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

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

Publication date:
2012

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

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

Michaël Marcozzi*
Faculty of Computer Science
University of Namur
Rue Grandgagnage, 21
Namur, Belgium
mmr@info.fundp.ac.be

Wim Vanhoof
Faculty of Computer Science
University of Namur
Rue Grandgagnage, 21
Namur, Belgium
wva@info.fundp.ac.be

Jean-Luc Hainaut
Faculty of Computer Science
University of Namur
Rue Grandgagnage, 21
Namur, Belgium
jlh@info.fundp.ac.be

ABSTRACT

In so-called constraint-based testing, symbolic execution is a common technique used as a part of the process to generate test data for imperative programs. Databases are ubiquitous in software and testing of programs manipulating databases is thus essential to enhance the reliability of software. This work proposes and evaluates experimentally a symbolic execution algorithm for constraint-based testing of database programs. First, we describe SimpleDB, a formal language which offers a minimal and well-defined syntax and semantics, to model common interaction scenarios between programs and databases. Secondly, we detail the proposed algorithm for symbolic execution of SimpleDB models. This algorithm considers a SimpleDB program as a sequence of operations over a set of relational variables, modeling both the database tables and the program variables. By integrating this relational model of the program with classical static symbolic execution, the algorithm can generate a set of path constraints for any finite path to test in the control-flow graph of the program. Solutions of these constraints are test inputs for the program, including an initial content for the database. When the program is executed with respect to these inputs, it is guaranteed to follow the path with respect to which the constraints were generated. Finally, the algorithm is evaluated experimentally using representative SimpleDB models.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Code inspections and walk-throughs*; F.4.1 [**Mathematical Logic and formal languages**]: Mathematical Logic—*Logic and constraint programming*; H.2.0 [**Database Management**]: General—*Security, integrity, and protection*

General Terms

Algorithms, Languages, Reliability

Keywords

Constraints-Based Testing, Test Inputs Generation, Symbolic Execution, Database Applications, Relational Constraints, Alloy

1. INTRODUCTION

Testing [16, 17] is a complex process which constitutes the primary approach to improve the reliability of software, motivating [31] much research to develop relevant automated approaches to test the quality of all aspects of software. In this work, we consider the automation of test inputs generation for functional unit testing of database-driven software.

Many advanced techniques have been developed so far (see e.g. [39]) to automate the generation of adequate test data for proper testing of units of code, with regard to their expected functions, and independently of any interaction with a database. Many approaches have also been proposed (e.g. [1, 7, 19, 27, 36]) to automate the generation of test database contents, to be used for testing databases and database management systems (DBMS), independently of the data-flow in the programs interacting with the database or DBMS under test. Databases are nowadays ubiquitous in software and many units of code interact intensively with a large, persistent and highly-structured relational database [6]. But barely few works (see [3, 10, 22]) 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 a technique to generate simultaneously input database contents and program input values, in order to test an imperative program interacting with a relational database through SQL statements. The technique adopts a white-box and structural testing approach. Given a finite set of execution paths in the control flow graph of the program, satisfying a given code coverage criterion, it generates, for each path in the set, test inputs leading to the execution of this path, using static forward symbolic execution.

Such symbolic execution was introduced by King [20], Clarke [4] and DeMillo & Offut [9] and has been advocated

*F.R.S.-FNRS Research Fellow

Working Paper University of Namur

in many constraint-based test data generation techniques (e.g. [2, 8, 12, 26, 28]), in the context of programs having no interaction with a database. Given a path through the program code to be tested, symbolic execution builds 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 to be tested. In order to build these constraints, symbolic execution considers the static single assignment form of the program and it expresses the control dependencies imposed by the execution of the particular execution path to be tested.

The technique proposed in this work adapts symbolic execution to the particular case of database programs. The core strategy of the technique is, as we proposed in [22], to model every variable of the program and every table (which is, fundamentally, 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 operation over these relational variables. By applying the classical static 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 constraints described by the relational schema of this database, such as, for example, the primary key or foreign key constraints. The technique proposed in this work expresses these schema constraints 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 two main contributions of this work can be summarized as follows.

First, we propose a formal language, called SimpleDB, to facilitate the formal analysis of database programs. SimpleDB refines end extends the ImperDB language that we defined in [22], introducing database schema specification within the language, transactions management, use of lists as input variables and a fully defined syntax and semantics. A SimpleDB *model* describes both an imperative program to be tested and the schema of the relational database manipulated by this program. SimpleDB is a tiny formal language, offering only a minimal set of classical well-defined primitives necessary for building database programs. These simplicity and formalness allow to specify algorithms able to automate testing of database programs in a simple, fast and rigorous way. But despite its simplicity, SimpleDB allows to model a large and interesting part of the possible interaction scenarios between a real program and a real database. Notably, SimpleDB proposes basic mechanisms for throwing and catching exceptions. This is an important aspect of the language, as it allows for a clean testing of all execution paths, including those that may lead to an erroneous interaction between the program and the database.

Secondly, we introduce, detail and evaluate a complete algorithm based on symbolic execution to generate test inputs for SimpleDB models. This algorithm extends, concretizes and permits experimentation to validate the raw strategy that we suggested in [22]: given a SimpleDB model and a path in the program described by this model, the algorithm generates the corresponding relational input constraints system in the Alloy language [14]. A prototype of the algorithm has been evaluated using sample SimpleDB models, and the generated constraints have been solved using the Alloy Analyzer [14], showing promising results.

The remainder of this paper is organized as follows. Section 2 details the syntax and semantics of the SimpleDB language. In section 3, we describe and illustrate the symbolic execution algorithm, able to generate Alloy input constraints for any finite execution path in any SimpleDB model. We provide experimental results in section 4, showing the efficiency of the algorithm over several sample SimpleDB models. Finally, some conclusions, related and future work are provided in section 5.

2. SIMPLEDB: A MINIMALIST SYNTAX AND SEMANTICS FOR DB PROGRAMS

In this section, we detail the syntax and semantics of the SimpleDB language using a step by step approach. For each step, a part of the BNF grammar, describing some of the syntactic constructs of the language, is presented. For each syntactic construct, additional syntactic rules are explained, as well as the construct's semantics. The major language design choices are discussed if needed. The chosen notation for the BNF grammar includes some additional meta-symbols: {...} (grouping), * (repetition zero or more times) and $^+$ (repetition one or more times). When a single nonterminal appears several times in a single production, subscript notation allows to distinguish between the occurrences.

A sample SimpleDB model is provided in figure 1. This model defines a database with two tables: one for authors and one for the theatrical plays these authors write. The number of plays written by an author is stored for each author. The model also defines a program manipulating this database: the program adds a set of new plays to the database and updates the authors' plays counts. If the author of an added play was not in the database, it is added to the database as well. The plays are inserted one by one in isolated transactions.

2.1 Model

⟨*model*⟩ ::= `MODEL` ⟨*id*⟩ ⟨*db-schema*⟩ ⟨*program*⟩ `ENDMODEL`

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

A SimpleDB model is given a name and details first the relational schema of the database and subsequently the code of the program working on this database.

2.2 Database schema

⟨*db-schema*⟩ ::= ⟨*table*⟩*

⟨*table*⟩ ::= `TABLE` ⟨*id*⟩ `(` ⟨*attrib*⟩$^+$ ⟨*pr-key*⟩ ⟨*f-key*⟩* ⟨*constr*⟩* `);`

Figure 1: SimpleDB model with program inserting plays and updating author's count in a plays database

```
MODEL example
TABLE author (name,numberOfPlays,PRIMARY KEY(name),numberOfPlays > 0);
TABLE play (title,theAuthor,PRIMARY KEY(title),FOREIGN KEY(theAuthor) REFERENCES author);
COMMIT();
LOAD(newPlays);
WHILE (!(newPlays=NIL)) DO
        error = 0;
        READ(authorName);
        authors = SELECT name FROM author WHERE (name = authorName);
        isEmpty = CATCH(NEXT(authors));
        IF (isEmpty=1) THEN
                INSERT INTO author VALUES (authorName,1);
        ELSE
                UPDATE author SET numberOfPlays = (numberOfPlays + 1) WHERE (name = authorName);
        ENDIF;
        error = CATCH(INSERT INTO play VALUES (newPlays.HEAD,authorName));
        IF (error=0) THEN
                COMMIT();
        ELSE
                ROLLBACK();
        ENDIF;
        newPlays = newPlays.TAIL;
ENDWHILE;
COMMIT();
ENDMODEL
```

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. Each table is identified by its name, contains at least one attribute and endorses exactly one primary key. Foreign keys and additional constraints can be declared for a table. Semantics of schema definition primitives in SimpleDB is the same as defined in the classical SQL DDL semantics [11].

⟨*attrib*⟩ ::= ⟨*id*⟩ `,`

⟨*pr-key*⟩ ::= `PRIMARY KEY (` ⟨*id*⟩ `)`

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

⟨*constr*⟩ ::= `,` ⟨*id*⟩ {`<` | `=` | `>`} ⟨*natural*⟩

⟨*natural*⟩ ::= {1 | ... | 9}{0 | ... | 9}$^+$ | {0 | ... | 9}

Each attribute in a table is simply identified by its name and is of integer type. The exclusive use of integer values in SimpleDB models does not limit the expressive power of our model of database programs since all other usual primitive types such as booleans, strings, and floating point numbers, but also data structures such as sets, arrays and matrices, can easily be mapped to integers and/or simulated using lists of integers. It does, however, make both the modeling and the use of a constraint solver conceptually simpler. These last reasons explain why the examples of SimpleDB models used in this work manipulate values which are not usually integers, like author names in the example of table 1.

Among the attributes of a table, one is declared to be the primary key of the table. Any attribute can be be declared to be a foreign key referencing another table in the schema. An attribute can reference several different tables. 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. Cycles in tables referencing are not allowed. Simple arithmetic constraints can be declared over each of the attributes of a table.

2.3 Program code

SimpleDB allows to write imperative programs processing integers and lists of integers and interacting with a database through transactions involving one or several simple SQL Select, Insert, Update or Delete statements.

⟨*program*⟩ ::= `COMMIT();` ⟨*stmt*⟩* `COMMIT();`

The code of the program is a sequence of statements which starts and ends with a commit statement. Variables must not be declared in SimpleDB. A variable can be used in all the statements subsequent to its initialization through a read, load or assignment (⟨*id*⟩`=` ...) statement. A variable cannot be used outside of the code block where it was initialized. This means that a variable initialized inside a loop body/if branch cannot be used in the statements outside of this loop body/if branch. A variable can be of three types: integer, list of integers or table. The type of a variable is fixed by its initialization statement and any type change in a subsequent statement will result in a compile-time error. SQL statements semantics in SimpleDB is the same as defined by the classical SQL DML semantics [11].

2.3.1 Imperative statements and lists management

⟨*stmt*⟩ ::= `IF` ⟨*cond*⟩ `THEN` ⟨*stmt*⟩*$_{then}$ `ELSE` ⟨*stmt*⟩*$_{else}$ `ENDIF ;`
| `WHILE` ⟨*cond*⟩ `DO` ⟨*stmt*⟩* `ENDWHILE ;`
| ⟨*id*⟩ `=` ⟨*expr*⟩ `;` (Assignment of variable ⟨*id*⟩)

⟨*cond*⟩ ::= `TRUE` (Logical true value)
| `FALSE` (Logical false value)
| `(`⟨*cond*⟩$_1$ {`&&` | `||`} ⟨*cond*⟩$_2$`)` (Logical conjunction and disjunction)
| `(!` ⟨*cond*⟩`)` (Logical negation)
| `(`⟨*int-expr*⟩$_1$ {`<` | `=` | `>`} ⟨*int-expr*⟩$_2$`)` (Arithmetic comparison)
| `(`⟨*id*⟩ `= NIL)` (List emptiness test for list in variable ⟨*id*⟩)

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

⟨*int-expr*⟩ ::= ⟨*id*⟩ (Integer-typed variable)
| ⟨*natural*⟩ (Natural number)
| (⟨*int-expr*⟩$_1$ {+ | - | * | /} ⟨*int-expr*⟩$_2$) (Arithmetics)
| (- ⟨*int-expr*⟩) (Unary minus)
| ⟨*id*⟩.HEAD (First element of list in variable ⟨*id*⟩)

⟨*list-expr*⟩ ::= ⟨*id*⟩ (List of integers-typed variable)
| NIL (Empty list)
| [⟨*int-expr*⟩ , ⟨*list-expr*⟩] (Appending integer ⟨*int-expr*⟩ at the beginning of list ⟨*list-expr*⟩)
| ⟨*id*⟩.TAIL (List in variable ⟨*id*⟩ without its first element)

SimpleDB allows to control the program flow using classical condition and loop statements. Classical variable assignment statement evaluates an integer or list of integers expression and assigns the obtained value to a variable of the program. SimpleDB allows all basic logic operations in if and while conditions, as well as arithmetic comparisons and list emptiness testing. SimpleDB allows full arithmetics over integers and has the basic operations over lists of integers (concatenation of an element to a list and selecting the head, respectively, tail of a list). Lists are immutable.

2.3.2 Interacting with the outside world

⟨*stmt*⟩ ::= READ (⟨*id*⟩) ;
| LOAD (⟨*id*⟩) ;

Read (respectively Load) statement assigns an integer (respectively list of integers) value from the outside world to one of the variables of the program. This models different kinds of interaction between the program and the outside world (except from the interaction with the database) such as parameters received from a calling program, user prompt, network access, reading from a file, etc.

2.3.3 Reading data from the database

⟨*stmt*⟩ ::= ⟨*id*⟩=SELECT{⟨*id*⟩$_i$,}*⟨*id*⟩$_n$ FROM ⟨*id*⟩$_{tab}$ WHERE ⟨*db-cond*⟩;
| NEXT (⟨*id*⟩) ;
| ⟨*id*⟩ = CATCH (NEXT (⟨*id*⟩)) ;

⟨*db-cond*⟩ ::= TRUE
| FALSE
| (⟨*db-cond*⟩$_1$ {&& | ||} ⟨*db-cond*⟩$_2$)
| (! ⟨*db-cond*⟩)
| (⟨*id*⟩ {< | = | >} ⟨*int-expr*⟩) (⟨*id*⟩ refers to an attribute of the table being read/modified)

In order to access database data within the program, a SQL Select query must be processed over the database content and the table returned by this query must be assigned to a variable of the program. These two steps are executed by the ⟨*id*⟩ = SELECT ... statement. Subsequently, the table in the assigned variable can be accessed by the program, but only one row at a time, using a cursor pointing at the single readable row of the table. After assigning a table to a variable, the Next statement must be called over this variable to set the cursor in front of the first row of the table in the variable. Every subsequent Next statement will move the cursor one row ahead. The integer value of attribute ⟨*id*⟩$_{att}$ in the row pointed to by the cursor of the table assigned to variable ⟨*id*⟩$_{tab}$ can be accessed using the following syntax:

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

If the cursor is in front of the last row of the table or if the Select query returned an empty table, every call to the Next statement will result in an exception to be thrown within the program. A Next statement can be wrapped in a Catch statement. The later will set an integer-typed variable to 1 if an exception has been thrown by the statement it wraps, and to 0 otherwise. If an exception remains uncaught, the program immediately terminates, revealing a potential fault in the code or database design.

Unlike the classical SQL semantics and in order to avoid any non-deterministic behavior, the SimpleDB semantics requires that the tables returned by Select queries are always sorted by ascending order of the primary key attribute values. Put simply, the following SimpleDB query over the database in figure 1:

SELECT name,numberOfPlays
FROM author
WHERE (numberOfPlays > 10)

is always equivalent to the following SQL query :

SELECT name,numberOfPlays
FROM author
WHERE (numberOfPlays > 10)
ORDER BY name

In the Where clause of a SQL statement, as well as in the Set clause of an Update SQL statement, ⟨*id*⟩ in ⟨*int-expr*⟩ can represent both a program variable and an attribute of the read or written table. In case of potential ambiguity, ⟨*id*⟩ always represents the attribute in the table.

2.3.4 Writing data into the database

⟨*stmt*⟩ ::= ⟨*db-write*⟩ ;
| ⟨*id*⟩ = CATCH (⟨*db-write*⟩) ;

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

SQL Insert, Update and Delete statements allow the program to write data into the database. If the execution of a such a statement provokes a violation of one of the database schema constraints, the database remains unmodified by the statement and an exception is thrown within the program. As with Next statements, exceptions either can be caught using a Catch statement or make the program terminate, revealing a potential fault in the code or database design.

2.3.5 Transactions management

⟨*stmt*⟩ ::= COMMIT() ;
| ROLLBACK() ;

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. In any SimpleDB program, a new transaction is started at program start, and all uncommitted changes are saved at program end, using a Commit statement.

3. AN ALGORITHM FOR SYMBOLIC EXECUTION OF SIMPLEDB PROGRAMS

3.1 Inputs and outputs

The symbolic execution algorithm proposed here receives as inputs a SimpleDB model and an execution path in the program defined within this model. It produces as output a relational constraints system, whose solutions are such that when the program 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 path in the program's control flow graph. In particular, it defines which branch of each of the encountered If statements was taken, how many times the body of each of the encountered loops was executed and which of the encountered Next and ⟨*db-write*⟩ statements threw an exception. In the case of ⟨*db-write*⟩ statements throwing exceptions, the path also specifies which database schema constraint caused the exception to be thrown.

The constraints system generated as output by our algorithm allows to find values for the inputs of the input SimpleDB program. Inputs of a SimpleDB program are composed of the content of each database table defined by the model at program start, as well as all the values gathered from the outside world by the Read and Load statements executed by the program.

3.2 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 program 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 program statements executed by the path, in the order in which the path specifies they are executed. Every time a statement sets or changes the value of a program variable or of a database table, symbolic execution generates a new constraint stating how the symbolic variable representing this new value can be computed as a function of the other symbolic variables. Every time a statement offers a choice in the way it can be executed that depends on the values of the program variables and of the database tables (conditions, loops, next and ⟨*db-write*⟩ statements), symbolic execution generates a constraint over the symbolic variables such that when the program is executed with respect to values satisfying the constraint, the execution is guaranteed to take the path under consideration.

Once all the statements encountered along the path have been analyzed, the set of relational constraints generated during analysis can be solved to find values for the relational symbolic variables that satisfy these constraints. If such a solution exists, the values of the symbolic variables corresponding to the program inputs constitute some test input data that will guarantee the execution of the considered execution path. If the constraints have no solution, then the considered path is infeasible.

The generated relational constraints are expressed using a widely used and well-documented language, offering good analysis tools, called Alloy [14].

3.3 Symbolic variables and relational constraints generation rules

In this section, we illustrate the execution of the algorithm over the sample SimpleDB model detailed in figure 1. We detail each step of the symbolic execution process over the path where the While loop is executed once, the Next statement throws an exception, the THEN branch of both the If statements are taken, and both the two Insert statements do not violate any database integrity constraint. 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 of the algorithm.

The algorithm always starts by generating symbolic variables and relational constraints for the database tables defined within the model. For each table, an Alloy type is first defined so that every symbolic variable representing the content of the table will be of this type. Then a symbolic variable is created to represent the initial content of the table (1). Finally, constraints are generated to enforce on this content the primary key (2) as well as all the foreign keys (3) and arithmetic constraints (4) defined in the table. For the model of table 1, the generated Alloy code is as follows. The reader should note that symbolic variables will be always created using the **sig** keyword followed by the name of the variable and by its type. Here a symbolic variable *authorINPUTDB2* was defined to represent the initial content of the table AUTHOR, and a symbolic variable *playINPUTDB1* was defined to represent the initial content of the table PLAY. Relational constraints will be always generated using the **fact** keyword.

```
module example // Name of the Alloy constraints model
// START of Alloy type definition for table AUTHOR
sig author{name : Int,numberOfPlays : Int}
pred equalauthor[a:author,b: author]
{a·name = b·name && a·numberOfPlays = b·numberOfPlays}
fact{all disj a,b: author | !equalauthor[a,b]}
// END of Alloy type definition for table AUTHOR

(1) sig authorINPUTDB2 in author {}
(2) fact{all disj a,b:authorINPUTDB2 | !((a·name=b·name))}
(4) fact{all a: author | a·numberOfPlays > 0}

// START of Alloy type definition for table PLAY
sig play{theAuthor : Int,title : Int}
pred equalplay[a:play,b: play]
{a·theAuthor = b·theAuthor && a·title = b·title}
fact{all disj a,b: play | !equalplay[a,b]}
// END of Alloy type definition for table PLAY

(1) sig playINPUTDB1 in play {}
(2) fact{all disj a,b: playINPUTDB1 | !((a·title = b·title))}
(3) fact{all a: playINPUTDB1 |
      one b:authorINPUTDB2 |a·theAuthor = b·name}
```

The second step executed by our algorithm is to generate a relational Alloy type for lists, which are not supported by default in Alloy:

```
one sig Nil {}
sig List {head: Int,tail : List + Nil}
```

The algorithm can then proceed with symbolic execution of the program defined in the model. The algorithm considers each statement one by one and follows the path received

as input. In the case of the example model and path considered in this section, the Load statement is symbolically executed first. Symbolic execution for Load simply creates a new symbolic variable of type list to represent the loaded value:

```
one sig newplaysINPUTPROG1 in List + Nil {}
```

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. In this case, it means that the current content (represented by the symbolic variable *newplaysINPUTPROG1*) of the SimpleDB variable *newPlays* should not be the empty list:

```
fact{!(newplaysINPUTPROG1 = Nil)}
```

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. Symbolic execution for Assignment creates a new symbolic variable of the correct type to represent the new value of the assigned variable and generates a constraint to specify that this new symbolic variable contains the value that can be computed by evaluating the expression on the right of the "=" symbol:

```
one sig errorINTERNALPROG1 in Int {}
fact{errorINTERNALPROG1 = 0}
```

Symbolic execution for Read simply creates a new symbolic variable of type integer to represent the read value:

```
one sig authornameINPUTPROG2 in Int {}
```

Symbolic execution for Select creates a new symbolic variable of the type of the table on which the Select query is executed, to represent the Select result table. A relational constraint is then generated to specify that a row is part of the Select result table if and only if it is part of the current content of the table on which the Select query is executed and that it enforce the WHERE condition of the Select query:

```
sig authorsINTERNALPROG2 in author {}
fact{all e: author | (e in authorINPUTDB2
    && (e·name = authornameINPUTPROG2))
    ⇔ e in authorsINTERNALPROG2}
```

Symbolic execution for Catch first proceeds with symbolic execution of the statement wrapped by the Catch and then acts as an Assignment (2) statement. Symbolic execution for Next will depend on the cursor state (which must be stored by the algorithm) of the "nexted" SimbleDB variable and on whether the Next statement should throw an exception or not according to the input path:

1. No call to Next made before:
   (a) No exception is thrown: Add a relational constraint stating that the symbolic variable corresponding to the current content of the "nexted" SimbleDB variable should contain at least one element.
   (b) An exception is thrown: Add a relational constraint stating that the symbolic variable corresponding to the current content of the "nexted" SimbleDB variable should contain no element. (1)
2. Call(s) to Next made before and no exception thrown:
   (a) No exception is thrown: Create a new symbolic variable to represent the content of the "nexted" SimpleDB variable. Generate a relational constraint stating that this new variable can be obtained from the old one by removing the element with the lowest primary key value from the old variable. Generate a relational constraint stating that the newly created symbolic variable should contain at least one element.
   (b) An exception is thrown: Add a relational constraint stating that the symbolic variable corresponding to the current content of the "nexted" SimbleDB variable should contain exactly one element.
3. Last call to Next threw an exception:
   (a) No exception is thrown: This is not allowed by the SimpleDB semantics.
   (b) An exception is thrown: Do nothing.

In the example considered here, the algorithm chose option 1.b:

```
(1) fact{#authorsINTERNALPROG2=0}
(2) one sig isemptyINTERNALPROG3 in Int {}
(2) fact{isemptyINTERNALPROG3=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. In this case, in means that the current value (represented by the symbolic variable *isemptyINTERNALPROG3*) of the SimpleDB variable *isEmpty* should be one:

```
fact{(isemptyINTERNALPROG3 = 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 no constraint was violated during insert. In the example considered here, 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 authorINTERNALDB1 in author {}
(2) fact{one e:author |
    authorINTERNALDB1=authorINPUTDB2+ e
    && e·numberOfPlays = 1
    && e·name = authornameINPUTPROG2}
(3) fact{no e:authorINPUTDB2 |
    e· name=authornameINPUTPROG2}
```

Here again, symbolic execution for Next statement first proceeds with symbolic execution of the wrapped statement and then acts as an Assignment statement (5). Symbolic execution for this Insert statement acts identically as for the previous Insert (1)(2)(3), but a relational constraint is added as well, stating that the newly inserted row references an existing row in the current content of the Author table (4).

Table 1 details the symbolic variables and relational constraints generated by the algorithm for every possible behavior of an Insert, Update or Delete statement. Table 2 describes the rules used by the algorithm to translate between SimpleDB and Alloy expressions and conditions. Table 3 explains the abbreviations defined in table 1 and 2.

```
(1) sig playINTERNALDB2 in play {}
(2) fact{one e:play | playINTERNALDB2=playINPUTDB1+ e
    && e·theAuthor = authornameINPUTPROG2
    && e·title = newplaysINPUTPROG1·head}
(3) fact{no e:playINPUTDB1 |
    e· title =newplaysINPUTPROG1·head}
(4) fact{one e:authorINTERNALDB1 |
    e· name=authornameINPUTPROG2}
(5) one sig errorINTERNALPROG4 in Int {}
(5) fact{errorINTERNALPROG4=0}
```

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{(errorINTERNALPROG4 = 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 Assignment creates a new symbolic variable of the correct type to represent the new value of the assigned variable and generates a constraint to specify that this variable contains the value that can be computed by evaluating the expression on the right of the "=" symbol:

```
one sig newplaysINTERNALPROG5 in List + Nil {}
fact{newplaysINTERNALPROG5=newplaysINPUTPROG1·tail}
```

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{!(!( newplaysINTERNALPROG5 = Nil))}
```

As all the statements in the path have been symbolically executed, the algorithm stops and returns the generated Alloy constraints model. The Alloy analyzer [14] can be asked to find a solution for these constraints using the following commands:

```
assert inputsExist {!(newplaysINPUTPROG1 in List + Nil
    && authornameINPUTPROG2 in Int
    && authorINPUTDB2 in author
    && playINPUTDB1 in play) }
check inputsExist
```

The solution returned by the Alloy analyzer for the symbolic variables representing the input values of the program constitute input data (in this case, two empty initial contents for the author and play tables, one list containing only the integer 7 as input for the LOAD statement and the integer 7 as input for the READ statement) for the program which guarantee that the considered path will be executed:

{authorINPUTDB2={}, playINPUTDB1={}, newplaysINPUTPROG1=[7,NIL], authornameINPUTPROG2=7}

4. EXPERIMENTAL EVALUATION

The algorithm proposed in this work has been prototyped and evaluated experimentally using three SimpleDB test models. For each program, several execution paths have been symbolically executed by our algorithm. For each path, we asked the Alloy analyzer [14] to find several solutions for the constraints generated by the algorithm. The evaluation process was completed by checking that each of these computed solutions was actually input data with respect to which the experimented program was guaranteed to follow the selected path. Both the SimpleDB test models and the tested paths were carefully selected to offer a good coverage of the promised abilities of our algorithm. 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 an equivalent set of boolean constraints, and solves them using a SAT solver. The main statistics measured during the experimentation process for each of the three models are synthesized in table 4. The whole material used and produced during the experimentation process, including the source code of the SimpleDB test models and the Alloy constraints generated by our algorithm, as well as all the performance-related information gathered during tests can be found on the web[1].

The first SimpleDB test model contains eighty-five lines of SimpleDB code that performs repeated manipulations of integers and lists using assignment, If and While statements. First, two lists are loaded and their size are compared. The program also reads as much integers from the outside world as the number of elements in the shortest of the two lists. A third list is created using these integers in the inverted order of the one in which they were read. If the two first lists have the same size, the elements of the three lists are 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 eight paths selected for this first test model were chosen so that only input lists with particular size and content allow the path to be followed during program execution. Between eighteen and one hundred and three symbolic variables and between thirty and one hundred and forty relational constraints were generated by the algorithm for each of the tested paths.

The second SimpleDB test model contains seventy five lines of SimpleDB code that 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

[1]See `http://info.fundp.ac.be/~mmr/issta13`

Table 1: Symbolic variables and relational constraints generation for $\langle db\text{-}write \rangle$ statements

$\langle db\text{-}write \rangle$ ::= `INSERT INTO` $\langle id \rangle$ `VALUES(`$\langle int\text{-}expr \rangle_1$,..., $\langle int\text{-}expr \rangle_i$,...,$\langle int\text{-}expr \rangle_n$`)`	**if** (no exception thrown in path for this INSERT) { **sig** freshAlloyVar **in** $\langle id \rangle$ {} **fact** { **one** e:$\langle id \rangle$ | freshAlloyVar=alloyName+e && e·att$_*$ = alloyOf($\langle int\text{-}expr \rangle_*$) } **fact** { **no** e:alloyName | e· pk =alloyOf($\langle int\text{-}expr \rangle_{\mathsf{pk}^{pos}}$) } *// Primary key is not violated* **fact** { **one** e:fk$_*^{tab}$ | e· fk$_*^{pk}$ = alloyOf($\langle int\text{-}expr \rangle_{\mathsf{fk}_*^{pos}}$) } *// Every foreign key is not violated* } **else** { **if** (Exception thrown in path: inserted primary key value already exists in the table) { **fact** { **one** e:alloyName | e· pk =alloyOf($\langle int\text{-}expr \rangle_{\mathsf{pk}^{pos}}$) } } **else if** (Exception thrown in path: an inserted foreign key value references no row) { **fact** { **no** e:fk$_i^{tab}$ | e· fk$_i^{pk}$ = alloyOf($\langle int\text{-}expr \rangle_{\mathsf{fk}_i^{pos}}$) } } **else if** (Exception thrown in path: an inserted attribute violates an arithmetic constraint) { **fact** {!($\langle int\text{-}expr \rangle_{\mathsf{co}^{pos}}$ co^{right})}} }
$\langle db\text{-}write \rangle$ ::= `UPDATE` $\langle id \rangle$ `SET` $\langle id \rangle_{att}$`=`$\langle int\text{-}expr \rangle$ `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·att$_{*-\{\langle id \rangle_{att}\}}$= e·att$_{*-\{\langle id \rangle_{att}\}}$ && y·$\langle id \rangle_{att}$ = alloyOf($\langle int\text{-}expr \rangle$,$\langle id \rangle$,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·att$_{*-\{\langle id \rangle_{att}\}}$= e·att$_{*-\{\langle id \rangle_{att}\}}$ && y·$\langle id \rangle_{att}$ = alloyOf($\langle int\text{-}expr \rangle$,$\langle id \rangle$,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** { **if** (Exception thrown in path: 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($\langle int\text{-}expr \rangle$,$\langle id \rangle$,a) = alloyOf($\langle int\text{-}expr \rangle$,$\langle id \rangle$, 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($\langle int\text{-}expr \rangle$,$\langle id \rangle$,b)))) } **else if** (Exception thrown in path: 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·$\langle id \rangle_{att}$)) } } **else if** (Exception thrown in path: 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($\langle int\text{-}expr \rangle$,$\langle id \rangle$,a)) } } **else if** (Exception thrown in path: an inserted attribute violates an arithmetic constraint) { **fact** { **some** a:alloyName | alloyOf($\langle db\text{-}cond \rangle$,$\langle id \rangle_{tab}$, a) && !(alloyOf($\langle int\text{-}expr \rangle$,$\langle id \rangle$, a))co$_i^{right}$)}} }
$\langle db\text{-}write \rangle$ ::= `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** { *// 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 2: Translation of SimpleDB expressions and conditions into Alloy

Parameters	**alloyOf(Paramaters)**
$\langle id \rangle$	alloyName
$n \in \mathbb{N}$	n
(⟨*int-expr*⟩$_1$ {+|-|*|/} ⟨*int-expr*⟩$_2$)	(alloyOf(⟨*int-expr*⟩$_1$)).{add | sub | mul | div}[alloyOf(⟨*int-expr*⟩$_2$)]
(- ⟨*int-expr*⟩)	(- (alloyOf(⟨*int-expr*⟩)))
⟨*id*⟩.HEAD	alloyName.head
⟨*id*⟩$_{tab}$(⟨*id*⟩$_{att}$)	{a:alloyName$_{tab}$ | all b:alloyName$_{tab}$ | a.pk$_{tab}$<=b.pk$_{tab}$}.⟨*id*⟩$_{att}$
NIL	Nil
⟨*id*⟩.TAIL	alloyName.tail
TRUE	(0=0)
FALSE	(0=1)
(⟨*cond*⟩$_1$ {&&||} ⟨*cond*⟩$_2$)	(alloyOf(⟨*cond*⟩$_1$) {&&||} alloyOf(⟨*cond*⟩$_2$))
(! ⟨*cond*⟩)	(!(alloyOf(⟨*cond*⟩)))
(⟨*int-expr*⟩$_1$ {<|=|>} ⟨*int-expr*⟩$_2$)	((alloyOf(⟨*int-expr*⟩$_1$)) {<|=|>} (alloyOf(⟨*int-expr*⟩$_2$)))
(⟨*id*⟩=NIL)	(alloyName=Nil)
alloyOf(x,y,z) behaves in a similar way to alloyOf(x) except in the two following cases	
⟨*id*⟩, table, row	**if** (table contains ⟨*id*⟩) **then** row.⟨*id*⟩ **else** alloyName
(⟨*id*⟩{<|=|>} ⟨*int-expr*⟩), table, row	(row.⟨*id*⟩ {<|=|>} (alloyOf(⟨*int-expr*⟩,table,row)))

Table 3: Abbreviations list and details

Abbreviation	**Meaning**
freshAlloyVar	A new Alloy variable name that has still not been used in the Alloy code generated so far.
alloyName$^{superscript}_{subscript}$	**if** (⟨*id*⟩$^{superscript}_{subscript}$ refers to a database table name) **then** The symbolic variable that represents the current content of table ⟨*id*⟩$^{superscript}_{subscript}$) **else** The symbolic variable that represents the current content of the SimpleDB variable ⟨*id*⟩$^{superscript}_{subscript}$
att$_i$	Name of the i$_{th}$ attribute in the list of attributes of table ⟨*id*⟩
pk$^{superscript}_{subscript}$	Name of the primary key attribute of table ⟨*id*⟩$^{superscript}_{subscript}$.
pk^{pos}	Position of primary key in the list of attributes of table ⟨*id*⟩
fk$^{tab}_i$	Name of the table referenced by the i$_{th}$ foreign key in the list of foreign keys of table ⟨*id*⟩
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 ⟨*id*⟩
fk$^{pos}_i$	Position of the foreign key attribute, declared by the i$_{th}$ foreign key in the list of table ⟨*id*⟩, in the list of attributes of table ⟨*id*⟩
fk$_i$	Name of the foreign key attribute declared by the i$_{th}$ foreign key in the list of table ⟨*id*⟩
ifk$^{tab}_i$	Name of the table where is declared the i$_{th}$ foreign key referencing table ⟨*id*⟩ in the whole schema
ifk$^{att}_i$	Name of the foreign key attribute declared by the i$_{th}$ foreign key referencing table ⟨*id*⟩ in the schema
co$^{pos}_i$	Position of the attribute constrained by the i$_{th}$ arithmetic constraint declared in table ⟨*id*⟩
co$^{right}_i$	Right part of the i$_{th}$ arithmetic constraint declared in table ⟨*id*⟩ (i.e. right part of "a>0" is ">0")
xx$_*$ means "for each xx$_i$" and xx$_{*-\{y\}}$ means "for each xx$_i$ except from y"	

Table 4: Experimental evaluation statistics

Model	**Code lines**	**Tested paths**	**Symbolic variables**		**Relational constraints**		**Constraints solving time**	
			Min	**Max**	**Min**	**Max**	**Min**	**Max**
1	85	8	18	103	30	140	132 ms	4,620 ms
2	65	8	4	47	17	131	309 ms	262,320 ms
3	71	3	21	47	35	76	118 ms	2,215 ms

deleted. The eight paths experimented over this program were carefully selected to test the generation of correct constraints for most possible behaviors for SQL statements over different kinds of tables structures and constraints. Between four and forty seven symbolic variables and between seventeen and one hundred and thirty one relational constraints were generated by the algorithm for each of the tested paths.

The third SimpleDB test model contains seventy one lines of SimpleDB code that mixes SQL statements with imperative 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 rollbacked to state it was when the transaction was launched. The selected paths for this third test model focus on the transaction management and contain a path in which both transactions are commited, one in which the first transaction is committed but the second is rolled back due to a foreign-key constraint violation, and a third one in which both transactions are partially executed but rolled back, also due to a foreign-key constraint violation in each of the transactions. Between twenty one and forty seven symbolic variables and between thirty five and seventy six relational constraints were generated for each of the tested paths.

For each of the tested paths among the three examples, every solution of the constraints generated by our algorithm provided a correct set of program inputs leading to the execution of the path. Concerning the infeasible path of the second test model, the Alloy analyzer was able to detect that the generated relational constraints were not satisfiable and did not proceed with SAT solving. Performance of the constraint solving process was acceptable in most cases, except for the longest paths involving many SQL statements, where it took up to four minutes to solve the constraints on a recent dual core x86 processor with 8GB of memory.

5. CONCLUSION AND RELATED WORK

In this work, we have proposed and detailed a complete algorithm to execute symbolically database programs. Given a database program and an execution path in this program, the algorithm uses static analysis to generate a symbolic variable for each potential value taken by a program variable or database table before and during the path execution. It generates as well an Alloy relational constraints model constraining these symbolic variables to guarantee the execution of the considered path. Any solution to these produced constraints describes input data for the program, including an initial content for the database, 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, satisfying a given code coverage criterion, the proposed algorithm can be used to generate inputs for each path in the set. These inputs can then be used in turn as adequate test data for structural white-box testing of the program.

An early approach that has considered test data generation for imperative programs interacting with a relational SQL database is [3]. The paper proposes to transform the program, thereby inserting new variables representing the database structure, and translating all SQL statements and integrity checks into imperative program code. Classical white-box testing approaches can then be applied to the modified program. In [10], the authors propose an algorithm for testing an imperative program performing SELECT queries on a relational SQL database, based on a simultaneous concrete and symbolic (concolic) dynamic execution of some of its execution paths. Concolic execution runs the program on random input data and on a randomly populated input database. Given the 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. These constraints must be combined with the constraints derived from the database schema. In [34], authors adopt a similar concolic approach where the program is executed on a parameterized mock database. In [21] and [30], 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 [29], the same concolic approach is applied considering advanced code coverage criteria. Compared to all of these approaches, our approach does not need to transform the original program, offers a clean modeling of the problem as a single relational constraints system generation problem, and allows to account for Insert, Update, and Delete statements, as well as transactions management primitives, that are commonly used in database applications. On the other hand, our approach only considers static SQL where the concolic ones allow to account for dynamic SQL. Finally, translation between database schemas/programs and Alloy models has already been considered in other contexts [5, 15].

In future work, we intend to make our technique able to generate inputs for more complex interaction scenarios between databases and programs. First, it would be relevant to evaluate how and up to which extent the symbolic execution mechanism proposed here for simple SQL statements and simple relational database schemas can be generalized to more elaborate ones. Secondly, it should be investigated how dynamic SQL can be integrated with our approach, possibly relying on static analysis [37, 35] or on concolic execution. Thirdly, it happens frequently that SQL statements have a non-deterministic behavior, either because the underlying DBMS executing the statement behaves non-deterministically, or because the database is modified concurrently by several programs. Whether and how the approach proposed here can encompass such non-deterministic behaviors remains a topic for further research.

Finally, 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 [13, 18, 32, 33, 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.

This work is unpublished work preliminary to [25, 23, 24].

6. ACKNOWLEDGMENTS

This work has been funded by the Belgian Fund for Sci-

entific Research (F.R.S.-FNRS). The authors would like to thank Vincent Englebert for useful discussions.

7. REFERENCES

[1] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *in the IDEA System. Int. Symp. on Advanced Database Technologies and Their Integration*, 1994.

[2] 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.

[3] 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*, pages 363–374. Springer, 1999.

[4] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, May 1976.

[5] 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*, SEFM '09, pages 285–294, Washington, DC, USA, 2009. IEEE Computer Society.

[6] C. Date. *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 8 edition, 2003.

[7] C. de la Riva, M. J. Suárez-Cabal, and J. Tuya. Constraint-based test database generation for sql queries. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 67–74, New York, NY, USA, 2010. ACM.

[8] 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*, LOPSTR'09, pages 128–142, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, Sept. 1991.

[10] 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*, ISSTA '07, pages 151–162, New York, NY, USA, 2007. ACM.

[11] I. O. for Standardization. Information technology—database languages-sql-part 1: Framework (sql/framework), iso/iec 9075-1: 1999 and information technology—database languages-sql-part 2: Foundation.

[12] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. *SIGSOFT Softw. Eng. Notes*, 23(2):53–62, Mar. 1998.

[13] 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*, ASE '06, pages 69–80, Washington, DC, USA, 2006. IEEE Computer Society.

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

[15] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. *SIGSOFT Softw. Eng. Notes*, 25(5):14–25, Aug. 2000.

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

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

[18] 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*, ESEC/FSE-11, pages 98–107, New York, NY, USA, 2003. ACM.

[19] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and 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, pages 238–247, Washington, DC, USA, 2008. IEEE Computer Society.

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

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

[22] 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*, DBTest '12, pages 6:1–6:6, New York, NY, USA, 2012. ACM.

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

[24] M. Marcozzi, W. Vanhoof, and J.-L. Hainaut. Testing database programs using relational symbolic execution. Technical report, University of Namur, 2014.

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

[26] C. Meudec. ATGen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing Verification and Reliability*, 11(2):81–96, 2001.

[27] A. Neufeld, G. Moerkotte, and P. C. Lockemann. Generating consistent test data for a variable set of

general consistency constraints. *VLDB J.*, 2(2):173–213, 1993.

[28] A. J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction procedure for test data generation. *Softw. Pract. Exper.*, 29(2):167–193, Feb. 1999.

[29] K. Pan, X. Wu, and T. Xie. Database state generation via dynamic symbolic execution for coverage criteria. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, DBTest '11, pages 4:1–4:6, New York, NY, USA, 2011. ACM.

[30] 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.

[31] 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*, AST '06, pages 85–91, New York, NY, USA, 2006. ACM.

[32] 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*, SIGSOFT '04/FSE-12, pages 253–262, New York, NY, USA, 2004. ACM.

[33] M. J. Suárez-Cabal and J. Tuya. Structural coverage criteria for testing SQL queries. *Journal of Universal Computer Science*, 15(3):584–619, 2009.

[34] 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.

[35] S. Thomas, L. Williams, and T. Xie. On automated prepared statement generation to remove sql injection vulnerabilities. *Inf. Softw. Technol.*, 51(3):589–598, Mar. 2009.

[36] M. Veanes, P. Grigorenko, P. Halleux, and N. Tillmann. Symbolic query exploration. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '09, pages 49–68, Berlin, Heidelberg, 2009. Springer-Verlag.

[37] G. Wassermann, C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. *ACM Trans. Softw. Eng. Methodol.*, 16(4), Sept. 2007.

[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*, ICST '09, pages 396–405, Washington, DC, USA, 2009. IEEE Computer Society.

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