A relational symbolic execution algorithm for constraint-based testing of database programs

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

Published in:
IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013

DOI:
10.1109/SCAM.2013.6648200

Publication date:
2013

Document Version
Publisher's PDF, also known as Version of record

2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)

A Relational Symbolic Execution Algorithm for Constraint-Based Testing of Database Programs

Michaël Marcozzi*
Faculty of Computer Science
University of Namur
Rue Grandgagnage, 21
Namur, Belgium
Email: michael.marcozzi@unamur.be

Wim Vanhoof
Faculty of Computer Science
University of Namur
Rue Grandgagnage, 21
Namur, Belgium
Email: wim.vanhoof@unamur.be

Jean-Luc Hainaut
Faculty of Computer Science
University of Namur
Rue Grandgagnage, 21
Namur, Belgium
Email: jean-luc.hainaut@unamur.be

***Abstract*—In constraint-based program testing, symbolic execution is a technique which allows to generate test data exercising a given execution path, selected within the program to be tested. Applied to a set of paths covering a sufficient part of the code under test, this technique permits to generate automatically adequate test sets for units of code. As databases are ubiquitous in software, generalizing such a technique for efficient testing of programs manipulating databases is an interesting approach to enhance the reliability of software. In this work, we propose a relational symbolic execution algorithm to be used for testing of simple Java methods, reading and writing with transactional SQL in a relational database, subject to integrity constraints. This algorithm considers the Java method under test as a sequence of operations over a set of constrained relational variables, modeling both the database tables and the method variables. By integrating this relational model of the method and database with the classical symbolic execution process, the algorithm can generate a set of Alloy constraints for any finite path to test in the control-flow graph of the method. Solutions of these constraints are data which constitute a test case, including valid content for the database, which exercises the selected path in the method. A tool implementing the proposed algorithm is demonstrated over a number of examples.**

I. Introduction

Testing [1], [2] constitutes the primary approach to improve the reliability of software. This motivates [3] much research to develop relevant automated techniques to test efficiently all aspects of software. In particular, two successful white-box approaches have emerged [4] to generate automatically adequate test data for testing units of code, with regard to their expected functions. In search-based testing (e.g. [5]), optimization heuristics are used to search the input space of the program for adequate test data. In constraint-based testing (e.g. [6], [7], [8], [9], [10], [11]), the program is transformed into constraints whose solutions are adequate test data.

Among constraint-based test data generation strategies, classical symbolic execution [12] works in three successive steps. First, a finite set of finite paths in the control flow graph [13] of the program are selected for testing. Several selection algorithms have been proposed (see e.g. [14]) to select a set of paths in the program which satisfies a given code coverage criterion [13]. Secondly, each selected path is symbolically executed [15], [16], [17], [12] to build a set of path constraints over the program inputs. These constraints are such that when the program is executed with respect to input values satisfying them, the execution is guaranteed to follow the path from which they were generated. In order to build these constraints, the symbolic execution algorithm considers the static single assignment form of the program and expresses the control dependencies imposed by the path to be tested. Thirdly, the constraints generated for each of the selected paths are solved, producing each time a new test case, which will exercise a different path among those initially selected in the program.

Databases are nowadays ubiquitous in software and many units of code interact intensively with a large, persistent and highly-structured relational database [18]. But barely few works (see [19], [20], [21]) have studied how to automate the generation of test inputs for such database programs, to test the correct *interaction* between the code and the database.

In this work, we propose to generalize the classical symbolic execution technique to the case of programs interacting with a relational database through SQL statements. We intend to generate a set of test cases for the program, including, for each test case, an input database content and values for the program inputs, to satisfy a given code coverage criterion. The path selection step of classical symbolic execution does not require any particular modification to handle database programs, as the presence of SQL statements in a program typically does not modify the way its control flow graph can be explored. The existing path selection algorithms can thus be used as well for database programs. However, the symbolic execution and constraints solving steps need to be adapted. First, because the generated constraints must now describe the content

*F.R.S.-FNRS Research Fellow

978-1-4673-5739-5/13/$31.00

of the relational tables of the database, subject to complex integrity constraints, as well as the content of the variables of the program. Secondly, because the symbolic execution process must be able to generate constraints modeling the complex behavior of the SQL statements used in the program, able to fail when violating the integrity constraints of the database.

The core strategy of our technique is, as we suggested in [21], to model every variable of the program and every table (which is, mathematically, a relation) in the database as a relational variable containing a mathematical relation over simple domains, like integers. Each statement in the program, including both imperative and SQL statements, can then be modeled as a simple relational operation over these relational variables. By applying the classical symbolic execution mechanism over this relational version of the program, we can derive a set of path constraints over the program input variables. The generated constraints are here relational constraints and the input variables refer both to the classical inputs of the program and to the content of each of the database tables at the program start. As some of the relational variables manipulated by the relational version of the program model tables in a database, they must obey the integrity constraints described by the relational schema of this database, such as, for example, the primary key or foreign key constraints. These schema constraints are modeled as relational constraints as well, and the path and schema constraints can then be combined into an unique input constraints system. Each solution to this relational constraints system represents a test input, including an initial state for each table in the database, with respect to which the program can be executed and is guaranteed to follow the execution path to be tested.

The main contribution of this work is a relational symbolic execution algorithm for database programs. This algorithm generates test data for a language composed of simple static Java methods, interacting with a relational database using SQL statements, through JDBC. Given the SQL DDL code describing the database schema, the Java code of the method and a finite path in the control flow graph of this method, the algorithm generates the corresponding relational input constraints in the Alloy language [22]. These constraints can then be solved using the Alloy analyzer [22]. The algorithm has been coded in Java and used to generate sets of test cases for a number of sample Java methods. The generated test sets satisfy classical code coverage criteria [13].

The remainder of this paper is organized as follows. Section II details the part of the Java/SQL syntax which is supported by our algorithm. In section III, we describe and illustrate our symbolic execution algorithm. We provide three examples of the whole test data generation process in section IV. Finally, some conclusions, related and future work are provided in section V.

II. Syntax of the tested Java/SQL programs

In this section, we define precisely which 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. It includes common imperative statements, expressions and conditions, typical methods for List objects manipulation, SQL base primitives used in online transaction processing (OLTP) programs, base transaction management statements and typical database schema constraints constructs.

A sample database program in the language handled by our symbolic execution algorithm is provided in figure 1. This sample describes a database with two tables: one for shop departments and one for the articles stored in each of these departments. The total number of articles stored in a department is saved for each department. The sample also describes the method manipulating this database: it adds a set of new articles to the database and updates the departments' article counts. If the department of an added article was not in the database, it is added to the database as well. The articles are inserted one by one in isolated transactions. If a transaction was successful, the index of the added article is saved in a list.

In the next paragraphs, the chosen notation for the BNF grammar of the syntax 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.

A. 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*⟩

B. Database schema

The relational database schema is a list of table definitions. This list can be empty. In such a case, the program works independently of any database. This allows our symbolic execution algorithm to be used for test input generation in the case of classical Java methods, with no interaction with a database. 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 updated 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.

⟨*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}

Fig. 1. SQL DDL and Java code of a database program inserting articles and updating departments' articles count in a shop database.

```
CREATE TABLE department (                    CREATE TABLE article (
 id INTEGER NOT NULL,                         barcode INTEGER NOT NULL,
 numberOfArticles INTEGER NOT NULL,           theDep INTEGER NOT NULL,
 CONSTRAINT dPK PRIMARY KEY (id),             CONSTRAINT aPK PRIMARY KEY (barcode),
 CHECK(numberOfArticles > 0))                 CONSTRAINT aFK FOREIGN KEY(theDep)REFERENCES department(id))
```

```
1  static void sample (Connection con,Scanner in,List<Integer> newArticles) throws SQLException {
2   int i = 0;
3   List<Integer> addedArticles = new ArrayList<Integer>();
4   while (!(newArticles==null) & i<newArticles.size()) {
5     int error = 0;
6     int departmentId = in.nextInt();
7     ResultSet departments = con.createStatement().executeQuery("SELECT id FROM department WHERE id="+departmentId);
8     if (!departments.next())
9      con.createStatement().execute("INSERT INTO department VALUES ("+departmentId+",1)");
10    else
11     con.createStatement().execute("UPDATE department SET numberOfArticles=numberOfArticles+1 WHERE id ="+departmentId);
12    try {
13     con.createStatement().execute("INSERT INTO article VALUES ("+newArticles.get(i)+","+departmentId+")");
14    } catch (SQLException e) {
15     error = 1;
16    };
17    i = i + 1;
18    if (error==0) {
19     con.commit();
20     addedArticles.add(i−1);
21    } else
22     con.rollback ();
23  }; }
```

C. Method parameters and body

We consider simple static 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 a connexion to the database (typed as '`java.sql.Connection`'), an input scanner (typed as '`java.util.Scanner`') and some lists of integers (typed as '`java.util.List<java.lang.Integer>`') as input parameters. The connection with the database is supposed to stay reliable and every SQL statement to be processed without any technical problem during the whole method execution. Semantics of all the Java constructs conforms to the classical Java specification and documentation. Semantics of all SQL statements conforms to the classical SQL specification provided by ISO.

⟨*java-method*⟩ ::= `static void` ⟨*id*⟩ `(`⟨*db-con*⟩`,`⟨*inp*⟩ ⟨*parameters*⟩`) throws SQLException {` ⟨*stmt*⟩* `}`
⟨*db-con*⟩ ::= `Connection con`
⟨*inp*⟩ ::= `Scanner in`
⟨*parameters*⟩ ::= {`,` `List<Integer>` ⟨*id*⟩ }*

1) 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(Integer)`', '`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*⟩ `);`
⟨*cond*⟩ ::= `true`
  | `false`
  | `(`⟨*cond*⟩$_1$ {`&` | `|`} ⟨*cond*⟩$_2$`)`
  | `(!` ⟨*cond*⟩`)`
  | `(`⟨*int-expr*⟩$_1$ {`<` | `==` | `>`} ⟨*int-expr*⟩$_2$`)`
  | `(`⟨*id*⟩ `== null)`
⟨*expr*⟩ ::= ⟨*int-expr*⟩ | ⟨*list-expr*⟩
⟨*int-expr*⟩ ::= ⟨*id*⟩
  | ⟨*integer*⟩
  | `(`⟨*int-expr*⟩$_1$ {`+` | `-` | `*` | `/`} ⟨*int-expr*⟩$_2$`)`
  | `(-` ⟨*int-expr*⟩ `)`
  | `(`⟨*id*⟩`.get(`⟨*int-expr*⟩`))`
  | `(`⟨*id*⟩`.size())`
⟨*list-expr*⟩ ::= ⟨*id*⟩
  | `null`
  | `new ArrayList<Integer>()`

2) Interacting with the outside world: the input 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();

3) 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*⟩ ::= ⟨*select*⟩
  | ⟨*id*⟩`.next();`
⟨*select*⟩ ::= {`ResultSet`}?⟨*id*⟩`=`
  `con.createStatement().executeQuery`
  `("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*⟩$_2$)
| (`-` ⟨*db-int-expr*⟩)
| "`+(` ⟨*int-expr*⟩ `)+`"
⟨*int-expr*⟩ ::= ⟨*id*⟩$_{tab}$`.getInt("`⟨*id*⟩$_{att}$`")`
⟨*cond*⟩ ::= (⟨*id*⟩`.next(`⟨*int-expr*⟩`)`)

4) 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, an exception is thrown within the program and the method execution is stopped. Such exceptions can be caught using a try/catch structure.

⟨*stmt*⟩ ::= ⟨*db-write*⟩
| `try {` ⟨*db-write*⟩ `} catch (SQLException e) {` ⟨*stmt*⟩* `};`
⟨*db-write*⟩ ::=`con.createStatement().execute("INSERT INTO` ⟨*id*⟩ `VALUES (`{⟨*db-int-expr*⟩$_i$`,`}*⟨*db-int-expr*⟩$_n$`)");`
| `con.createStatement().execute("UPDATE` ⟨*id*⟩$_{tab}$ `SET` ⟨*id*⟩$_{att}$`=`⟨*db-int-expr*⟩ {`WHERE` ⟨*db-cond*⟩}$^?$`");`
| `con.createStatement().execute("DELETE FROM` ⟨*id*⟩ {`WHERE` ⟨*db-cond*⟩}$^?$`");`

5) Transactions management: SQL transactions are managed through the classical commit and rollback statements. A commit statement makes permanent all the changes made to the database by the program during the current transaction, closes this transaction and opens a new one. A rollback statement restores the database to its state at the start of the current transaction, closes this transaction and opens a new one. All pending transactions are supposed committed at the beginning of the tested method.

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

III. A relational symbolic execution algorithm for database programs

A. Inputs and outputs

The symbolic execution algorithm proposed here receives as inputs the SQL DLL code describing the schema of the tested database, the Java code of the tested method and an execution path in this method. It produces as output a relational constraints system, whose solutions are such that when the method is executed with respect to any of these solutions, its execution will follow the given path.

The execution path received as input by our algorithm is supposed to be a finite path in the method's control flow graph [13]. It defines which branch of each encountered If statement was taken, how many times the body of each encountered While loop was executed (this number must be finite), if the Catch clause of each of the encountered Try/Catch statements was executed, and if the method execution was brutally stopped by an uncaught exception thrown by a ⟨*db-write*⟩ statement violating database integrity.

The relational constraints generated as output by our algorithm are expressed using a widely used and well-documented language, offering good analysis tools, called Alloy [22]. Solving this constraints system allows to find values for each of the inputs of the analyzed Java method. These inputs include the content of each database table at method start, the content of every list received as a parameter by the method and the value returned by each '`in.nextInt()`' call made by the method. If the constraint system produced for a given path has no solution, this means that the path is infeasible.

B. Algorithm principle

The principle of the algorithm is to perform a relational symbolic execution of the program path received as input. Each of the different values taken by the method variables and by the database tables during the execution of the path is represented by a corresponding symbolic relational variable. First, symbolic execution generates constraints stating that the variables corresponding to the initial content of each table in the database conform to the database schema. Then, symbolic execution analyzes one by one the method statements executed by the path, in the order in which the path specifies that they are executed. Every time a statement sets or changes the value of a method variable or database table, symbolic execution generates a new constraint stating how the symbolic variable representing this new value can be computed from the other symbolic variables. Every time an If, While, Try/Catch or DB-Write statement is encountered, symbolic execution generates a constraint over the symbolic variables such that when the program is executed with respect to values satisfying this constraint, the execution is guaranteed to take the considered path.

C. Relational constraints generation rules

In this section, we illustrate the execution of the algorithm over the sample database program detailed in figure 1. We detail each step of the symbolic execution process over the path where the While loop is executed once, the Then branch of both the If statements are taken, the Catch clause of the Try/Catch is not executed and no uncaught exception is thrown (lines 1-9, 12-13, 17-20, 4 and 23). At each step, we present the rules used by our algorithm to generate Alloy symbolic variables and constraints. This step by step rules description process allows us to introduce the whole symbolic execution mechanism.

The algorithm always starts by generating symbolic variables and relational constraints for the database tables defined in the database schema. For each table, an Alloy type is first (1) defined so that every symbolic variable representing a set of rows of the table will be a subset of this type. Then, a symbolic variable is created to represent the initial content of the table (2). Finally, constraints are generated to enforce on this content the primary key (3) as well as all the foreign keys (4) and check constraints (5) defined in the table. For the database described in figure 1, the generated Alloy code is as follows:

module testCase *// Name of the Alloy constraints model*
// Type for TABLE department
(1) **sig** department{id : **Int**,numberOfArticles : **Int**}
pred equaldepartment[a:department,b: department] {
a·id = b·id && a·numberOfArticles = b·numberOfArticles}
fact{**all disj** a, b: department | !equaldepartment[a,b]}
(2) **sig** departmentINPDB2 **in** department {}
(3) **fact**{**all disj** a, b: departmentINPDB2 | !((a·id = b·id))}
(5) **fact**{**all** a: department | a·numberOfArticles > 0}
// Type for TABLE article
(1) **sig** article {barcode : **Int**,theDep : **Int**}
pred equalarticle[a: article , b: article]
{a·barcode = b·barcode && a·theDep = b·theDep}
fact{**all disj** a, b: article | !equalarticle [a, b]}
(2) **sig** articleINPDB1 **in** article {}
(3) **fact**{**all disj** a, b: articleINPDB1 |
!((a·barcode = b·barcode))}
(4) **fact**{**all** a: articleINPDB1 |
one b:departmentINPDB2 |a·theDep = b·id}

The second step executed by our algorithm is to generate a relational Alloy type for Java '`List<Integer>`' objects (1), and generic Alloy predicates and functions for their add/remove/get/size methods (2). Lists are modeled in Alloy as an interval of indexes from 0 to a natural number, where each index i is mapped to an integer value representing the i^{th} element of the list. The '`null`' value is represented by the Null singleton type (3).

(1) *// List type definition*
(3) **one sig** Null {}
sig List { index: **set Int**,elems: index →**one Int** }
fun head[l: **set Int**] : **Int**
{ { e:l | (**all** p: l | (!(e=p)) ⇔ (e< p)) } }
fun tail [l: **set Int**] : **set Int**
{ l − { e:l | (**all** p: l | (!(e=p)) ⇔ (e< p)) } }
pred isSuccessive[l: **set Int**]
{ **all** i: tail [l] | (**one** j:l | i = 1·add[j]) }
fact { **all** l: List |
(#l·index=0 | head[l·index]=0 && isSuccessive[l·index])}
pred isNull[l: List + Null] { l=Null }
pred isEmpty[l: List] { #l·index=0 }
(2) *// Methods add − remove − get − size*
pred add[l: List, e: **Int**,newList: List]
{ newList·index = l·index + #l·index &&
(**all** i: l · index | newList·elems[i] = l·elems[i])
&& newList·elems[#l·index] = e }
pred remove[l: List, e: **Int**,newList: List]
{ size [l] ≥ e &&
newList·index = l·index − (#l·index)·sub[1]
&& (**all** i: newList·index | (i< e ⇔
newList·elems[i] = l·elems[i])
&& (i≥ e ⇔ newList·elems[i] = l·elems[i · add[1]])) }
fun get[l: List, i: **Int**] : **Int** { l·elems[i] }
fun size[l: List] : **Int** { #l·index }

The third step executed by our algorithm is to define relational variables for the initial content of each list parameter of the method. For the method considered in this section, the following code is generated:

one sig newarticlesIN1 **in** List + Null {}

The algorithm can then proceed with symbolic execution of the tested method. It follows the path received as input and considers all statements one by one. In the case of the method and path considered in this section, the two first statements to be executed are Assignment statements. Symbolic execution for Assignment creates a new symbolic variable of the correct type to represent the new value of the assigned variable (1) and generates a constraint to specify that this new symbolic variable contains the value computed by evaluating the expression on the right of the '=' symbol (2).

(1) **one sig** iREL1 **in Int** {}
(2) **fact** { iREL1 = 0 }

(1) **one sig** addedarticlesREL2 **in** List {}
(2) **fact** { isEmpty[addedarticlesREL2] }

The second statement in the path is a While statement. As the path specifies that the loop body must be executed this time, a relational constraint is generated to specify that the loop condition should be true:

fact{(!isNull [newarticlesIN1]&&(iREL1< size[newarticlesIN1]))}

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:

one sig errorREL2 **in Int** {} **fact** { errorREL2 = 0 }

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

one sig departmentidIN2 **in Int** {}

Symbolic execution for Select statements creates a new symbolic variable of the type of the table on which the Select query is executed (1), to represent the content of the ResultSet variable. A relational constraint is then generated (2) 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 if it exists:

(1) **sig** departmentsREL3 **in** department {}
(2) **fact** { **all** e: department | (e **in** departmentINPDB2 &&
(e·id = departmentidIN2)) ⇔ e **in** departmentsREL3}

Symbolic execution for Select also creates a symbolic variable to represent the specific order in which the rows were returned by the Select query. This relational variable (1) represents an interval of indexes from 0 to the number of rows returned by the select statement (2), where each index is mapped to one of the returned rows (3):

(1) **sig** departmentsREL3indexes **in Int** {
mapdepartmentsREL3: **one** departmentsREL3 }
(2) **fact** { (#departmentsREL3indexes=0 ||
(head[departmentsREL3indexes] = 0 &&
isSuccessive [departmentsREL3indexes]))
&& #departmentsREL3indexes = #departmentsREL3 }
(3) **fact** { **all disj** a, b: departmentsREL3indexes |
!(a·mapdepartmentsREL3=b·mapdepartmentsREL3) }

As the path specifies that the Then branch of the If statement must be executed this time, a relational constraint is generated to specify that the If condition should be

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 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:

fact{!(#departmentsINTERNALPROG3indexes≥ 1)}

Symbolic execution for Insert creates a new symbolic variable (1) for the new content of the table on which the Insert statement is executed. Then a relational constraint is generated stating that this new variable can be obtained by adding one row with the correct attribute values to the old one (2). Finally, relational constraints are added to specify that, in the considered path, no constraint was violated during insert. In this case, a relational constraint is added to state that there should not be any row in the previous content of the table whose primary key value is the same as in the row inserted by the statement (3):

(1) **sig** departmentRELDB1 **in** department {}
(2) **fact** { **one** e:department | departmentRELDB1 = departmentINPDB2 + e && e·numberOfArticles = 1 && e·id = departmentidIN2 }
(3) **fact** { **no** e:departmentINPDB2 | e·id=departmentidIN2 }

The same process is applied for the second insert statement. A relational constraint (1) is added to state that the inserted article should reference an existing row in the department table:

sig articleRELDB2 **in** article {}
fact { **one** e:article | articleRELDB2 = articleINPDB1 + e && e·theDep = departmentidIN2 && e· barcode = get[newarticlesIN1,iREL1] }
fact{**no** e:articleINPDB1|e·barcode=get[newarticlesIN1,iREL1]}
(1) **fact**{**one** e:departmentRELDB1|e·id=departmentidIN2}

The assignment statement is then symbolically executed:

one sig iREL4 **in Int** {}
fact { iREL4 = (iREL1)·add[1] }

As the path specifies that the Then branch of the If statement must be executed this time, a relational constraint is generated to specify that the If condition should be true:

fact { (errorREL2 = 0) }

Symbolic execution for Commit statements simply does nothing. Symbolic execution for Rollback statements tells the algorithm to use the symbolic variable representing the content of each database table before the last executed Commit statement (saved by the algorithm) to represent the current content of the table.

Symbolic execution for Add and Remove statements creates a new relational variable to represent the new content of each variable which is currently referencing the List object modified by the statement. A constraint is added to relate the old and new object rerferenced by these variables using the add/remove Alloy predicates defined earlier:

one sig addedarticlesREL6 **in** List {}
fact { add[addedarticlesREL2, (iREL4)·sub[1],addedarticlesREL6] }

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

fact{!((! isNull [newarticlesIN1]&&(iREL4< size[newarticlesIN1])))}

As all the statements in the path have been symbolically executed, the algorithm stops and returns the generated Alloy constraints model. The Alloy analyzer [22] can be asked to find a valuation for the defined relational variables which satisfies the constraints, using the commands detailed below. It should be observed that this valuation provides an assignment for the method inputs that will exercise the symbolically executed path, as well as the value of each method variable and database table at the end of the method execution over these inputs. This is particularly useful, as to produce unit tests, one needs both adequate inputs and the corresponding outputs produced by the method.

assert inputsExist{!(newarticlesIN1 **in** List + Null && departmentidIN2 **in Int** && articleINPDB1 **in** article && departmentINPDB2 **in** department) }
check inputsExist

Table I describes the full set of rules used by the algorithm to translate between Java/SQL and Alloy expressions and conditions. Table II details the relational constraints generated by the algorithm for every possible behavior of an Insert, Update or Delete statement. Table III explains the abbreviations used in tables I and II.

IV. Examples of test set generation

The algorithm proposed in this work has been coded in Java and exercised over a set of sample database programs. For each program, a set of execution paths have been selected to satisfy a classical coverage criterion [13]. Each path was symbolically executed and we asked the Alloy analyzer [22] to find solutions for the constraints generated by the algorithm. Each of these computed solutions was checked to be actually input data with respect to which the experimented program was guaranteed to follow the selected path.

The Alloy analyzer is a program which allows to solve Alloy constraints in order to find structures that satisfy them. Basically, it transforms the set of relational constraints into a set of boolean constraints, and solves them using a SAT solver. The process requires to define the maximal scope of the structures the Alloy analyzer should search in [22]. The set of possible structures within this scope is then explored, and the found solutions are returned one by one on demand. In case no solutions to the constraints can be found within a given maximal scope, the solving process is repeated with an increased value for the scope, until it eventually reaches a threshold value.

Three sample database programs were selected to offer a clear illustration of how our algorithm can symbolically

TABLE I. TRANSLATION OF JAVA/SQL EXPRESSIONS AND CONDITIONS INTO ALLOY

Parameters	**alloyOf(Paramaters)**
$\langle id \rangle$	alloyName
$z \in \mathbb{Z}$	z
($\langle int\text{-}expr \rangle_1$ {+|-|*|/} $\langle int\text{-}expr \rangle_2$)	(alloyOf($\langle int\text{-}expr \rangle_1$)).{add | sub | mul | div}[alloyOf($\langle int\text{-}expr \rangle_2$)]
(- $\langle int\text{-}expr \rangle$)	(- (alloyOf($\langle int\text{-}expr \rangle$)))
$\langle id \rangle$.get($\langle int\text{-}expr \rangle$)	get[alloyName,alloyOf($\langle int\text{-}expr \rangle$)]
$\langle id \rangle$.size()	size[alloyName]
$\langle id \rangle_{tab}$.getInt("$\langle id \rangle_{att}$")	$\mathsf{numberOfCallsToNext}_{tab}$.map$\mathsf{alloyName}_{tab}[\langle id \rangle_{att}]$ fact{#$\mathsf{alloyName}_{tab}$indexes >= $\mathsf{numberOfCallsToNext}_{tab}$}
$\langle id \rangle$=null;	fact{ isNull[alloyName] }
$\langle id \rangle$=new ArrayList<Integer>();	fact{ isEmpty[alloyName] }
true false	(0=0) (0=1)
($\langle cond \rangle_1$ {&||} $\langle cond \rangle_2$)	(alloyOf($\langle cond \rangle_1$) {&&|||} alloyOf($\langle cond \rangle_2$))
(! $\langle cond \rangle$)	(!(alloyOf($\langle cond \rangle$)))
($\langle int\text{-}expr \rangle_1$ {<|==|>} $\langle int\text{-}expr \rangle_2$)	((alloyOf($\langle int\text{-}expr \rangle_1$)) {<|=|>} (alloyOf($\langle int\text{-}expr \rangle_2$)))
($\langle id \rangle$==null)	isNull[alloyName]
$\langle id \rangle$.next()	(#alloyNameindexes >= numberOfCallsToNext)
($\langle db\text{-}cond \rangle_1$ {AND|OR} $\langle db\text{-}cond \rangle_2$), row	(alloyOf($\langle db\text{-}cond \rangle_1$,row) {&&|||} alloyOf($\langle db\text{-}cond \rangle_2$,row))
(NOT $\langle db\text{-}cond \rangle$), row	(!(alloyOf($\langle db\text{-}cond \rangle$,row)))
($\langle id \rangle${<|=|>} $\langle db\text{-}int\text{-}expr \rangle$), row	(row.$\langle id \rangle$ {<|=|>} (alloyOf($\langle db\text{-}int\text{-}expr \rangle$,row)))
$z \in \mathbb{Z}$, row	z
$\langle id \rangle$, row	row.$\langle id \rangle$
($\langle db\text{-}int\text{-}expr \rangle_1$ {+|-|*|/} $\langle db\text{-}int\text{-}expr \rangle_2$), row	(alloyOf($\langle db\text{-}int\text{-}expr \rangle_1$,row)).{add | sub | mul | div}[alloyOf($\langle db\text{-}int\text{-}expr \rangle_2$,row)]
(- $\langle db\text{-}int\text{-}expr \rangle$), row	(- (alloyOf($\langle db\text{-}int\text{-}expr \rangle$,row)))
"+$\langle int\text{-}expr \rangle$+", row	alloyOf($\langle int\text{-}expr \rangle$)

TABLE II. RELATIONAL CONSTRAINTS GENERATION FOR $\langle db\text{-}write \rangle$ STATEMENTS

INSERT INTO $\langle id \rangle$ VALUES ($\langle db\text{-}int\text{-}expr \rangle_1$,..., $\langle db\text{-}int\text{-}expr \rangle_i$,..., $\langle db\text{-}int\text{-}expr \rangle_n$)	**if** (no exception thrown in path by this INSERT) { **sig** freshAlloyVar **in** $\langle id \rangle$ {} **fact** { **one** e:$\langle id \rangle$ | freshAlloyVar=alloyName+e && e·att_* = alloyOf($\langle db\text{-}int\text{-}expr \rangle_*$) } **fact** { **no** e:alloyName | e·pk =alloyOf($\langle db\text{-}int\text{-}expr \rangle_{\mathsf{pk}^{pos}}$) } Primary key is **not** violated **fact** { **one** e:fk_*^{tab} | e·fk_*^{pk} = alloyOf($\langle db\text{-}int\text{-}expr \rangle_{\mathsf{fk}_*^{pos}}$) } Every foreign key is **not** violated }**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: **fact** { **one** e:alloyName | e·pk =alloyOf($\langle db\text{-}int\text{-}expr \rangle_{\mathsf{pk}^{pos}}$) } An inserted foreign key value references no row: **fact** { **no** e:fk_i^{tab} | e·fk_i^{pk} = alloyOf($\langle db\text{-}int\text{-}expr \rangle_{\mathsf{fk}_i^{pos}}$) } An inserted attribute can violate a check constraint: **fact** {!($\langle db\text{-}int\text{-}expr \rangle_{\mathsf{co}^{pos}}$ co^{right})} }
UPDATE $\langle id \rangle$ SET $\langle id \rangle_{att}$=$\langle db\text{-}int\text{-}expr \rangle$ WHERE $\langle db\text{-}cond \rangle$	**if** (no exception thrown in path for this UPDATE) { **sig** freshAlloyVar **in** $\langle id \rangle$ {} **fact** { **all** e:alloyName | (alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle$, e) && (**one** y:freshAlloyVar | y·$\mathsf{att}_{*-\{\langle id \rangle_{att}\}}$= e·$\mathsf{att}_{*-\{\langle id \rangle_{att}\}}$ && y·$\langle id \rangle_{att}$ = alloyOf($\langle db\text{-}int\text{-}expr \rangle$,$\langle id \rangle$,e))) || (!(alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle$, e)) && (**one** y:freshAlloyVar | equal$\langle id \rangle$[y,e])) } **fact** { **all** y:freshAlloyVar | **one** e:alloyName | (alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle$, e) && y·$\mathsf{att}_{*-\{\langle id \rangle_{att}\}}$= e·$\mathsf{att}_{*-\{\langle id \rangle_{att}\}}$ && y·$\langle id \rangle_{att}$ = alloyOf($\langle db\text{-}int\text{-}expr \rangle$,$\langle id \rangle$,e))) || (!(alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle$, e)) && (**one** y:freshAlloyVar | equal$\langle id \rangle$[y,e])) } **fact** { **all disj** a, b:freshAlloyVar | !(a·pk = b·pk) } Primary key is **not** violated If $\langle id \rangle_{att}$ = fk_i, updated rows should still reference a row **fact** { **all** a:freshAlloyVar | **one** b:fk_i^{tab} | a·$\langle id \rangle_{att}$ = b·fk_i^{pk} } If $\langle id \rangle_{att}$ = pk, **none** of the updated rows should have been referenced by another row **fact** { **all** e:alloyName | **no** f:ifk_*^{tab} | (alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle$, e) && e·$\langle id \rangle_{att}$ = f·ifk_*^{att}) } } **else** { Logical disjunction between every possible constraint violation given the database schema and this update: Update on primary key leads to duplicate primary keys: **fact** { **some disj** a,b:alloyName | ((alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, a) && alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$,b) && (alloyOf($\langle db\text{-}int\text{-}expr \rangle$,$\langle id \rangle$,a) = alloyOf($\langle db\text{-}int\text{-}expr \rangle$,$\langle id \rangle$, b))) || (!(alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, a)) && alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$,b) && (a·$\langle id \rangle_{att}$ = alloyOf($\langle db\text{-}int\text{-}expr \rangle$,$\langle id \rangle$,b)))) } Trying to update the primary key of a referenced row: **fact** { **some** a:alloyName | alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, a) && (**some** ifk_j^{tab} | (b·ifk_j^{att} =a·$\langle id \rangle_{att}$)) } Update on foreign key let row without row to reference: **fact** { **some** a:alloyName | alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, a) && (**no** b:fk_i^{tab} | b·fk_i^{pk}=alloyOf($\langle db\text{-}int\text{-}expr \rangle$,$\langle id \rangle$,a)) } An inserted attribute violates an arithmetic constraint: **fact** { **some** a:alloyName | alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, a) && !(alloyOf($\langle db\text{-}int\text{-}expr \rangle$,$\langle id \rangle$,a))$\mathsf{co}_i^{right}$)}} }
DELETE FROM $\langle id \rangle$ WHERE $\langle db\text{-}cond \rangle$	**if** (no exception thrown in path for this DELETE) { **sig** freshAlloyVar **in** $\langle id \rangle$ {} **fact** {freshAlloyVar = alloyName − {e:alloyName | alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, e)}} Not trying to delete a referenced row **fact** {**all** e:alloyName | **no** f:ifk_j^{tab} | (alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, e) && e·pk = f·ifk_j^{att})} } **else** { Logical disjunction between every possible constraint violation given the database schema and this update: Trying to delete a referenced row: **fact** {**some** e:alloyName | alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, e) && (**some** f:ifk_j^{tab} | e·pk = f·ifk_j^{att}) } } }

TABLE III. Abbreviations list

Abbreviation	Meaning
freshAlloyVar	A new Alloy variable name that has still not been used in the Alloy code generated so far.
$\mathsf{alloyName}^{superscript}_{subscript}$	**if** ($\langle id\rangle^{superscript}_{subscript}$ refers to a database table name) **then** The symbolic variable that represents the current content of table $\langle id\rangle^{superscript}_{subscript}$) **else** The symbolic variable that represents the current content of the Java variable $\langle id\rangle^{superscript}_{subscript}$
$\mathsf{numberOfCallsToNext}_{sub}$	Number of call made to the next() method of the ResultSet object in variable $\langle id\rangle_{sub}$
att_i	Name of the i_{th} attribute in the list of attributes of table $\langle id\rangle$
$\mathsf{pk}^{superscript}_{subscript}$	Name of the primary key attribute of table $\langle id\rangle^{superscript}_{subscript}$.
pk^{pos}	Position of primary key in the list of attributes of table $\langle id\rangle$
fk^{tab}_i	Name of the table referenced by the i_{th} foreign key in the list of foreign keys of table $\langle id\rangle$
fk^{pk}_i	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^{pos}_i	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 $\langle id\rangle$
fk_i	Name of the foreign key attribute declared by the i_{th} foreign key in the list of table $\langle id\rangle$
ifk^{tab}_i	Name of the table where is declared the i_{th} foreign key referencing table $\langle id\rangle$ in the whole schema
ifk^{att}_i	Name of the foreign key attribute declared by the i_{th} foreign key referencing table $\langle id\rangle$ in the schema
co^{pos}_i	Position of the attribute constrained by the i_{th} check constraint declared in table $\langle id\rangle$
co^{right}_i	Right part of the i_{th} check constraint declared in table $\langle id\rangle$ (i.e. right part of "a>0" is ">0")
xx_* means "for each xx_i" and $\mathsf{xx}_{*-\{y\}}$ means "for each xx_i except from y"	

TABLE IV. Statistics for the selected samples

#	SLOC	SQL statements	Criterion	Maximal scope value	Variables	Constraints	Solving time
1	45	1	Branch coverage	3	67	113	9245 ms
2	56	18	Statement coverage	20	47	120	499685 ms
3	72	4	Branch coverage	3	120	180	16437 ms

execute list management primitives, SQL statements and transactions.

The first sample is a Java method which performs repeated manipulations of integers and lists using assignment, If and While statements. First, two lists are received as parameters. If they are both not null and have the same size, the method reads as many integers from the outside world as the number of elements in the lists. A third list is created using these integers in the order in which they were read. The elements of the three lists are then compared. If the second list is an inverted version of the first one and if the third list is a copy of the first list where the value of each element was doubled, then the three lists are inserted into a database table. The database schema and the way the lists are inserted in this database constrain the elements in the first list to be different from each other and their value to range between one and five.

The second sample is a Java method which performs repeated reads and writes in a database containing four tables that represent customers making purchases of products. Customers with few purchases are prospect customers. First, the program inserts a new customer and new purchases into the database. Then it computes the total number and cost of the purchases made by each customer, as well as the total number of purchases for each product, and then updates the corresponding customer and product attributes. All unpurchased products are deleted, and the name of the customers having made no purchase is changed. Customers whose total count and cost of purchase is lower than two are registered as prospect customers. Finally, a product is replaced by another one in every purchase details, and this product is deleted.

The third sample is a Java method which mixes SQL statements with traditional Java code and uses SQL transactions. The database contains two tables that represent authors writing theater plays. The code contains two transactions. During a first transaction, some authors are added and some removed from the database. During a second transaction, plays are added for the previously added authors, and some statistics are computed for each author. If a database schema constraint is violated by a SQL statement in one of the two transactions, this whole transaction is cancelled and the database is rolled back to the state it was when the transaction was launched.

The source code of these three samples, the Alloy constraints generated by our algorithm and the resulting generated test cases, as well as all the detailed performance-related information can be found on the web[1]. The main information are synthesized in table IV, including the SLOC and the number of SQL statements in the sample, the coverage criterion satisfied by the generated test set, the maximal scope value used in the Alloy analyzer to solve the constraints produced for the sample, the total number of relational variables and constraints solved to find a test set for the sample, and the time for solving them. These measures were realized 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 using the MiniSat solver. The oder-of-magnitude difference in constraints solving time for the second sample is due to the use of a larger maximal scope value in the Alloy analyzer.

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 modeling and use of the constraint generation rules conceptually simpler. This does not 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 modeled into Alloy (see e.g. [23]). These reasons explain why the previously described

[1]See http://info.fundp.ac.be/ mmr/scam13

samples manipulate integer values, like author or play names, which are not usually typed as integers.

V. Conclusion and related work

In this work, we have proposed and demonstrated a complete algorithm to execute symbolically simple static Java methods using SQL CRUD statements and transactions to interact with a relational database, subject to integrity constraints. Given the schema of the database, the code of the method and an execution path in this method, the algorithm generates a relational symbolic variable for each potential value taken by a method variable or database table before and during the path execution. It generates as well an Alloy relational constraints model constraining these relational symbolic variables to guarantee the execution of the considered path. Any solution to the produced relational constraints describes input data for the method (as well as the corresponding output), with respect to which the program can be executed and is guaranteed to follow the considered execution path. Given a set of execution paths to test in the database program chosen using an existing algorithm [12] to satisfy a given code coverage criterion [13], the proposed algorithm can be used to generate test data, including database initial and final states, for each path in the set.

An early approach that has considered test data generation for imperative programs interacting with a relational SQL database is [19]. The paper proposes to transform the program, thereby inserting new variables representing the database structure, and translating all SQL statements (and thus integrity checks) into imperative program code. Classical white-box testing approaches can then be applied to the modified program. In [20], the authors propose an algorithm for testing an imperative program performing SELECT queries on a relational SQL database, based on concolic execution [11]. Concolic execution is an extension [12] of the classical symbolic execution used in this work. It mixes symbolic execution with a simultaneous concrete dynamic execution of the considered execution paths. In [20], the program is first run on random input data and on a randomly populated input database. Given such a dynamic exploration of an execution path of the program, the authors model and solve the problem of finding other inputs, allowing to explore dynamically another execution path, as a set of integer and string constraints over the quantity and field contents of the records in the database and over the input variables of the tested program. In [24], authors adopt a similar concolic approach where the program is executed on a parameterized mock database. In [25] and [26], authors adapt this approach to testing of programs running on an existing database, so that input test data can be selected in this database instead of being generated from scratch. In [27], the same concolic approach is applied considering advanced code coverage criteria. Finally, translation between database schemas/programs/queries and Alloy has been considered in other contexts [28], [29], [30].

Compared to [20], our approach does not need to transform the original program by adding potentially complex pieces of imperative code, to simulate the execution of SQL statements by a DBMS. Compared to [20], [24], [25], [26], [27], our approach handles Insert, Update and Delete statements, as well as transactions management primitives (Commit and Rollback) and database integrity constraints (like primary keys or foreign keys) which are crucial components of database applications. On the other hand, our approach only considers SQL statements whose structure is completely defined statically, where the concolic process used in [20], [24], [25], [26], [27] can allow to account, at least at some extent, for dynamically crafted SQL statements. This point should be tempered by the fact that our approach evaluates a particular path in the program at a time, which makes easier to reassemble statically the SQL statements supposed to be dynamically assembled during the execution of the path.

In future work, it would be relevant to evaluate how and up to which extent the symbolic execution mechanism proposed can be practically generalized to a larger part of the Java and SQL languages. Similarly, it should be investigated how and up to which extent fully-dynamic SQL can be integrated with our approach, possibly relying on static analysis [31], [32], [33] or on concolic execution. These tasks notably pave the way towards an extensive evaluation of the method on large scale industrial systems. Secondly, it happens frequently that SQL statements have a non-deterministic behavior, either because the statement has a non-deterministic semantics, or because the database is modified concurrently by several programs. How the approach proposed here can handle at best such non-deterministic behaviors remains a topic for further research. Thirdly, our approach allows to be used with respect to any classical code coverage criterion based on the notion of an execution path. Nevertheless, several works [34], [35], [36], [37], [38] propose test adequacy criteria particularly adapted to the testing of database-driven programs. Integrating such particular coverage criteria into our constraint-based approach is a topic of ongoing research. Finally, [39] and [40] propose new approaches to solve and evaluate the satisfiability of sets of relational constraints. It would be relevant to study how the technique proposed here could benefit from these approaches, notably to detect unfeasible paths more efficiently or to offer improved constraints solving time performance.

Acknowledgment

This work has been funded by the Belgian Fund for Scientific Research (F.R.S.-FNRS). The authors would like to thank Vincent Englebert for useful discussions and the anonymous reviewers for their valuable comments.

References

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

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

[3] R. Ramler and 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*, ser. AST '06. New York, NY, USA: ACM, 2006, pp. 85–91.

[4] J. Malburg and G. Fraser, "Combining search-based and constraint-based testing," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 436–439.

[5] P. McMinn, "Search-based software test data generation: a survey: Research articles," *Softw. Test. Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, Jun. 2004.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," in *In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS*, 2006.

[7] F. Degrave, T. Schrijvers, and W. Vanhoof, "Towards a framework for constraint-based test case generation," in *Proceedings of the 19th international conference on Logic-Based Program Synthesis and Transformation*, ser. LOPSTR'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 128–142.

[8] A. Gotlieb, B. Botella, and M. Rueher, "Automatic test data generation using constraint solving techniques," *SIGSOFT Softw. Eng. Notes*, vol. 23, no. 2, pp. 53–62, Mar. 1998.

[9] C. Meudec, "ATGen: automatic test data generation using constraint logic programming and symbolic execution," *Software Testing Verification and Reliability*, vol. 11, no. 2, 2001.

[10] A. J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction procedure for test data generation," *Softw. Pract. Exper.*, vol. 29, no. 2, pp. 167–193, Feb. 1999.

[11] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 263–272, Sep. 2005.

[12] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communication of the ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[13] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997.

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

[15] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[16] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Softw. Eng.*, vol. 2, no. 3, pp. 215–222, May 1976.

[17] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, Sep. 1991.

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

[19] M. Y. Chan and 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.

[20] 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*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 151–162.

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

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

[23] M. L. Crane and J. Dingel, "Runtime conformance checking of objects using alloy," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, pp. 2 – 21, 2003.

[24] K. Taneja, Y. Zhang, and 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.

[25] C. Li and C. Csallner, "Dynamic symbolic database application testing," in *Proceedings of the Third International Workshop on Testing Database Systems*, ser. DBTest '10. New York, NY, USA: ACM, 2010, pp. 7:1–7:6.

[26] K. Pan, X. Wu, and T. Xie, "Generating program inputs for database application testing," in *Proc. 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, November 2011.

[27] ——, "Database state generation via dynamic symbolic execution for coverage criteria," in *Proceedings of the Fourth International Workshop on Testing Database Systems*. New York, NY, USA: ACM, 2011.

[28] A. Cunha and H. Pacheco, "Mapping between alloy specifications and database implementations," in *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, ser. SEFM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 285–294.

[29] D. Jackson and M. Vaziri, "Finding bugs with a constraint solver," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 5, pp. 14–25, Aug. 2000.

[30] S. A. Khalek and S. Khurshid, "Systematic testing of database engines using a relational constraint solver," in *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 50–59.

[31] G. Wassermann, C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, Sep. 2007.

[32] S. Thomas, L. Williams, and T. Xie, "On automated prepared statement generation to remove sql injection vulnerabilities," *Inf. Softw. Technol.*, vol. 51, no. 3, pp. 589–598, Mar. 2009.

[33] A. Christensen, A. M,àö,àèller, and M. Schwartzbach, "Precise analysis of string expressions," in *Static Analysis*, ser. Lecture Notes in Computer Science, R. Cousot, Ed. Springer Berlin Heidelberg, 2003, vol. 2694, pp. 1–18.

[34] W. G. J. Halfond and A. Orso, "Command-form coverage for testing database applications," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 69–80.

[35] G. M. Kapfhammer and 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*, ser. ESEC/FSE-11. New York, NY, USA: ACM, 2003, pp. 98–107.

[36] M. J. Suárez-Cabal and 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*, ser. SIGSOFT '04/FSE-12. New York, NY, USA: ACM, 2004, pp. 253–262.

[37] M. J. Suarez-Cabal and J. Tuya, "Structural coverage criteria for testing SQL queries," *Journal of Universal Computer Science*, vol. 15, no. 3, pp. 584–619, 2009.

[38] C. Zhou and P. Frankl, "Mutation testing for java database applications," in *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ser. ICST '09. Washington, DC, USA: IEEE Comp. Soc., 2009, pp. 396–405.

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

[40] E. Torlak, F. S.-H. Chang, and D. Jackson, "Finding minimal unsatisfiable cores of declarative specifications," in *Proc. of the 15th international symposium on Formal Methods*, ser. FM '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 326–341.