with transactional SQL in a relational database, the latter subject to data integrity constraints. The algorithm is integrated in a test data generation tool and experimentally evaluated over thousands of program paths including real application code. The target language for the constraints produced by the tool is the SMTv2 standard and the used solver is Microsoft Z3. The results show that the proposed approach allow to generate meaningful test data for database programs, including valid database content, in reasonable time.

Keywords: Software Testing, Symbolic Execution, Constraints, Satisfiability Modulo Theories (SMT), Quantifiers, First-Order Logic, Relational Algebra, Structured Query Language (SQL), Database Applications

☆

Email addresses: michael.marcozzi@unamur.be (Michaël Marcozzi), wim.vanhoof@unamur.be (Wim Vanhoof), jean-luc.hainaut@unamur.be (Jean-Luc Hainaut)

[1]F.R.S.-FNRS Research Fellow

1. Introduction

In current software development practice, testing [1, 2] remains the primary approach to improve the reliability of software. This motivates [3] much research on efficient techniques to automate all aspects of the software testing process. Of particular interest is the automation of *test data generation* [4] for functional testing of units of code, where the idea is to automatically generate a representative set of inputs (and outputs) for a program fragment under test (typically a function or method). These data can subsequently be compared with the function's expected behaviour in order to detect failures and correct faults. Moreover, once a suitable set of test data has been generated (and verified), it can be used as reference data for continued (regression) testing of the code.

While different approaches exist towards the automatic generation of test data, *symbolic execution* [5] has been recognised as a promising technique for so-called *white-box* or *glass-box* testing [6, 7, 8, 9, 10], where the idea is to generate input data that in some way *cover* a sufficiently large part of the control-flow graph of the function under test [4]. The symbolic execution process traverses the control-flow graph of a program or function by executing the code over symbolic values instead of concrete values [5]. Each time a control dependency is encountered, the symbolic execution process proceeds along one of the possible paths, thereby generating constrains upon the symbolic values such that when the program variables have values that satisfy these constraints, the real execution would proceed along the selected path. The process terminates when a sufficiently large (and diverse) number of paths through the control-flow graph have been explored according to some coverage criterion [4]. For each path, the constraints that have been collected along are regrouped in a so-called *path-constraint* which is subsequently solved, resulting in a set of concrete test values for the program variables that make the real execution proceed along the given path. Test data generation based on symbolic execution is now at the core of various popular open-source and commercial testing tools [9].

Although test data generation techniques are maturing, barely few works [11, 12, 13] have studied how to automate the generation of representative test data for programs that interact with a database, i.e., programs that intensively read and write into a large, persistent, and highly-structured relational database [14] using SQL statements. In this work, we detail, formalise and evaluate an approach that defines a symbolic execution for SQL code and integrates it with a more traditional symbolic execution of typical program code. The data generated by our technique constitutes test data that includes database content in addition to values for the program variables. Consequently, it can serve to test the program at hand, including the interaction between the code and the relational database.

Enabling test data generation for database intensive programs is a non-trivial extension of known data generation techniques. In a database program, the database can be seen as a particular kind of container for some of the values manipulated by the program. During symbolic execution, these values must thus be represented symbolically and subsequently constrained to allow the proper generation of test data, including an input and output database content. Nevertheless, the symbolic representation of these values raises difficulties as the database is a container of particularly complex shape: its content must obey the so-called *database schema*, defined using SQL DDL code [14]. This database schema defines a set of tables where the database content will be stored. Each table can be seen as representing a mathematical relation, i.e., a set of tuples with no limit on the

potential number of tuples. The schema typically also describes a set of *data integrity constraints* that must be enforced by the relations in the tables, like the primary key, foreign key or check constraints. These constraints are particularly complex as they are first-order logic constraints. For instance, a primary key constraint states that for all couples of tuples in the relation represented by a table, the value of the primary key fields cannot be all equal.

A program typically interacts with the database by using SQL [14] *query statements* and *DML statements* that are embedded in the program's source code. As such, SELECT queries allow to gather values from the database tables in order to copy them into the program variables. DML statements INSERT, UPDATE and DELETE allow to modify the content of the database tables, typically in function of the value of the program variables. Symbolic execution of such SQL statements raises difficulties due to their complex behavior. Firstly, SQL is a declarative language: SQL statements express the desired action over the content of the database relations, but they do not explicit the (often complex) control-flow necessary to compute this action. In practice, during execution, these SQL statements are sent by the database program – using a dedicated API – to the DataBase Management System (DBMS) [14], an external component responsible for the interpretation and execution of SQL code over the database. The DBMS keeps an optimised and persistent internal representation of the database and manages concurrent distant accesses by allowing the database programs to use *transactions* [14]. Secondly, the execution of DML INSERT, UPDATE or DELETE statements by the DBMS does not only consist in modifying the content of the database, but also in checking that these modifications do not let the database in a state where the integrity constraints defined in the schema are violated. If an integrity constraint violation is detected, the DML statement execution fails and the database remains unmodified. As a consequence, during symbolic execution each INSERT, UPDATE or DELETE statement will have to be treated as an if-then-else statement with a particularly complex condition:

```
if (Program variables and Database are in a state
    where the SQL statement will not violate any constraint) {
        Execute the SQL statement!
} else {
        Signal a constraint violation!
}
```

The technique that we detail in this work overcomes these difficulties by modelling every table in the database as a variable typed as a mathematical relation over simple domains, like the integers. Each SQL statement in the program can then be modelled as a relational operation over these *relational variables* as well as the traditional program variables. By defining a symbolic execution over this relational version of the program, we can derive a set of path constraints over the values of both the program's relational and traditional variables. The generated path constraints will thus include symbols representing simple program values as well as symbols representing the relations in the database tables. Furthermore, each path constraint must be combined with the schema constraints in order enforce data integrity of the database content. The result is a complex constraint system that mixes traditional constraints on the program's variables with relational constraints over the relational variables. Each solution to the combined constraint system describes

test data, *including* a valid initial and final state for each table in the database, such that when the program is executed with respect to these data values (including the database), the execution will follow the path represented by the constraint system. In our work, we use the SMTv2 [15] language logic to express these combined constraint systems, in a similar way to the scheme proposed in [16] for relational constraints modelling. The constraints are solved using the Z3 [17] solver, which is at the core of several existing symbolic execution tools (e.g. [18, 19]).

The main contribution of this work is a relational symbolic execution algorithm for simple database programs. This algorithm integrates the technique described in the previous paragraphs for symbolic execution of SQL code with a classical symbolic execution process for basic Java statements. This algorithm can be used for symbolic execution of a language composed of simple Java methods interacting with a relational database using SQL statements and transactions through JDBC. Given the SQL DDL code describing the database schema, the Java/SQL code of the method and a finite path in the control flow graph of this method, the algorithm generates the corresponding constraints in the SMTv2 language. A test generation tool based on this algorithm has been coded and used to generate test data for a number of sample Java methods and databases.

This work extends our previous work presented in [20, 21]. It notably adds the support of SMTv2 as output constraint language. This allows the technique to benefit from the power of SMT solvers for relational constraint solving [16]. It also provides an extended experimental evaluation of the technique over thousands of paths in various pieces of code, including real application code.

The remainder of this paper is organised as follows. Our relational symbolic execution algorithm is described in section 2. First, we formally define the part of the Java/SQL syntax that is supported by the algorithm. Then, we systematically describe the constraint generation rules to be used for the symbolic execution of this sub-language. A test generation tool based on the algorithm is described and evaluated experimentally over a set of sample programs in section 3. Finally, some conclusions, related and future work are discussed in section 4.

2. A relational symbolic execution algorithm for simple Java/SQL programs

2.1. Syntax of the tested Java/SQL programs

In this section, we define precisely – using a BNF grammar – what subset of the Java/SQL syntax our algorithm can execute symbolically. This subset has been chosen to offer a good compromise between simplicity and expressiveness. Without loss of generality, we consider a single java method that interacts with a relational database. The database schema describes a set of tables. Each table has a primary key and the attributes can be constrained by foreign key constraints and check constraints. The code of the method can contain if-then-else blocks, while loops, return statements and local variable assignments with typical operators for lists and integers. The method will interact with the database through SQL base statements, as they are typically used in online transaction processing (OLTP) programs, and through SQL base primitives for transaction management. The method receives as input parameters a JDBC connection to the database, a set of integer lists and an input scanner for integers. The lists model any structured group of inputs transmitted to the code at method call. The scanner models the method's access to simple data from the "outside world", like user prompt, network access, etc.

Figure 1 provides an example database program in this particular sublanguage. The example describes a database with two tables: one for library shelves and one for the books stored in each of these shelves. The total number of books stored in a shelf is saved for each shelf. The example also describes a method manipulating this database: it adds a set of new books to the database and updates the shelves' books counts. If a book is added to a non-existent shelf, then the shelf is itself added to the database as well. The books are inserted one by one in isolated transactions. If a transaction was successful, the code of the added book is saved in a list, which is returned at the end of the method's execution.

In the next paragraphs, the chosen notation for the BNF grammar of the syntax is standard but includes some additional meta-symbols: {...} (grouping), $^?$ (zero or one times), * (zero or more times) and $^+$ (one or more times). When a single nonterminal appears several times in a single production, subscript notation allows to distinguish between the occurrences.

2.1.1. Database program

A database program is composed of the SQL DDL code of the database schema and of the code of the Java method under test.

⟨*database-program*⟩ ::= ⟨*sql-ddl*⟩ ⟨*java-method*⟩

2.1.2. Database schema

The relational database schema is a list of table definitions. This list can be empty, in what case the program is a traditional program that works independently of any database. In the list, each table is identified by its name, contains at least one attribute and endorses exactly one primary key. Foreign keys and additional check constraints can be declared for a table. A row in a table cannot be deleted or see its primary key value modified as long as there exists at least another row in the database that references it (ON DELETE/UPDATE NO ACTION). Semantics of all the schema creation primitives conforms to the classical SQL DDL specification provided by ISO.

Figure 1: SQL DDL and Java/SQL DML code of a database program adding books and updating shelves' books count in a library database.

```
CREATE TABLE shelf (
 id INTEGER NOT NULL,
 numberOfBooks INTEGER NOT NULL,
 CONSTRAINT sPK PRIMARY KEY (id),
 CHECK(numberOfBooks > 0));
```

```
CREATE TABLE book (
 code INTEGER NOT NULL,
 shelfId INTEGER NOT NULL,
 CONSTRAINT bPK PRIMARY KEY (code),
 CONSTRAINT bFK FOREIGN KEY(shelfId)
                REFERENCES shelf (id));
```

```
1  List<Integer> addBooks (Connection con, Scanner in, List<Integer> newBooks) throws SQLException {
2   int i = 0;  List<Integer> addedBooks = new ArrayList<Integer>();
3   while ( !(newBooks == null) & (i < newBooks.size()) ) {
4     int error = 0;  int theShelf = in.nextInt();
5     ResultSet shelves = con.createStatement().executeQuery("SELECT id FROM shelf WHERE id="+theShelf);
6     if ( ! shelves.next() )
7       con.createStatement().execute("INSERT INTO shelf VALUES ("+theShelf+",1)");
8     else
9       con.createStatement().execute("UPDATE shelf SET numberOfBooks=numberOfBooks+1 WHERE id ="+theShelf);
10    try {
11      con.createStatement().execute("INSERT INTO book VALUES ("+newBooks.get(i)+","+theShelf+")");
12    } catch (SQLException e) {
13      error = 1;
14    };
15    if (error==0) {
16      con.commit();
17      addedBooks.add(newBooks.get(i));
18    } else
19      con.rollback();
20    i = i + 1;
21  };
22  return addedBooks; }
```

⟨*sql-ddl*⟩ ::= ⟨*table*⟩*

⟨*table*⟩ ::= `CREATE TABLE` ⟨*id*⟩ `(`⟨*att*⟩$^+$ ⟨*p-key*⟩ ⟨*f-key*⟩* ⟨*chk*⟩*`);`

⟨*att*⟩ ::= ⟨*id*⟩ `INTEGER NOT NULL ,`

⟨*p-key*⟩ ::= `CONSTRAINT` ⟨*id*⟩$_{cst}$ `PRIMARY KEY (` ⟨*id*⟩$_{att}$ `)`

⟨*f-key*⟩ ::= `,CONSTRAINT` ⟨*id*⟩$_{cst}$ `FOREIGN KEY (` ⟨*id*⟩$_{att}$ `) REFERENCES` ⟨*id*⟩$_{tab}$ `(` ⟨*id*⟩$_{refid}$ `)`

⟨*chk*⟩ ::= `,CHECK (`⟨*id*⟩ {`<` | `=` | `>`} ⟨*integer*⟩`)`

⟨*id*⟩ ::= {a |...| z | A |...| Z}{a |...| z | A |...| Z | 0 |...| 9}*

⟨*integer*⟩ ::= -$^?$ {1 | ... | 9}{0 | ... | 9}* | 0}

2.1.3. Method signature and body

We consider simple Java methods manipulating only internal variables and parameters. Variables can only be typed as '`int`', '`java.util.List<java.lang.Integer>`' or '`java.sql.ResultSet`'. The method receives as input parameters a connexion to the database (typed as '`java.sql.Connection`'), a scanner (typed as '`java.util.Scanner`') and some lists of integers (typed as '`java.util.List<java.lang.Integer>`'), where two distinct list parameters cannot reference a single list object. Its return type can be either '`void`', '`int`' or '`java.util.List<java.lang.Integer>`'.

⟨*java-method*⟩ ::= ⟨*type*⟩ ⟨*id*⟩ `(`⟨*db-con*⟩`,`⟨*inp*⟩ ⟨*parameters*⟩`) throws SQLException {` ⟨*stmt*⟩* `}`

⟨*type*⟩ ::= `void`
| `int`
| `List<Integer>`

⟨*db-con*⟩ ::= `Connection con`

⟨*inp*⟩ ::= `Scanner in`

⟨*parameters*⟩ ::= {`, List<Integer>` ⟨*id*⟩ }*

The connection with the database is supposed to stay reliable and every SQL statement to be processed without any technical problem during the method's execution. The semantics of all the Java constructs conforms to the classical Java specification and documentation. The semantics of all SQL statements conforms to the classical SQL specification provided by ISO.

Common statements and lists management. Common condition, loop and assignment statements, as well as common integer expressions and boolean conditions can be used. Lists can be manipulated using the '`add(int)`', '`remove(int)`', '`get(int)`' and '`size(int)`' methods. The '`java.util.ArrayList<Integer>`' implementation of these methods is supposed to be used. A list variable can be '`null`'.

⟨*stmt*⟩ ::= `if (`⟨*cond*⟩`) {`⟨*stmt*⟩$_{then}$*`} {else {`⟨*stmt*⟩$_{else}$*`}}`$^?$`;`
| `while (`⟨*cond*⟩`) {` ⟨*stmt*⟩* `};`
| {`int` | `List<Integer>`}$^?$ ⟨*id*⟩ `=` ⟨*expr*⟩`;`
| ⟨*id*⟩`.add(` ⟨*int-expr*⟩ `);`
| ⟨*id*⟩`.remove(` ⟨*int-expr*⟩ `);`
| `return` ⟨*id*⟩`;`

⟨*cond*⟩ ::= `true`
  | `false`
  | `(!` ⟨*cond*⟩`)`
  | `(`⟨*cond*⟩$_1$ {`&` | `|`} ⟨*cond*⟩$_2$`)`
  | `(`⟨*int-expr*⟩$_1$ {`<` | `==` | `>`} ⟨*int-expr*⟩$_2$`)`
  | `(`⟨*id*⟩ `== null)`

⟨*expr*⟩ ::= ⟨*int-expr*⟩ | ⟨*list-expr*⟩

⟨*int-expr*⟩ ::= ⟨*id*⟩
  | ⟨*integer*⟩
  | `(`⟨*int-expr*⟩$_1$ {`+` | `-`} ⟨*int-expr*⟩$_2$`)`
  | `(`⟨*id*⟩`.get(` ⟨*int-expr*⟩ `))`
  | `(`⟨*id*⟩`.size())`

⟨*list-expr*⟩ ::= ⟨*id*⟩
  | `null`
  | `new ArrayList<Integer>()`

Interacting with the outside world. The scanner parameter of the method can be used to get integer data from the "outside world" (user prompt, network access, reading from a file, etc.). This interaction is supposed to always succeed, without any technical problem.

⟨*stmt*⟩ ::= {`int`}$^?$ ⟨*id*⟩ `= in.nextInt();`

Reading data from the database. Data can be read from the database using simple SQL queries. The obtained ResultSet can be accessed using the '`next()`' and '`getInt(String)`' methods.

⟨*stmt*⟩ ::= { `ResultSet` }$^?$ ⟨*id*⟩ `= con.createStatement().executeQuery("` ⟨*select-query*⟩ `");`
  | ⟨*id*⟩`.next();`

⟨*select-query*⟩ ::= `SELECT` {⟨*id*⟩$_i$`,`}*⟨*id*⟩$_n$ `FROM` ⟨*id*⟩$_{tab}$ { `WHERE` ⟨*db-cond*⟩ }$^?$

⟨*db-cond*⟩ ::= `(`⟨*db-cond*⟩$_1$ {`AND` | `OR`} ⟨*db-cond*⟩$_2$`)`
  | `(NOT` ⟨*db-cond*⟩`)`
  | `(`⟨*id*⟩ {`<` | `=` | `>`} ⟨*db-int-expr*⟩`)`

⟨*db-int-expr*⟩ ::= ⟨*id*⟩
  | ⟨*integer*⟩
  | `(`⟨*db-int-expr*⟩$_1$ {`+` | `-`} ⟨*db-int-expr*⟩$_2$`)`
  | `"+(` ⟨*int-expr*⟩ `)+"`

⟨*int-expr*⟩ ::= ⟨*id*⟩$_{tab}$`.getInt("` ⟨*id*⟩$_{att}$ `")`

⟨*cond*⟩ ::= `(`⟨*id*⟩`.next(` ⟨*int-expr*⟩ `))`

Writing data into the database. Data can be written into the database using simple SQL INSERT, UPDATE or DELETE statements. If the execution of a such a statement provokes a violation of one of the database schema integrity constraints, the database remains unmodified by the statement and an exception is thrown within the program and the method's execution is stopped. Such exceptions should be caught using a try/catch structure.

⟨*stmt*⟩ ::= `con.createStatement().execute("` ⟨*db-write*⟩ `");`
  | `try { con.createStatement().execute("` ⟨*db-write*⟩ `"); }`
    `catch (SQLException e)`
    `{` ⟨*stmt*⟩* `};`

⟨*db-write*⟩ ::= `INSERT INTO` ⟨*id*⟩ `VALUES (` { ⟨*int-expr*⟩$_i$ `,` }* ⟨*int-expr*⟩$_n$ `)`
| `UPDATE` ⟨*id*⟩$_{tab}$ `SET` ⟨*id*⟩$_{att}$ `=` ⟨*db-int-expr*⟩ { `WHERE` ⟨*db-cond*⟩ }?
| `DELETE FROM` ⟨*id*⟩ { `WHERE` ⟨*db-cond*⟩ }?

Transactions management. SQL transactions are managed through the classical commit and rollback statements. We suppose that a new transaction is automatically started at the beginning of the method's execution. The first call to commit or rollback will end this transaction and then starts a new one. Any subsequent call to commit or rollback will end the current transaction and start a new one. When a commit statement is executed, it makes permanent all the changes made to the database by the program since the current transaction was started. When a rollback statement is executed, it restores the database to its state at the start of the current transaction. We suppose that all the changes made to the database since the last transaction was started are automatically committed at the end of the method's execution.

⟨*stmt*⟩ ::= `con.commit() ;`
| `con.rollback() ;`

2.2. Relational symbolic execution algorithm

2.2.1. Inputs and outputs

The proposed algorithm, which is described in this section, receives as inputs the SQL DLL code describing the schema of the database, the Java code of the method under test and a single execution path through this method. It produces as output a constraint system mixing classical constraints with relational constraints. Solutions to this system are such that when the method is executed with respect to any of these solutions, its execution will follow the given path.

Coupling this algorithm with any existing technique (e.g. [23, 24, 25, 26]) able to select a set of paths to test in the program's control flow graph will allow to generate test data for these paths. The set of paths for which test data are computed, as well as the process used to select these paths, are thus parameters of the method that we propose. This allows the method to be used within the context of different code coverage criteria [4].

The subset of Java/SQL supported by our algorithm allows integers as only primary type in Java code and database tables. This choice was adopted to make the modelling and use of the constraint generation rules conceptually simpler. However, this does not fundamentally limit the power of the proposed technique since all other usual primitive types such as booleans, strings, and floating point numbers, but also data structures such as sets, arrays and matrices, and Java objects can be mapped to integers, simulated using lists of integers, or directly modelled into SMTv2.

The execution path received as input by our algorithm is supposed to be a finite path in the method's control flow graph [4]. It defines which branches were taken at each of the encountered if statements, for each encountered loop how many times its body was executed (this number must be finite), and for each encountered try/catch statement whether the catch clause was executed. A path terminates either when the end of the method is reached or when a return statement is executed.

Our algorithm translates the path into a constraint system, combining the *path constraint* with the *database schema integrity constraints*, expressed in SMTv2 [15], a widely adopted

language used as the standard language for many SMT solvers. The constraint system produced for a path can notably be solved using the Z3 solver [17]. Both Z3 and SMTv2 have been shown to be adequate [16] for relational constraint solving and are already at the core of several symbolic execution tools like [18, 19].

Solving the constraint system generated by the algorithm for a given path allows to find values for both the inputs and outputs of the analysed Java method. The inputs include the content of each database table at the start of the method's execution, the value of every list received as argument by the method, and a value for the part of the input stream that is consumed during the method's execution. The outputs include the content of each database table at the end of the method's execution, the final value of each of the argument lists of the method, and possibly the value returned by the return statement. If the constraint system produced for a given path has no solution, this means that the path is infeasible. As the produced constraints are written in a logic that is not decidable in general [16], it can happen that for a given path the solver may neither be able to find a solution for the generated constraint system, nor be able to establish that such a solution does not exist.

2.2.2. Algorithm principle

The algorithm performs a symbolic execution of the program path received as input. Each of the successive values taken by the method's variables and by the database tables during the execution of the path is represented by corresponding symbols and defined by constraints.

First, symbolic execution generates constraints over the symbols representing the initial values of the database tables. These constraints state that, initially, each table contains data that conform to the database schema integrity constraints.

Then, symbolic execution analyses one by one the method's statements in the order specified by the path. Each time a statement sets or changes the value of a method variable or database table, symbolic execution generates constraints over the symbols representing the new value. These constraints define how the new value can be computed from the values of the database tables and program variables before the statement's execution. Moreover, every time the value of a database table is changed, constraints are also added to state that the new value enforce the database schema integrity constraints.

Finally, every time an if, while or try/catch statement is encountered, symbolic execution generates an additional constraint over the symbols such that when the program is executed with respect to values satisfying this constraint, the execution is guaranteed to take the considered path.

2.2.3. Constraint generation rules

In the following paragraphs, we illustrate the execution of the algorithm over the example database program given in Figure 1. We detail each step of the symbolic execution process over a path where the while loop is executed once, the else branch of both the if statements is taken, and the catch clause of the try/catch is executed (lines 1-6, 8-15, 18-21, 3 and 22). At each step, we present the rules used by our algorithm to generate the corresponding SMTv2 symbols and constraints. All the rules that are part of the complete set of rules defining our symbolic execution algorithm for the fraction of Java/SQL defined in Section 2.1 are either presented during

one of these steps, or described formally in a set of tables available at the end of this section.

The first step executed by our algorithm is to generate SMTv2 symbols and constraints for the SQL DDL code of the database schema. For the database schema described in Figure 1, the generated SMTv2 code is detailed in the frame below, where new symbols and new symbol types are defined using the SMTv2 keyword "declare", while new constraints are enforced using the SMTv2 keyword "assert". First of all, the algorithm generates new symbol types for the kind of objects stored in each table defined by the schema (the lines prefixed by (0) in the SMTv2 code below). It will then generate symbols and constraints describing the input content of each of these tables. The used modelling is inspired by the one proposed in [16] for relational types. First, a symbol is created (1) to represent the initial set of objects in each table. Typed as a boolean function, it returns true for each object present in the input content of the table. Symbols typed as integer functions are then generated (2) to associate to each object in the table one of its attribute values. Finally, constraints are generated to enforce on this input content all the check constraints (3), primary key constraints (4), and foreign key constraints (5) defined in the schema.

Note that the original SQL table and attribute names, as well as the original Java variable names, are used as SMTv2 symbols, suffixed by the natural number 1 (e.g. *book*1 or *id*1 in the SMTv2 code below), which represents the fact that the current symbols represent the initial values of the represented tables, attributes or variables. Subsequent values of a same table, attribute or variable will be represented by the same symbol suffixed with successive numbers.

```
; New types for tables
(0) (declare-sort book)
(0) (declare-sort shelf)

; Input content of table Book
(1) (declare-fun book1 (book) Bool)
(2) (declare-fun shelfId1 (book) Int)
(2) (declare-fun code1 (book) Int)
(4) (assert (forall ((a book) (b book))
        (=> (and (and (book1 a) (book1 b)) (= (code1 a) (code1 b))) (= a b))))

; Input content of table Shelf
(1) (declare-fun shelf1 (shelf) Bool)
(2) (declare-fun numberOfBooks1 (shelf) Int)
(2) (declare-fun id1 (shelf) Int)
(3) (assert (forall ((a shelf)) (> (numberOfBooks1 a) 0)))
(4) (assert (forall ((a shelf) (b shelf))
        (=> (and (and (shelf1 a) (shelf1 b)) (= (id1 a) (id1 b))) (= a b))))

; Foreign keys
(5) (assert (forall ((a book))
        (=> (book1 a) (exists ((b shelf)) (and (shelf1 b) (= (shelfId1 a) (id1 b))))))))
```

The second step executed by our algorithm is to define a new SMTv2 symbol type (called BoundedList) for lists of integers. All the symbols that will be subsequently generated to represent the value of a variable typed as a Java list will be part of this new

SMTv2 type. A BoundedList symbol represents a record composed of three fields: the *isNull* field is typed as boolean, the *size* field is typed as integer and the *elements* field is typed as array of integers. If the *isNull* field is true, then the symbol represents the Java `null` value. Otherwise, the field *size* represents the size of the list, and the field *elements* represents an array whose indexes 0 to (*size* − 1) contain the elements of the list in the right order.

```
(declare-datatypes ()
   ((BoundedList (mk-bounded-list (isNull Bool) (size Int) (elements (Array Int Int))))))
```

The third step executed by our algorithm is to define symbols (typed as BoundedList) for the initial content of each list parameter of the method. For the example method considered in this section, the following code is generated:

```
(declare-const newbooks1 BoundedList)
(assert (=> (not (isNull newbooks1)) (>= (size newbooks1) 0)))
```

The algorithm can then proceed with the symbolic execution of the method. It follows the path received as input and considers all statements one by one. In the case of our example, the two first statements to be executed are assignments. Symbolic execution for assignment creates a new symbol of the correct type to represent the new value of the assigned variable (1) and generates constraints to specify that this new symbol contains the value computed by evaluating the expression on the right of the '`=`' symbol (2). In the particular case where a list variable is assigned to a different list variable, the shared content of the two variables is represented by a single symbol.

```
(1) (declare-const i1 Int)
(2) (assert (= i1 0))
```

```
(1) (declare-const addedbooks1 BoundedList)
(2) (assert (not (isNull addedbooks1)))
(2) (assert (= (size addedbooks1) 0))
```

The next statement in the path is a while statement. As the path specifies that the loop body must be executed, a constraint is generated to specify that the loop condition at this point of time should be true (1). As the '`size()`' method cannot be called on a null object without causing a runtime error, a constraint is automatically added to ensure that the current value of the '`newBooks`' variable of the method is not null (2).

```
(1) (assert (and (not (isNull newbooks1)) (< i1 (size newbooks1))))
(2) (assert (not (isNull newbooks1)))
```

Then the algorithm proceeds with symbolic execution of the statements in the loop body, as specified within the input path. The first statement is an assignment statement:

```
(declare-const error1 Int)
(assert (= error1 0))
```

Symbolic execution for use of the input scanner simply creates a new symbol to represent the scanned value:

```
(declare-const theshelf1 Int)
```

Symbolic execution for select statements creates new symbols to represent the content of the ResultSet variable. A first symbol (1) describes the number of rows returned by the select query. These rows are available through a second symbol (2) which is a function that returns them in the order in which they are returned by the ResultSet: (*shelves1List* 0) will be the first returned row, (*shelves1List* 1) the second one and so on.

```
(1) (declare-const shelves1Size Int)
(2) (declare-fun shelves1List (Int) shelf)
```

The modelling proposed in [16] for constraining the content and cardinality of relations is then used to specify that a row is part of the ResultSet if and only if it is part of the current content of the table on which the select query is executed and that it enforces the WHERE condition of the select query. In practice, new constraints are added (1) to define a function *shelves1InvertedList* which is the inverse of *shelves1List*. This function is used (1) to ensure that *shelves1List* defines a one to one correspondence between the integers $0 \leqslant i \leqslant$ *shelves1Size* and the elements in the ResultSet. Helper code (2) is also added to ensure a proper instantiation of the universal quantifiers by the solver.

```
(1) (declare-fun shelves1InvertedList (shelf) Int)
(2) (declare-fun shelves1Trigger (Int) Bool)
(1) (assert (and (>= shelves1Size 0)
                 (=> (= shelves1Size 0)
                     (forall ((c shelf)) (not (and (shelf1 c) (= (id1 c) theshelf1 )))))))
(1) (assert (forall ((c shelf))
   (=> (and (shelf1 c) (= (id1 c) theshelf1))
       (and (>= (shelves1InvertedList c) 0) (< (shelves1InvertedList c) shelves1Size )))))
(1) (assert (forall ((c shelf))
   (=> (and (shelf1 c) (= (id1 c) theshelf1))
       (= c (shelves1List (shelves1InvertedList c))))))
(1) (assert (forall ((i Int))
   (=> (and (>= i 0) (< i shelves1Size))
       (= i (shelves1InvertedList (shelves1List i))))))
(1) (assert (forall ((i Int))
   (! (=> (and (>= i 0) (< i shelves1Size))
          (and (shelf1 (shelves1List i)) (= (id1 (shelves1List i)) theshelf1)))
(2)    :pattern (shelves1Trigger i))))
(2) (assert (=> (>= 0 shelves1Size) (shelves1Trigger 1)))
(2) (assert (forall ((i Int))
   (! (=> (and (>= i 0) (< i shelves1Size))
          (shelves1Trigger (+ i 1)))
   :pattern (shelves1Trigger i))))
```

As the path specifies that the else branch of the if statement must be executed this time, a constraint is generated to specify that the condition of the if should be false, i.e. that `shelves.next()` should return true.

For each ResultSet object, the algorithm records the number of times the `next()` method has been called on this object. This value represents the index increased by one of the row pointed by the cursor of the ResultSet at the current execution state of the

path. When the boolean value returned by the '`next()`' method is used in an if or while condition, this value states if the number of rows in the ResultSet is greater or equal to the number of times the '`next()`' method has been called so far on this ResultSet. In this case, `shelves.next()` will return true if the ResultSet `shelves` contains more than one row (as `shelves.next()` has been called once on the ResultSet):

```
(assert (not (not (>= shelves1Size 1))))
```

Symbolic execution for update creates a new symbol (1) typed as an integer function, that will replace the previous symbol associating the attribute value to each object in the table. As this new symbol is the second one to represent the value of the attribute *numberOfBooks*, it is named *numberOfBooks*2. A couple of constraints (2)(3) is then generated to relate the old and new attribute values in the table: one for the rows that do not match the WHERE condition (2), and one for those that do (3). Finally, constraints are added to specify that no integrity constraint was violated during the update. In this case, a constraint (4) is added to state that the updated attribute values still enforce the check constraint defined in the database schema.

```
(1) (declare-fun numberOfBooks2 (shelf) Int)
(2) (assert (forall ((p shelf))
        (=> (or (and (shelf1 p) (not (= (id1 p) theshelf1))) (not (shelf1 p)))
            (= (numberOfBooks2 p) (numberOfBooks1 p)))))
(3) (assert (forall ((p shelf))
        (=> (and (shelf1 p) (= (id1 p) theshelf1))
            (= (numberOfBooks2 p) (+ (numberOfBooks1 p) 1)))))
(4) (assert (forall ((a shelf))
        (> (numberOfBooks2 a) 0)))
```

Subsequently, as in our example the path specifies that the catch block of the try/catch statement must be executed, a constraint (1) is added to ensure that the program variables and the database are in a state where the INSERT execution will violate a schema integrity constraint. In this case, the constraint states that the inserted row has a similar primary key as the primary key of an existing row in the table or that the inserted row has a foreign key value that does not reference any existing row in the shelf table. Constraints are also automatically added to ensure that the '`size()`' (2) and '`get(int)`' (3) methods do not cause any runtime error.

```
(1) (assert (or (exists ((a book)) (and (book1 a)
                                        (= (code1 a) (select (elements newbooks1) i1))))
                (forall ((a shelf)) (=> (shelf1 a)
                                        (not (= (id1 a) theshelf1 ))))))
(2) (assert (not (isNull newbooks1)))
(3) (assert (>= i1 0))
(3) (assert (< i1 (size newbooks1)))
```

The content of the catch block is then symbolically executed:

```
(declare-const error2 Int)
(assert (= error2 1))
```

As the path specifies that the else branch of the if statement must be executed this time, a constraint is generated to specify that the condition of the if should be false:

```
(assert (not (= error2 0)))
```

Symbolic execution for Rollback statements tells the algorithm to represent the current content of each database table using the symbols that were representing the content of the table just before the last start of a new transaction (saved by the algorithm at the beginning of the method execution and after each call to commit or abort). In this case, the database state is restored to its state at the method start, i.e. the algorithm rewinds the counters for the database symbols and symbols *book*1, *code*1, *shelfId*1, *shelf*1, *id*1 and *numberOfBooks*1 represent the content of the database after the '`con.rollback()`' statement.

The assignment statement is then symbolically executed:

```
(declare-const i2 Int)
(assert (= i2 (+ i1  1)))
```

As the path specifies that the loop body must not be executed any more, a constraint is generated to specify that, at this point in time, the loop condition should be false:

```
(assert (not (and (not (isNull newbooks1)) (< i2 (size newbooks1)))))
(assert (not (isNull newbooks1)))
```

As a return statement is met, the algorithm stops and returns the generated SMTv2 constraint model. The Z3 solver [17] can be asked to find a valuation for the defined symbols satisfying the constraints. As the algorithm records what symbols represent the initial, respectively final, values of a variable or table, the input and output values of the method (for the considered path) can easily be extracted from the solution to the constraint system.

For our example, the data that were obtained from the solution to the constraint system are summarised in the following tables:

Inputs

Name	**Symbol(s)**	**Value**
TABLE shelf	CONTENT: *shelf*1, ATTRIBUTES: *id*1 *numberOfBooks*1	id | n.Books 6 | 1 12 | 1
TABLE book	CONTENT: *book*1 ATTRIBUTES: *code*1 *shelfId*1	code | s.Id 4 | 12
newBooks	*newbooks*1	[4]
in.nextInt()	*theshelf*1	[6]

Outputs

Name	**Symbolic(s)**	**Value**
TABLE shelf	CONTENT: *shelf*1, ATTRIBUTES: *id*1 *numberOfBooks*1	id | n.Books 6 | 1 12 | 1
TABLE book	CONTENT: *book*1 ATTRIBUTES: *code*1 *shelfId*1	code | s.Id 4 | 12
newBooks	*newbooks*1	[4]
addedBooks	*addedbooks*1	[]

For sake of completeness, the following tables define the constraint generation rules used by our algorithm in case of an Insert (table 1), Update (table 2), Delete (table 3) or Add/Remove (table 4) statement. Table 5 explains the abbreviations used in the previous tables.

Table 1: Constraints generation rules for INSERT statements

`INSERT INTO` $\langle id \rangle$ `VALUES` ($\langle$*int-expr*$\rangle_1$, ... , $\langle$*int-expr*$\rangle_i$, ... , $\langle$*int-expr*$\rangle_n$)
if (*no exception thrown in path for this INSERT*) { *; Inserted primary key value does not already exist* (**assert**(**forall**((a $\langle id \rangle$))(⇒ (name($\langle id \rangle$) a)(**not** (= (name(pk) a) *smt2Of*($\langle$*int-expr*$\rangle_{\mathsf{pk}^{pos}}$)))))) *; Inserted values constrained by the i^{th} foreign key reference existing rows* (**assert** (**exists** ((a fk_i^{tab}))(**and** (= (name(fk_i^{pk}) a) *smt2Of*($\langle$*int-expr*$\rangle_{\mathsf{fk}_i^{pos}}$)) (name($\mathsf{fk}_i^{tab}$) a)))) *; Symbol for new table content* (**declare−fun** freshSym ($\langle id \rangle$) **Bool**) *; Constraints describing new table content* (**assert** (**forall** ((a $\langle id \rangle$)) (⇒ (name($\langle id \rangle$) a) (freshSym a)))) (**assert** (**exists** ((a $\langle id \rangle$)) (**and** (= (att_i a) *smt2Of*($\langle$*int-expr*$\rangle_i$)) (freshSym a)))) (**assert** (**forall** ((a $\langle id \rangle$)) (⇒ (**and** (**not** (name($\langle id \rangle$) a))(**not** (= (att_i a) *smt2Of*($\langle$*int-expr*$\rangle_i$))))(**not** (freshSym a))))) *; No duplicate inserted row* (**assert** (**forall** ((a $\langle id \rangle$) (b $\langle id \rangle$)) (⇒ (**and** (**and** (freshSym a) (freshSym b)) (= (pk a) (pk b))) (= a b)))) } **else** { *// Logical disjunction between every possible constraint* *// violation given the database schema and this insert:* *; The inserted primary key value already exists in the table* (**exists** ((a $\langle id \rangle$)) (**and** (name($\langle id \rangle$) a) (= (name(pk) a) *smt2Of*($\langle$*int-expr*$\rangle_{\mathsf{pk}^{pos}}$)))) *; i^{th} inserted foreign key value does not reference a row:* (**forall** ((a fk_i^{tab}))(⇒ (name(fk_i^{tab}) a) (**not** (= (name(fk_i^{pk}) a) *smt2Of*($\langle$*int-expr*$\rangle_{\mathsf{fk}_i^{pos}}$))))) *; An inserted attribute violates the i^{th} check constraint :* (**not** (co_i^{right} *smt2Of*($\langle$*int-expr*$\rangle_{\mathsf{co}_i^{pos}}$))) } }

Table 2: Constraints generation for UPDATE statements

`UPDATE` ⟨*id*⟩ `SET` ⟨*id*⟩$_{att}$ = ⟨*db-int-expr*⟩ `WHERE` ⟨*db-cond*⟩
if (*no exception thrown in path for this UPDATE*) { *; Symbol for new attribute values* (**declare–fun** freshSym (⟨*id*⟩) **Int**) *; Constraints describing new attribute values* (**assert** (**forall** ((a ⟨*id*⟩)) (⇒ (**or** (**and** (name(⟨*id*⟩) a) (**not** *smt2Of*(⟨*db-cond*⟩,⟨*id*⟩,a))) (**not** (name(⟨*id*⟩) a))) (= (freshSym a) (name(⟨*id*⟩$_{att}$) a))))) (**assert** (**forall** ((a ⟨*id*⟩)) (⇒ (**and** (name(⟨*id*⟩) a) *smt2Of*(⟨*db-cond*⟩,⟨*id*⟩,a)) (= (freshSym a) *smt2Of*(⟨*db-int-expr*⟩,⟨*id*⟩,a))))) *; Update on attribute constrained by i^{th} foreign key does not let pending references* (**assert** (**forall** ((a ⟨*id*⟩)) (⇒ (name(⟨*id*⟩) a) (**exists** ((b fk_i^{tab})) (**and** (name(fk_i^{tab}) b) (= (freshSym a) (name(fk_i^{pk}) b))))))) *; Update on attribute constrained by primary key does not let duplicate attribute values* (**assert** (**forall** ((a ⟨*id*⟩) (b ⟨*id*⟩)) (⇒ (**and** (**and** (name(⟨*id*⟩) a) (name(⟨*id*⟩) b)) (= (freshSym a) (freshSym b))) (= a b)))) *; Update on primary key referenced by i^{th} foreign key does not let pending references* (**assert** (**forall** ((a ifk_i^{tab})) (⇒ (name(ifk_i^{tab}) a) (**exists** ((b ⟨*id*⟩))(**and** (name(⟨*id*⟩) b) (= (name(ifk_i^{att}) a) (freshSym b))))))) *; Update on attribute constrained by i^{th} check constraint does not violate the constraint* (**assert** (**forall** ((a ⟨*id*⟩)) (co_i^{right} (freshSym a)))) } **else** { *// Logical disjunction between every possible constraint* *// violation given the database schema and this update:* *; Update on primary key leads to duplicate attribute values* (**exists** ((a ⟨*id*⟩) (b ⟨*id*⟩)) (**and** (**and** (name(⟨*id*⟩) a) (**and** (name(⟨*id*⟩) b) (**not** (= a b)))) (**or** (**and** *smt2Of*(⟨*db-cond*⟩,⟨*id*⟩,a) (**and** *smt2Of*(⟨*db-cond*⟩,⟨*id*⟩,b) (= *smt2Of*(⟨*db-int-expr*⟩,⟨*id*⟩,a) *smt2Of*(⟨*db-int-expr*⟩,⟨*id*⟩,b)))) (**and** (**not** *smt2Of*(⟨*db-cond*⟩,⟨*id*⟩,a)) (**and** *smt2Of*(⟨*db-cond*⟩,⟨*id*⟩,b) (= (name(⟨*id*⟩$_{att}$) a) *smt2Of*(⟨*db-int-expr*⟩,⟨*id*⟩,b))))))) *; Update on primary key referenced by the i^{th} foreign key lets pending references* (**exists** ((a ⟨*id*⟩) (b ⟨*id*⟩)) (**and** (**and** (name(⟨*id*⟩) a) (name(ifk_i^{tab}) b)) (**and** (**and** (**not** (= (name(⟨*id*⟩$_{att}$) a) *smt2Of*(⟨*db-int-expr*⟩,⟨*id*⟩,a))) (= (name(⟨*id*⟩$_{att}$) a) (name(ifk_i^{pk}) b))) *smt2Of*(⟨*db-cond*⟩,⟨*id*⟩,a)))) *; Update on attribute constrained by i^{th} foreign key lets pending references* (**exists** ((a ⟨*id*⟩)) (**and** (**and** (name(⟨*id*⟩) a) *smt2Of*(⟨*db-cond*⟩,⟨*id*⟩,a)) (**not** (**exists** ((b name(fk_i^{tab}))) (= (name(fk_i^{pk}) b) *smt2Of*(⟨*db-int-expr*⟩,⟨*id*⟩,a)))))) *; Update on attribute constrained by i^{th} check constraint violates the constraint* (**exists** ((a ⟨*id*⟩)) (**and** (**and** (name(⟨*id*⟩) a) *smt2Of*(⟨*db-cond*⟩,⟨*id*⟩,a)) (**not** (co_i^{right} *smt2Of*(⟨*db-int-expr*⟩,⟨*id*⟩,a))))) }

Table 3: Constraints generation for DELETE statements

`DELETE FROM` ⟨*id*⟩ `WHERE` ⟨*db-cond*⟩
if (*no exception thrown in path for this DELETE*) { *; Symbol for new table content* (**declare–fun** freshSym (⟨*id*⟩) **Bool**) *; Constraints describing new table content* (**assert** (**forall** ((a ⟨*id*⟩)) (= (freshSym a) (**and** (name(⟨*id*⟩) a) (**not** *smt2Of*(⟨*db-cond*⟩,⟨*id*⟩,a)))))) *; Delete does not let pending references for i^{th} foreign key* (**assert**(**forall** ((a fk_i^{tab}) (b ⟨*id*⟩)) (⇒ (**and** (name(⟨*id*⟩) b) (**and** (**not** (freshSym b)) (name(ifk_i^{tab}) a))) (**not** (= (name(pk) b) (name(ifk_i^{att}) a)))))) } **else** { *// Logical disjunction between every possible constraint* *// violation given the database schema and this update:* *; Delete lets pending references for i^{th} foreign key* (**exists** ((a fk_i^{tab}) (b ⟨*id*⟩)) (**and** (**and** (**and** (name(⟨*id*⟩) b) (name(ifk_i^{tab}) a)) *smt2Of*(⟨*db-cond*⟩,⟨*id*⟩,b)) (= (name(pk) b) (name(ifk_i^{att}) a)))) }

Table 4: Constraints generation for add(int) and remove(int) statements

⟨*id*⟩`.add(` ⟨*int-expr*⟩ `);`
(**declare–const** freshSym BoundedList) (**assert** (**not** (isNull name(⟨*id*⟩)))) (**assert** (**not** (isNull freshSym))) (**assert** (= (size freshSym) (+ (size name(⟨*id*⟩)) 1))) (**assert** (= (elements freshSym) (**store** (elements name(⟨*id*⟩)) (size name(⟨*id*⟩)) *smt2Of*(⟨*int-expr*⟩))))
⟨*id*⟩`.remove(` ⟨*int-expr*⟩ `);`
(**declare–const** freshSym BoundedList) (**assert** (**not** (isNull name(⟨*id*⟩)))) (**assert** (**not** (isNull freshSym))) (**assert** (≥ (size name(⟨*id*⟩)) 1)) (**assert** (= (size freshSym) (– (size "+ oldVar+ ") 1))) (**assert** (≥ *smt2Of*(⟨*int-expr*⟩) 0)) (**assert** (< *smt2Of*(⟨*int-expr*⟩) (size name(⟨*id*⟩)))) (**assert** (**forall** ((i **Int**)) (⇒ (**and** (≥ i 0) (< i *smt2Of*(⟨*int-expr*⟩)))) (= (**select** (elements name(⟨*id*⟩)) i) (**select** (elements freshSym) i))))) (**assert** (**forall** ((i **Int**)) (⇒ (**and** (≥ i *smt2Of*(⟨*int-expr*⟩)) (< i (size freshSym))) (= (**select** (elements name(⟨*id*⟩)) (+ i 1)) (**select** (elements freshSym) i)))))

Table 5: Abbreviations list

Abbreviation	**Meaning**
freshSym	A new symbol name that has still not been used in the SMTv2 code generated so far.
$smt2Of(x)$	Java condition/expression x translated into a corresponding SMTv2 condition/expression.
$smt2Of(x,t,r)$	SQL condition/expression x evaluated for row r in table t translated into a corresponding SMTv2 condition/expression.
name(x)	**if** (x refers to a database table name) **then** The symbol that represents the current content of table x **else if** (x refers to a database attribute name) The symbol that represents the current values of attribute x **else if** (x refers to a Java variable name) The symbol that represents the current content of the Java variable x
att_i	Name of the i_{th} attribute in the list of attributes of table $\langle id \rangle$
pk	Name of the primary key attribute of table $\langle id \rangle$.
pk^{pos}	Position of primary key in the list of attributes of table $\langle id \rangle$
fk_i^{tab}	Name of the table referenced by the i_{th} foreign key in the list of foreign keys of table $\langle id \rangle$
fk_i^{pk}	Name of the primary key attribute of the table referenced by the i_{th} foreign key in the list of foreign keys of table $\langle id \rangle$
fk_i^{pos}	Position of the foreign key attribute, declared by the i_{th} foreign key in the list of table $\langle id \rangle$, in the list of attributes of table $\langle id \rangle$
ifk_i^{tab}	Name of the table where is declared the i_{th} foreign key referencing table $\langle id \rangle$ in the whole schema
ifk_i^{att}	Name of the foreign key attribute declared by the i_{th} foreign key referencing table $\langle id \rangle$ in the schema
co_i^{pos}	Position of the attribute constrained by the i_{th} check constraint declared in table $\langle id \rangle$
co_i^{right}	Inverted right part of the i_{th} check constraint declared in table $\langle id \rangle$ (i.e. inverted right part of "a $>$ 0" is "$<$ 0")

3. Relational symbolic execution for database programs testing: an experimental validation

3.1. Symbolic execution and path exploration

Originally introduced in [5], symbolic execution has been used as the core principle of many test data generation techniques. In most of these techniques (see e.g. [27, 28] for an overview), symbolic execution is performed for a finite set of finite paths that are *statically* computed from the control-flow graph in accordance with a given coverage criterion [4]. By solving the constraints associated to each of the paths, one obtains a set of inputs and outputs that satisfy the given coverage criterion.

On the other hand, test data generation techniques that combine symbolic execution with a *dynamic* path exploration process have also been proposed (see e.g. [24, 25, 26]). The point of using a dynamic path exploration process is to provide symbolic execution with additional runtime information, to make it more effective [10, 28]. In practice, the program is first executed on concrete inputs to produce concrete outputs, but the code is instrumented so that symbolic execution is performed together with this normal run of the program, thereby generating the constraint system corresponding to the concrete execution. By flipping some of the constraints among those generated by this symbolic execution, one may produce constraints whose solution describes new concrete inputs triggering the execution of a different path. The process is then repeated with these new inputs until a set of inputs and outputs has been generated for a set of paths covering a sufficient part of the code, again according to some coverage criterion [4].

The technique we propose in this paper is orthogonal to the path exploration process, and can be combined with both a static or dynamic approach. However, in order to evaluate the ability and efficiency our technique, we have built a prototype tool that implements our relational symbolic execution for the defined subset of Java/SQL DML based on a static path exploration process.

3.2. Evaluation process

Our tool works as follows. Given a program to test, the code is analysed statically to select a set of paths to test in the program's control-flow graph. In a nutshell, the tool simply performs a depth-first search of the control-flow graph, selecting all possible paths that execute the body of each loop in the program at most K times, where K is an parameter of the tool. Consequently, the test data generated by our prototype satisfy a finitely applicable variant of the common path-coverage criterion [4], similar to the loop count-K criterion originally proposed in [29].

For each generated path, the tool applies the relational symbolic execution algorithm proposed in section 2, and solves the produced constraints using the Z3 [17] solver. If the solver finds a solution for the constraints, input and output data (including database content) is extracted from this solution and recorded as constituting a test-case. Once all the paths have been explored, a so-called *test-suite* comprising all the generated test-cases is returned to the user. If, on the other hand, the constraint set is found to be unsatisfiable, this means that the considered path cannot correspond to a real execution, and the process simply continues with the next path. The tool also keeps a separate list of the paths for which the solver cannot solve properly the constraints. This can happen as the constraints are written in a logic that is not decidable in general [16].

We have evaluated our prototype tool by using it to generate test data for eighteen Java methods, each of them performing SQL operations over a relational database. The methods we have used for the evaluation can be divided into three groups:

- The first group of methods were simply crafted to systematically evaluate the correct symbolic execution of the different constructs of the Java/SQL sub-language proposed in section 2.1. As such, the methods in this group exercise the different behaviours of the integer and list operators, conditional and loop statements, SQL statements and transaction management primitives.
- The second group contains the methods that were used to evaluate a previous version of our prototype [21]. The first method in this group performs repeated manipulations of integers and lists using assignment, if and while statements. The second method performs many interleaved reads and writes in a database containing four tables (representing regular or prospect customers that make purchases of products). The third method mixes SQL statements with traditional Java code and uses SQL transactions. The manipulated database contains two tables that represent authors writing theater plays.
- The third group contains Java methods extracted from two real-world applications, namely UnixUsage[2] and RiskIt[3], that have also been used as a basis for evaluation of other works on test data generation for database applications by symbolic execution [30, 13]. UnixUsage is a monitoring application for Unix, manipulating a database with eight tables and thirty one attributes. RiskIt is an insurance rate estimation application, manipulating a database with thirteen tables and fifty-seven attributes.

Together, the eighteen methods from our testbed constitute a set of five hundreds lines of code, containing notably eighty SQL statements (including SELECT, INSERT, UPDATE, DELETE statements, as well as transaction management code), over databases containing up to thirteen tables (subject to primary key, foreign key and check constraints). The code of these methods, as well as the generated test data, can be found on the web[4]. The prototype tool is itself coded in Java 1.6 and run on a dual core Intel Core i5 processor at 1.8GHz (256 KB L2 cache per core and 3 MB L3 cache) with 8GB of dual channel DDR3 memory at 1600 MHz. The runtime environment was the Oracle JVM 1.6.0_45 under a 32-bit edition of Windows 8.1. The version 4.3.0 of the Microsoft Z3 solver was used.

3.3. Results description

Table 6 synthesises the obtained results. In total 10.159 paths were symbolically executed and all generated constraint sets were properly solved by the solver: 10.046 paths were discovered to be infeasible (i.e. not corresponding to a real execution), while test data were extracted from the computed solutions for the remaining 113 paths. The high number of infeasible paths is principally due to the methods in the second group (and, to a lesser extend, the methods for testing conditional constructs from the first

[2] `http://sourceforge.net/projects/se549unixusage`
[3] `https://riskitinsurance.svn.sourceforge.net`
[4] `http://info.fundp.ac.be/~mmr/scp`

group), methods that were particularly constructed in this way for testing the constraint-generation code. The total time for constraint solving was nine minutes and thirty seconds.

Although the methods from the first group are all small and contain only simple operations, the results basically serve as a proof of concept showing the ability of our approach to generate, in a reasonable time, the correct constraints for a set of execution paths and, by solving them, to be able to either derive test data or correctly show the unfeasibility of the path.

The results for the methods in the second group reveal important improvements when compared to a previous version of our tool [21], in which the same methods were symbolically executed, but the generated constraints were expressed using the Alloy language [22] and solved using the Alloy analyzer [22]. The Alloy Analyzer solves a set of relational constraints by setting an upper bound on the size of the relations to be found. This makes finite the set of possible solutions, which can be exhaustively explored by transforming the constraints into an equivalent SAT problem, solved by a dedicated SAT solver. This mechanism is very costly in time and, moreover, did not allow to establish that a set of relational constraints is unsatisfiable [16]. As a consequence, [21] took more than eight minutes and a half to find test data for one path in the second sample and was not able to detect properly any infeasible path. In contrast, the current version of our tool, which makes use of the Z3 solver took thirty-two seconds to find test data or to establish unfeasibility for thirty-six paths in the second sample, including the path considered in [21].

Given the efficiency improvements with respect to our earlier work, we were now able to test our tool on real-world code. Test data generation for the third group of methods required to extend our symbolic execution algorithm to handle some new parts of Java and SQL, used in UnixUsage and RiskIt, or to simulate them in the base language recognised by the algorithm. In particular, the management of tables with no primary key or with multiple-attribute primary key was integrated in the algorithm and string management operations were simulated using either integers or lists of integers. As Table 6 show, correct test data was generated in just a few seconds and, moreover, the generated test data allowed to detect a possible failure in the code of RiskIt, where a runtime error is thrown when the method `createNewUser` is called on inputs where the inserted job does not reference any existing occupation or industry.

Table 6: Statistics for the selected samples

Name	SLOC	Number of SQL statements	Number of tables	K	Number of Paths		Solving Time
					Feasible	Infeasible	
Language units testing							
Integers	7	0	0	3	1	0	< 1s
Lists	9	0	0	3	1	0	< 1s
If and While	30	0	0	1	1	319	14s
Conditions	30	0	0	1	1	319	12s
Select	23	7	2	3	1	63	9s
Insert	14	7	2	3	1	7	<1s
Update	20	11	2	3	1	15	1s
Delete	6	3	2	3	1	1	<1s
Transactions	13	2	2	3	2	6	<1s
Samples from [21]							
Sample 1	45	1	1	5	23	3513	3m 16s
Sample 2	56	18	4	5	4	32	32s
Sample 3	72	4	2	2	63	5569	4m 35s
Samples from UnixUsage [30]							
courseNameExists	7	1	8	1	2	0	9s
getCommandsByCategory	10	1	8	1	2	0	<1s
getCourseIDByName	10	1	8		2	0	4s
getDepartmentIDByName	11	1	8	1	2	0	<1s
Samples from RiskIt [30]							
createNewUser	91	7	13	2	3	21	5s
deleteUsers	55	16	13	2	2	181	9s
ALL METHODS	509	80	up to 13	up to 5	113	10046	9m30s

4. Discussion

4.1. Conclusions

Symbolic execution is the core process of many existing test data generation techniques [9]. These approaches explore a representative [4] set of finite paths to test in the program, either statically [10] or dynamically [24, 25, 26], and execute them symbolically to generate meaningful test inputs and outputs for the program. In this work, we propose an approach for the symbolic execution of SQL code, which is a non-trivial extension to traditional symbolic execution because of the complex structure of relational databases and the complex behaviour of SQL statements. Given a database program mixing traditional code with SQL statements, each database table manipulated by the program is modelled as a variable typed as a relation and each SQL statement as a relational operation over both these relational variables and the traditional variables of the program. A classical symbolic execution process can then be applied to produce sets of mixed relational and conventional constraints over symbols representing the values of both the classical and relational variables of the program. The resulting path constraints can be unified with the data integrity constraints from the database schema. Any solution to the resulting constraint system for a path describes input and output data for the program, including a valid database content, with respect to which the program can be executed and is guaranteed to follow the same path for which the constraints were generated.

A complete symbolic execution algorithm has been developed for simple Java methods that, using SQL statements and transactions, read and write in a relational database; the latter typically subject to data integrity constraints. Given the schema of the database, the code of the method and an execution path in this method, our algorithm performs symbolic execution of the path and produces the corresponding constraints. The algorithm has been implemented in a tool that has been used to generate test data for a number of methods interacting with a database, including some real-world application code. For every tested method, we used the symbolic execution algorithm to generate test data for a set of paths that is statically generated by a bounded depth-first traversal of the program's control-flow graph [4]. Results revealed that the approach can properly consider thousands of execution paths in a reasonable time and thus generate meaningful test data.

Compared to our previous work [20, 21] where the generated constraints were expressed using the Alloy language [22] and solved using the Alloy analyser [22], the current implementation generates SMTv2 [15] constraints which are solved using the Z3 solver [17] following an approach similar to [16]. SMTv2 is the standard input language for SMT solvers and Z3 is used as constraint solving back-end in several symbolic execution tools [18, 19]. Results show that this change makes the approach able to properly detect infeasible paths, something that the implementation based on Alloy was unable to, and allows to reduce the time needed for constraint solving by several orders of magnitude.

4.2. Related work

An early work that has considered test data generation for programs interacting with a relational database is [11]. The paper suggests to transform the program by inserting new classical variables representing the database structure, and translating all SQL statements into traditional program code. Classical white-box testing approaches can then be applied to the normalised program. A conceptually similar but entirely automated technique is proposed in [13], where an off-the-shelf test generation technique, based on symbolic

execution and a dynamic path exploration process, is applied to the normalised version of the program. This latter approach is validated over real application code, considering methods containing compound SELECT statements.

Normalising SQL code into program code and then performing symbolic execution on the result is an alternate approach to ours, where the SQL code is directly transformed into relational constraints. Replacing a single SQL statement by an often complex piece of new program code that simulates the execution of the SQL statement by a DBMS, will dramatically increase the number of paths to be explored compared to the original code [11]. In particular, given a path in the original code, all the combinations between this original path and the different subpaths traversing the newly added code should be shown infeasible, in order to show that the original path was infeasible in the original code. As the new code will contain loops to enumerate the unbounded content of the database tables [13], there is an infinite number of subpaths that traverse this new code, and an infinite number of paths should thus be analysed in the combined code to detect properly if a path in the original code was infeasible. In contrast, in our approach, the execution paths to be considered are limited to paths in the *original* program code and, consequently, our technique is able to quickly detect the definite unfeasibility of a path.

In [12], the authors propose a symbolic execution algorithm for programs performing basic SELECT queries on a relational SQL database. Constraints are solved using an ad-hoc solver for linear arithmetic and strings. The proposed algorithm is integrated in a test generation technique based on a dynamic path exploration process and evaluated over real application code. Compared to this approach, our technique handles INSERT, UPDATE and DELETE statements, as well as transaction management primitives, which are crucial components of database applications. Moreover, [12] ignores the data integrity constraints defined in the database schema, leading to the possible generation of invalid test data. On the other hand, our approach only considers SQL statements whose structure is completely defined statically, where [12] (and also [13]) can allow to account, at least to some extent, for dynamically crafted SQL statements. This is linked to the fact that these two techniques use a dynamic path exploration process, where symbolic execution is performed in parallel with a concrete run of the program, giving direct access to the code of the dynamic SQL statements, as it is built by the program during the concrete run.

Following [12], several works have studied additional research questions in the framework of test generation for database programs using symbolic execution and a dynamic path exploration process. Among them, [31] proposes to use a mock database in case the original database is not available, [32] considers using advanced code coverage criteria, [33] considers programs where parts of the executed SQL queries are inputs of the program. Finally, [30] and [33] study situations where the test data should be selected in an existing database instead of being generated from scratch.

On a related but complementary level, a substantial amount of work (e.g. [34, 35, 36, 37, 38, 39, 40, 41, 42, 43]) has been done on how to generate test database content exhibiting some desirable properties, given only the database schema and a set of queries to be executed over the database. The main difference between our work and these approaches is that they essentially work without considering the control flow of the programs manipulating the database. Other work [44, 45] considers *mutation testing* of database programs, where our approach performs *structural testing*. In mutation testing, the quality of the test data is no more measured in terms of code coverage, as in structural testing, but in terms of program fault detecting ability (see [4] for a broader discussion).

Some works have also focused over testing of non-functional aspects of database programs, like security testing [46, 47] and performance testing [48].

4.3. Future work

In future work it should be investigated to what extent the symbolic execution mechanism can be generalised and extended towards dealing with a larger part of the SQL language. In particular, it should be investigated how and to what extent fully *dynamic* SQL can be integrated in our technique, possibly by relying on static analysis [49, 50, 51] or by coupling our symbolic execution algorithm with a dynamic path exploration process, in a similar way to [12, 13]. Ideas proposed in [33], where parts of the executed SQL queries are inputs of the program, should also be investigated. It also frequently happens that SQL statements have a non-deterministic behaviour, because the statement has a non-deterministic semantics, or because the interaction with the DBMS might not be reliable or because the database is modified concurrently by other methods or programs. How an approach for test data generation could handle handle such non-deterministic behaviour remains a topic for further research.

In the perspective of using an approach such as ours for testing large real-world programs that manipulate huge databases using many complex SQL statements, it is to be expected that a tighter integration between constraint generation (path exploration) and constraint solving would be beneficial. The constraint generator should be tailored so to generate very efficient sets of constraints, expressed in parts of the logic that are decidable or optimized for the heuristics used by the solver. Conversely, the use of solving algorithms or heuristics tailored to the kind of constraints produced by the symbolic execution of large pieces of complex SQL code should also be considered.

In addition to dealing with more involved database manipulations, the approach would also profit from an integration with existing symbolic execution test generation tools [9], handling a larger part of Java, like [52] or [53, 54]. In a similar vain, the use of existing Z3 plug-ins, like [55], allowing the native management of string constraints within the Z3 solver, should permit our algorithm to more easily handle string values, which are widely used in practice within database programs.

The problem of generating test data for programs manipulating an existing database is a very common problem in practice. Whether and how our technique should select test data from an *existing* database, instead of generating one from scratch, in a similar way to [30, 33], is an interesting matter of further research, as is the problem of considering or generating a single database content that could be used with several successive test cases.

Finally, being somehow parametrised with respect to the paths that should be considered, our approach allows to be used with respect to any traditional code coverage criterion based on the notion of execution path [4]. Nevertheless, several works [56, 57, 58, 59, 60] propose test adequacy criteria particularly tailored towards testing of database programs. Integrating such criteria into our constraint-based approach is a topic of ongoing research.

Acknowledgment

This work has been funded by the Belgian Fund for Scientific Research (F.R.S.-FNRS). The authors would like to thank Pierre-Yves Schobbens and Gilles Perrouin for useful discussions and the anonymous referees for their constructive remarks on earlier versions of this paper.

References

[1] P. C. Jorgensen, Software Testing: A Craftsman's Approach, Third Edition, 3rd Edition, AUERBACH, 2008.

[2] C. Kaner, H. Q. Nguyen, J. L. Falk, Testing Computer Software, 2nd Edition, Wiley & Sons, New York, NY, USA, 1993.

[3] R. Ramler, K. Wolfmaier, Economic perspectives in test automation: balancing automated and manual testing with opportunity cost, in: Proceedings of the 2006 international workshop on Automation of software test, AST '06, ACM, New York, NY, USA, 2006, pp. 85–91.

[4] H. Zhu, P. A. V. Hall, J. H. R. May, Software unit test coverage and adequacy, ACM Comput. Surv. 29 (4) (1997) 366–427.

[5] J. C. King, Symbolic execution and program testing, Commun. ACM 19 (7) (1976) 385–394.

[6] P. D. Coward, Symbolic execution systems: a review, Softw. Eng. J. 3 (6) (1988) 229–239.

[7] C. S. Pasareanu, W. Visser, A survey of new trends in symbolic execution for software testing and analysis, Int. J. Softw. Tools Technol. Transf. 11 (4) (2009) 339–353.

[8] E. J. Schwartz, T. Avgerinos, D. Brumley, All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask), in: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 317–331.

[9] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, W. Visser, Symbolic execution for software testing in practice: Preliminary assessment, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, New York, NY, USA, 2011, pp. 1066–1071.

[10] C. Cadar, K. Sen, Symbolic execution for software testing: three decades later, Communication of the ACM 56 (2) (2013) 82–90.

[11] M. Y. Chan, S. C. Cheung, Testing database applications with sql semantics, in: In Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications, Springer, 1999, pp. 363–374.

[12] M. Emmi, R. Majumdar, K. Sen, Dynamic test input generation for database applications, in: Proceedings of the 2007 international symposium on Software testing and analysis, ISSTA '07, ACM, New York, NY, USA, 2007, pp. 151–162.

[13] K. Pan, X. Wu, T. Xie, Guided test generation for database applications via synthesized database interactions, ACM Transactions on Software Engineering and Methodology.

[14] C. Date, An Introduction to Database Systems, 8th Edition, Addison-Wesley Longman Pub. Co., Inc., Boston, MA, USA, 2003.

[15] C. Barrett, A. Stump, C. Tinelli, The SMT-LIB Standard: Version 2.0, in: A. Gupta, D. Kroening (Eds.), Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK), 2010.

[16] A. A. El Ghazi, M. Taghdiri, Relational reasoning via smt solving, in: Proceedings of the 17th international conference on Formal methods, FM'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 133–148.

[17] L. De Moura, N. Bjørner, Z3: An efficient smt solver, in: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 337–340.

[18] N. Tillmann, J. De Halleux, Pex: white box test generation for .net, in: Proceedings of the 2nd international conference on Tests and proofs, TAP'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 134–153.

[19] P. Godefroid, M. Y. Levin, D. Molnar, Sage: Whitebox fuzzing for security testing, Queue 10 (1) (2012) 20:20–20:27.

[20] M. Marcozzi, W. Vanhoof, J.-L. Hainaut, Test input generation for database programs using relational constraints, in: Proceedings of the Fifth International Workshop on Testing Database Systems, DBTest '12, ACM, New York, NY, USA, 2012, pp. 6:1–6:6.

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

[22] D. Jackson, Software Abstractions: Logic, Language, and Analysis, The MIT Press, 2006.

[23] S. Bardin, P. Herrmann, Pruning the search space in path-based test generation, in: Proceedings of the 2009 International Conference on Software Testing Verification and Validation, ICST '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 240–249.

[24] P. Godefroid, N. Klarlund, K. Sen, Dart: directed automated random testing, SIGPLAN Not. 40 (6) (2005) 213–223.
[25] K. Sen, D. Marinov, G. Agha, Cute: a concolic unit testing engine for c, SIGSOFT Softw. Eng. Notes 30 (5) (2005) 263–272.
[26] C. Cadar, D. Engler, Execution generated test cases: How to make systems code crash itself, in: Proceedings of the 12th International Conference on Model Checking Software, SPIN'05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 2–23.
[27] J. Edvardsson, A survey on automatic test data generation, in: Proceedings of the Second Conference on Computer Science and Engineering in Linköping, ECSEL, 1999, pp. 21–28.
[28] D. D'Souza, T. Kavitha, J. Radhakrishnan (Eds.), IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India, Vol. 18 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
[29] W. E. Howden, Methodology for the generation of program test data, Computers, IEEE Transactions on 100 (5) (1975) 554–560.
[30] K. Pan, X. Wu, T. Xie, Generating program inputs for database application testing, in: Proc. 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011.
[31] K. Taneja, Y. Zhang, T. Xie, Moda: Automated test generation for database applications via mock objects, in: In Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE 2010), short paper, 2010.
[32] K. Pan, X. Wu, T. Xie, Database state generation via dynamic symbolic execution for coverage criteria, in: Proceedings of the Fourth International Workshop on Testing Database Systems, ACM, New York, NY, USA, 2011.
[33] C. Li, C. Csallner, Dynamic symbolic database application testing, in: Proceedings of the Third International Workshop on Testing Database Systems, DBTest '10, ACM, New York, NY, USA, 2010, pp. 7:1–7:6.
[34] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, E. J. Weyuker, An agenda for testing relational database applications: Research articles, Softw. Test. Verif. Reliab. 14 (1) (2004) 17–44.
[35] D. Chays, J. Shahid, P. G. Frankl, Query-based test generation for database applications, in: Proceedings of the 1st international workshop on Testing database systems, DBTest '08, ACM, New York, NY, USA, 2008, pp. 6:1–6:6.
[36] A. Neufeld, G. Moerkotte, P. C. Lockemann, Generating consistent test data for a variable set of general consistency constraints, VLDB J. 2 (2) (1993) 173–213.
[37] J. Zhang, C. Xu, S. C. Cheung, Automatic generation of database instances for white-box testing, in: Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development, COMPSAC '01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 161–165.
[38] C. Binnig, D. Kossmann, E. Lo, Reverse query processing, in: in the IDEA System. Int. Symp. on Advanced Database Technologies and Their Integration, 1994.
[39] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, S. Khurshid, Query-aware test generation using a relational constraint solver, in: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 238–247.
[40] M. Veanes, P. Grigorenko, P. Halleux, N. Tillmann, Symbolic query exploration, in: Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering, ICFEM '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 49–68.
[41] C. de la Riva, M. J. Suárez-Cabal, J. Tuya, Constraint-based test database generation for sql queries, in: Proceedings of the 5th Workshop on Automation of Software Test, AST '10, ACM, New York, NY, USA, 2010, pp. 67–74.
[42] D. Willmor, S. M. Embury, An intensional approach to the specification of test cases for database applications, in: Proceedings of the 28th international conference on Software engineering, ICSE '06, ACM, New York, NY, USA, 2006, pp. 102–111.
[43] C. Y. Chan, B. C. Ooi, A. Zhou (Eds.), Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007, ACM, 2007.
[44] C. Zhou, P. Frankl, Mutation testing for java database applications, in: Software Testing Verification and Validation, 2009. ICST '09. International Conference on, 2009, pp. 396–405.
[45] K. Pan, X. Wu, T. Xie, Automatic test generation for mutation testing on database applications, in: Automation of Software Test (AST), 2013 8th International Workshop on, 2013, pp. 111–117.
[46] W. G. J. Halfond, A. Orso, P. Manolios, Using positive tainting and syntax-aware evaluation to counter sql injection attacks, in: Proceedings of the 14th ACM SIGSOFT international symposium

on Foundations of software engineering, SIGSOFT '06/FSE-14, ACM, New York, NY, USA, 2006, pp. 175–185.
[47] Z. Su, G. Wassermann, The essence of command injection attacks in web applications, SIGPLAN Not. 41 (1) (2006) 372–382.
[48] X. Wu, C. Sanghvi, Y. Wang, Y. Zheng, Privacy aware data generation for testing database applications, in: Proceedings of the 9th International Database Engineering & Application Symposium, IDEAS '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 317–326.
[49] G. Wassermann, C. Gould, Z. Su, P. Devanbu, Static checking of dynamically generated queries in database applications, ACM Trans. Softw. Eng. Methodol. 16 (4).
[50] S. Thomas, L. Williams, T. Xie, On automated prepared statement generation to remove sql injection vulnerabilities, Inf. Softw. Technol. 51 (3) (2009) 589–598.
[51] A. Christensen, A. M,àö,àèller, M. Schwartzbach, Precise analysis of string expressions, in: R. Cousot (Ed.), Static Analysis, Vol. 2694 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, pp. 1–18.
[52] K. Sen, G. Agha, Cute and jcute: concolic unit testing and explicit path model-checking tools, in: Proceedings of the 18th international conference on Computer Aided Verification, CAV'06, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 419–423.
[53] S. Anand, C. S. Păsăreanu, W. Visser, Jpf-se: A symbolic execution extension to java pathfinder, in: Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 134–138.
[54] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, M. Pape, Combining unit-level symbolic execution and system-level concrete execution for testing nasa software, in: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08, ACM, New York, NY, USA, 2008, pp. 15–26.
[55] Y. Zheng, X. Zhang, V. Ganesh, Z3-str: A z3-based string solver for web application analysis, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, ACM, New York, NY, USA, 2013, pp. 114–124.
[56] W. G. J. Halfond, A. Orso, Command-form coverage for testing database applications, in: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 69–80.
[57] G. M. Kapfhammer, M. L. Soffa, A family of test adequacy criteria for database-driven applications, in: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-11, ACM, New York, NY, USA, 2003, pp. 98–107.
[58] M. J. Suárez-Cabal, J. Tuya, Using an sql coverage measurement for testing database applications, in: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, SIGSOFT '04/FSE-12, ACM, New York, NY, USA, 2004, pp. 253–262.
[59] M. J. Suarez-Cabal, J. Tuya, Structural coverage criteria for testing SQL queries, Journal of Universal Computer Science 15 (3) (2009) 584–619.
[60] C. Zhou, P. Frankl, Mutation testing for java database applications, in: Proceedings of the 2009 International Conference on Software Testing Verification and Validation, ICST '09, IEEE Comp. Soc., Washington, DC, USA, 2009, pp. 396–405.