A Direct Symbolic Execution of SQL into Relational Constraints for Unit Testing of Data-Oriented Applications

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

Publication date:
2015

Document Version
Early version, also known as pre-print

A Direct Symbolic Execution of SQL into Relational Constraints for Unit Testing of Data-Oriented Applications

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

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

Abstract

Symbolic execution is a technique enabling the automatic generation of test inputs that exercise a set of execution paths within a code unit to be tested. If the paths cover a sufficient part of the code under test, the test data offer a representative view of the actual behaviour of this code. This notably enables detecting errors and correcting faults. Relational databases are ubiquitous in software, but symbolic execution of code units that manipulate them remains a non-trivial problem, particularly because of the complex structure of such databases and the complex behaviour of SQL statements. Finding errors in such code units is yet critical, as it can avoid corrupting important data. In this work, we define a symbolic execution translating database manipulation code directly into constraints and integrate it with a more traditional symbolic execution of normal program code. The database tables are represented by relational symbols and the SQL statements by relational constraints over these symbols. An algorithm based on these principles is presented for the symbolic execution of simple Java methods that implement transactional use cases by reading and writing in a relational database, the latter subject to data integrity constraints. The algorithm is integrated in a test generation tool and experimented over sample code. The target language for the constraints produced by the tool is the SMT-Lib standard and the used solver is Microsoft Z3. The results show that the proposed approach enables generating meaningful test data, including valid database content, in reasonable time. In particular, the Z3 solver is shown to be more scalable than the Alloy solver, used in our previous work, for solving relational constraints.

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

---

☆

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

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

1. Introduction

In current software development practice, testing [1, 2] remains the primary approach to detect errors in software. Testing being a complex and costly activity has motivated much research on efficient techniques to automate all aspects of the software testing process [3]. Of particular interest is the automation of *test data generation* [4] for functional unit testing, where the idea is to automatically generate a representative set of inputs for a unitary program fragment under test (typically a function or method). Running the considered code unit with the generated test data then offers a rather representative view of its actual behaviour, enabling one to detect errors efficiently. Moreover, once a suitable set of test data has been generated (and verified), it can be used as reference data for continued (regression) testing.

While different approaches exist towards the automatic generation of test data, *symbolic execution* [5] has been recognised as a state-of-art technique for so-called *white-box structural* test data generation [6, 7, 8, 9, 10]. In such an approach, the idea is to generate test data that in some way *cover* a sufficiently large part of the control-flow graph of the code unit under test [4]. In a nutshell, symbolic execution executes the unit under test over *symbolic input values* instead of concrete ones [5]. Each time a control dependency is encountered, it proceeds along one of the possible paths, thereby generating *constraints* upon the symbolic input values. These constraints are such that if the code inputs had concrete values satisfying them, the real execution would proceed along the selected path. Once a termination point is reached, the constraints collected along are regrouped in a so-called *path-constraint*. This path-constraint is then solved, producing concrete test inputs, making the real execution follow the whole path that has just been symbolically executed. This test generation process is repeated until test data have been generated for a sufficiently large and diverse number of paths through the control-flow graph, according to some coverage criterion [4]. Test data generation based on symbolic execution is now at the core of various popular open-source and commercial testing tools, some being used in an industrial context, notably at Microsoft and NASA [9].

Symbolic execution of imperative code has been widely studied [11, 12, 13, 14], as well as generalised to lower-level or higher-level programming paradigms, such as for x86 assembly [15] or Java [16, 17] and C# [18] object-oriented code. In so-called database programs, such classical code is mixed with SQL code to interact with a relational database. Enabling symbolic execution in the presence of SQL statements among the classical statements is a non-trivial extension of existing symbolic execution techniques. Such an extension could be beneficial, as enabling an efficient testing of the programs that manipulate databases would provide a powerful means to prevent data corruption.

From a theoretical standpoint, deciding the satisfiability of an SQL query is not computationally possible in general [19], so that generating test data for any particular execution path, in the presence of SQL statements in the path, is not generally computable either [20].

From a more practical standpoint, the database can be seen as nothing else than a particular kind of container for some of the input values manipulated by the code unit under test. During symbolic execution, these values must thus be represented symbolically and subsequently constrained to enable the proper generation of test data, including a valid input database content. The symbolic representation of these values raises difficulties as the database is a container of particularly complex shape: its content must obey the so-called *database schema*, defined using SQL DDL code [21]. This database schema defines a set of tables where the database content will be stored. Each table can be seen as a container for a mathematical relation, i.e., a set of tuples with no limit on the potential number of tuples. The schema typically also describes a set of *data integrity constraints* that must be enforced by the relations in the tables, like the primary key, foreign key or check constraints. These constraints are particularly complex as they are quantified first-order logic constraints. For instance, a primary key constraint states that for all couples of tuples in the relation represented by a table, the value of the primary key fields cannot be all equal.

Moreover, a code unit typically interacts with the database by using SQL [21] *query statements* and *modification statements* that are embedded in the unit's source code. As such, SELECT queries enable gathering values from the database tables in order to copy them into the code unit's variables. Modification statements INSERT, UPDATE and DELETE enable modifying the content of the database tables, typically a function of the value of the code unit's variables. Deriving adequate path-constraints in presence of such SQL statements raises practical difficulties as well. Firstly, SQL is a declarative language: SQL statements express the desired action over the content of the database relations, but they do not make explicit the (often complex) sequence of operations necessary to compute this action. In practice, during execution, these SQL statements are sent by the code unit – using a dedicated API – to the DataBase Management System (DBMS) [21], an external component responsible for the interpretation and execution of SQL code over the database. The DBMS keeps an optimised and persistent internal representation of the database and manages concurrent distant accesses by enabling the database programs to use *transactions* [21]. Secondly, the execution of INSERT, UPDATE or DELETE statements by the DBMS does not only consist in modifying the content of the database, but also in checking that these modifications do not leave the database in a state where the integrity constraints defined in the schema are violated. If an integrity constraint violation is detected, the modification statement's execution fails and the database remains unmodified. As a consequence, during symbolic execution each INSERT, UPDATE or DELETE statement will have to be treated as an if-then-else statement with a particularly complex condition:

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

Two research approaches have been developed so far to handle the presence of SQL statements within the classical code to be symbolically executed. The first approach [22, 23] introduces new native variables in the classical code to represent the database content and replaces the SQL queries and modification statements by native code acting on these new variables. Classical symbolic execution can then be applied on this *normalised* version of the tested code. The second approach [24] considers the result of any executed query as an *independant code input*, typed as a relation. The size and cell content of these input relations can be subsequently constrained within the path-constraint, composed of string and integer constraints. Solving this path-constraint enables generating rows to insert into the database, so that the queries executed by the tested code should return results driving the execution into the chosen path.

In this work, we propose a different approach, called *relational symbolic execution*, which enables a direct translation of both SQL queries and modification statements into relational constraints. Contrary to the normalisation approach, relational symbolic execution makes it possible to test the original code directly, without transforming it. Contrary to the input arrays approach, relational symbolic execution makes it possible to test code which writes data into the database and which involves queries whose results are interdependent.

In practice, relational symbolic execution models every table in the database as a variable from the code unit, typed as a mathematical relation. Each SQL statement in the code unit can then be modelled as a relational operation over these *relational variables* as well as the traditional code unit's variables. By defining a symbolic execution over this relational version of the code, we can derive path-constraints over the values of both the code unit's relational and traditional inputs. More precisely, the generated-path constraints will include symbols representing the classical input values of the code unit, as well as symbols representing the input relations stored in the database tables when the unit execution starts. Furthermore, each path-constraint is then combined with additional constraints, which guarantee that the relational variables in the code always fulfil the integrity constraints defined in the database schema. The result is a complex constraint system that mixes traditional constraints over the code inputs with relational constraints over the initial content of the database. Each solution, instantiated to the combined constraint system, describes test data, *including* a valid initial state for each table in the database, such that when the code is executed with respect to these data values (including the database), the execution will follow the path represented by the constraint system.

The main contribution of this work is a relational symbolic execution algorithm, which implements the technique described in the previous paragraph for a precise subset of Java, empowered by the JDBC API for using SQL primitives and transactions. The algorithm has been designed in the context of testing transactional software. In such a context, database programs can typically be divided into a set of rather small and simple methods, implementing each a precise business use case, like registering a book borrowing or saving a sale transaction. These use cases involve a short sequence of basic SQL interactions, touching a limited number of entities in the database. Given the Java/SQL code of a single Java method implementing such a use case, the SQL DDL code describing the part of the database schema touched by this method and a finite path in the control flow graph of the method, the algorithm generates the corresponding constraints.

A major challenge for relational symbolic execution is solving the mix of relational and classical constraints produced by the approach. In our previous work [25, 26], these constraints were expressed using the Alloy language [27] and solved using the Alloy analyser [27]. Recently, it has been proposed [28] to translate Alloy constraints into the SMT-Lib standard language [29], as some solvers based on this language, like Microsoft Z3 [30], enable detecting the unsatisfiability of the constraints, which is not possible within the Alloy framework. However, [28] advises to continue using the Alloy analyser for finding solutions to satisfiable constraints, as Z3 had exhibited some limitations for this task, due to the unavoidable presence of quantifiers when translating relational constraints into SMT-Lib. Nevertheless, more recent versions of Z3 take advantage of new model-based quantifier instantiation capabilities (MBQI) [31], providing the solver with a promising ability to find efficiently solutions for satisfiable quantified constraints. In this paper, the relational symbolic execution algorithm from our previous work has been redesigned to generate constraints in the SMT-Lib language and solved using an MBQI-powered version of Z3. Z3 is at the core of several existing state-of-art symbolic execution tools (e.g. [18, 15]). Moreover, it handles a much larger and more various panel of constraints than Alloy, the latter being limited to bounded-scope integer and relational constraints. Using Z3 as a back-end solver does strengthen thus the generalisability of our approach.

A test generation tool based on our new relational symbolic execution algorithm and the Z3 solver has been coded and used to generate test data for a number of sample Java methods and databases, including open-source code extracted from the web. These experiments enabled a comparative evaluation of Alloy and Z3 for solving the generated constraints. They also made it possible to compare, to some extent, the performance of our approach with that of related work and to evaluate its scalability limits.

The remainder of this paper is organised as follows. The relational symbolic execution algorithm is presented in section 2 and 3. Section 2 discusses the transactional context for which the algorithm was designed and formally defines the part of the Java/SQL syntax that is supported. Section 3 systematically describes the constraint generation rules used by the algorithm for the symbolic execution of the supported language. The test generation tool based on the algorithm is described, together with our experimental evaluation of the approach, in section 4. Finally, a synthesis of the research contribution (section 5), as well as a discussion of related work (section 6), threats to validity (section 7) and future work (section 8) are provided.

2. A study framework for unit testing of transactional database programs

2.1. Database programs in a transactional context

Interaction between software and a relational database is often said to occur either in an OLTP (On-Line Transaction Processing) or in an OLAP (On-Line Analytical Processing) context. In a nutshell, OLTP corresponds to the case where few simple operations over some precise data are performed live to support one individual human activity, like saving a bank transaction or processing a ticket registration. The OLAP context occurs when large amounts of data are mined from a database to support reporting or processing activities, like accounting, budget or marketing.

Our approach aims at automating unit testing in the context of OLTP. Database programs written in an OLTP context can typically be divided into rather small [32] code units "implementing [business] use cases which execute a sequence of actions whereas each action usually reads or updates only a small set of tuples in the database" [33]. Such a kind of a use case is aimed to support one particular human activity. An example extracted from [33] of such an activity is a library user that wants to borrow a book. The use case, implemented into a code unit, checks the user and book data and possibly allows and saves the borrowing.

The purpose of our work is to generate test data for such OLTP code units. In practice, the targeted code units have simple operations and do not involve complex computations. They perform at most a few dozen simple SQL operations acting over a group of at most a dozen of interrelated tables.

2.2. A study language for code units in transactional database programs

2.2.1. General presentation

OLTP code units can be written in many different programming languages, using different DBMS interfaces, and taking advantage of the numerous features of the various SQL dialects. Given this diversity, it is necessary to define a framework to properly study our relational symbolic execution approach. Practically, we have chosen to consider a formal Java subset with JDBC [34] and ISO SQL [35] primitives, which are all very popular technologies.

Our language enables to write code implementing an OLTP business use case. Such an implementation is composed of the DDL schema of the manipulated part of the database and of the code of the Java method carrying out the use case. The database schema describes a set of tables composed of integer attributes. Each table has a primary key and the attributes can be constrained by foreign key constraints and check constraints. The code of the method can contain if-then-else blocks, while loops and return statements. It can contain integer and list (local) variable assignments, with typical operators for lists and linear operators for integers. Assertions can be declared in the code. The method interacts with the database through SQL base statements whose structure is entirely described statically, and through SQL base primitives for transaction management. Statements writing in the database can throw runtime exceptions if the write operation violates the DDL schema. Such exceptions can be caught. The method

Figure 1: SQL and Java/JDBC code implementing an use case adding books in a library database.

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

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

```java
List<Integer> addBooks (Connection con, Scanner in, List<Integer> newBooks) throws SQLException {
 int i = 0;  List<Integer> addedBooks = new ArrayList<Integer>();
 while ( !(newBooks == null) & (i < newBooks.size()) ) {
   int error = 0;  int theShelf = in.nextInt();
   ResultSet shelves = con.createStatement().executeQuery("SELECT id FROM shelf WHERE id="+theShelf);
   if ( ! shelves.next() )
    con.createStatement().execute("INSERT INTO shelf VALUES ("+theShelf+",1)");
   else
    con.createStatement().execute("UPDATE shelf SET numberOfBooks=numberOfBooks+1 WHERE id ="+theShelf);
   try {
    con.createStatement().execute("INSERT INTO book VALUES ("+newBooks.get(i)+","+theShelf+")");
   } catch (SQLException e) {
    error = 1;
   };
   if (error==0) {
    con.commit();
    addedBooks.add(newBooks.get(i));
   } else
    con.rollback();
   i = i + 1;
 };
 return addedBooks; }
```

receives as input parameters a JDBC connection to the database, a set of integer lists and an input scanner for integers. The lists model any structured group of inputs transmitted to the code at method call. The scanner models the method's access to simple data from the "outside world", like user prompt, network access, etc.

Figure 1 provides a sample use case implemented in the language considered here. This code describes a database with two tables: one for library shelves and one for the books stored in each of these shelves. The total number of books stored in a shelf is saved for each shelf. The Java method adds a set of new books to the database and updates the shelves' books counts. If a book is added to a non-existent shelf, then the shelf is itself added to the database as well. The books are inserted one by one in isolated transactions. If a transaction was successful, the code of the added book is saved in a list, which is returned at the end of the method's execution.

As a matter of fact, our study language contains all base SQL primitives as well as a Turing-complete subset of Java. The symbolic execution of many Java constructs not considered here has been studied elsewhere (see e.g. [16, 17]) and is a problem orthogonal to this work, which studies the *integration* of SQL handling into classic symbolic execution. Our study language allows integers as the only datatype for the values stored in the database. Non-linear integer arithmetic operations are not permitted by the language. The sole consequence of allowing other datatypes, like bit-vectors, reals, timestamps or strings, as well as of allowing non-linear arithmetics, would be that our algorithm would produce bit-vector, real, timestamp, string and non-linear arithmetics constraints, in addition to the linear integer constraints produced here. As an impact, the underlying constraint solver may become unable to properly handle the produced constraints. Solving complex constraints in an efficient way is a well-known challenge for symbolic execution in general [10]. This issue is evaluated for our particular context in the discussion part of the paper.

In the remaining part of this subsection, we define precisely – using a BNF grammar – what subset of the Java/SQL syntax our algorithm can execute symbolically. This description is made with the implicit requirement that the written code is well-typed according the relevant standard Java and SQL typing rules.

The chosen notation for the BNF is standard but includes some additional meta-symbols: {...} (grouping), $^?$ (zero or one times), * (zero or more times) and $^+$ (one or more times). When a single nonterminal appears several times in a single production, subscript notation enables distinguishing between the occurrences.

2.2.2. OLTP use case implementation

An OLTP use case implementation is composed of the SQL DDL code of the touched part of the database (the tables containing the data actually manipulated in the use case and all the tables directly or indirectly linked to these tables by foreign key constraints, the other tables can be left empty or filled randomly during the testing process) and of the code of the Java method carrying out the use case.

$\langle$*oltp-use-case-implementation*$\rangle$ ::= $\langle$*sql-ddl*$\rangle$ $\langle$*Java-method*$\rangle$

2.2.3. Database schema

The relational database schema is a list of table definitions. This list can be empty, in which case the method is a classical method that works independently of any database.

In the list, each table is identified by its name, contains at least one attribute and endorses exactly one primary key. Foreign keys and additional check constraints can be declared for a table. A row in a table cannot be deleted or see its primary key value modified as long as there exists at least another row in the database that references it (ON DELETE/UPDATE NO ACTION). The semantics of all the schema creation primitives conforms to the standard [35] SQL DDL.

$\langle \textit{sql-ddl} \rangle ::= \langle \textit{table} \rangle^*$

$\langle \textit{table} \rangle ::= \texttt{CREATE TABLE}\ \langle \textit{id} \rangle\ \texttt{(}\langle \textit{att} \rangle^+\ \langle \textit{p-key} \rangle\ \langle \textit{f-key} \rangle^*\ \langle \textit{chk} \rangle^*\texttt{);}$

$\langle \textit{att} \rangle ::= \langle \textit{id} \rangle\ \texttt{INTEGER NOT NULL ,}$

$\langle \textit{p-key} \rangle ::= \texttt{CONSTRAINT}\ \langle \textit{id} \rangle_{cst}\ \texttt{PRIMARY KEY (}\ \langle \textit{id} \rangle_{att}\ \texttt{)}$

$\langle \textit{f-key} \rangle ::= \texttt{,CONSTRAINT}\ \langle \textit{id} \rangle_{cst}\ \texttt{FOREIGN KEY (}\ \langle \textit{id} \rangle_{att}\ \texttt{) REFERENCES}\ \langle \textit{id} \rangle_{tab}\ \texttt{(}\ \langle \textit{id} \rangle_{refid}\ \texttt{)}$

$\langle \textit{chk} \rangle ::= \texttt{,CHECK (}\langle \textit{id} \rangle\ \{\texttt{<} \mid \texttt{=} \mid \texttt{>}\}\ \langle \textit{integer} \rangle\texttt{)}$

$\langle \textit{id} \rangle ::= \{\text{a} \mid ... \mid \text{z} \mid \text{A} \mid ... \mid \text{Z}\}\{\text{a} \mid ... \mid \text{z} \mid \text{A} \mid ... \mid \text{Z} \mid 0 \mid ... \mid 9\}^*$

$\langle \textit{integer} \rangle ::= \text{-}^?\ \{1 \mid ... \mid 9\}\{0 \mid ... \mid 9\}^* \mid 0\}$

2.2.4. Method signature and body

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

$\langle \textit{Java-method} \rangle ::= \langle \textit{type} \rangle\ \langle \textit{id} \rangle\ \texttt{(}\langle \textit{db-con} \rangle\texttt{,}\langle \textit{inp} \rangle\ \langle \textit{parameters} \rangle\texttt{) throws SQLException}$
$\qquad \texttt{\{}\ \langle \textit{stmt} \rangle^*\ \texttt{\}}$

$\langle \textit{type} \rangle ::= \texttt{void}$
$\quad \mid \texttt{int}$
$\quad \mid \texttt{List<Integer>}$

$\langle \textit{db-con} \rangle ::= \texttt{Connection con}$

$\langle \textit{inp} \rangle ::= \texttt{Scanner in}$

$\langle \textit{parameters} \rangle ::= \{\texttt{, List<Integer>}\ \langle \textit{id} \rangle\ \}^*$

The connection with the database is assumed reliable and every SQL statement, being well formed, is processed for its effect on the database. The semantics of all the Java constructs conforms to the classical Java specification and documentation. The semantics of all SQL statements conforms to the standard [35] SQL specification.

2.2.5. Common statements and list management

Common condition, loop, assertion and assignment statements, as well as common integer expressions and boolean conditions can be used. Lists can be manipulated with 'add(int)', 'remove(int)', 'get(int)' and 'size()' methods. The 'Java.util.ArrayList' implementation of these methods is assumed to be used. A list variable can be 'null'.

$\langle \mathit{stmt} \rangle$ ::= $\texttt{if (}\langle \mathit{cond} \rangle\texttt{) \{}\langle \mathit{stmt} \rangle_{\mathit{then}}{*}\texttt{\}}\ \{\texttt{else \{}\langle \mathit{stmt} \rangle_{\mathit{else}}{*}\texttt{\}}\}^?\texttt{;}$
| $\texttt{while (}\langle \mathit{cond} \rangle\texttt{) \{ }\langle \mathit{stmt} \rangle{*}\texttt{ \};}$
| $\texttt{assert }\langle \mathit{cond} \rangle\texttt{ ;}$
| $\{\texttt{int} \mid \texttt{List<Integer>}\}^?\ \langle \mathit{id} \rangle\texttt{ = }\langle \mathit{expr} \rangle\texttt{;}$
| $\langle \mathit{id} \rangle\texttt{.add(}\langle \mathit{int\text{-}expr} \rangle\texttt{);}$
| $\langle \mathit{id} \rangle\texttt{.remove(}\langle \mathit{int\text{-}expr} \rangle\texttt{);}$
| $\texttt{return }\langle \mathit{id} \rangle\texttt{;}$

$\langle \mathit{cond} \rangle$ ::= $\texttt{true}$
| $\texttt{false}$
| $\texttt{(! }\langle \mathit{cond} \rangle\texttt{)}$
| $\texttt{(}\langle \mathit{cond} \rangle_1\ \{\texttt{\&} \mid \texttt{|}\}\ \langle \mathit{cond} \rangle_2\texttt{)}$
| $\texttt{(}\langle \mathit{int\text{-}expr} \rangle_1\ \{\texttt{<} \mid \texttt{==} \mid \texttt{>}\}\ \langle \mathit{int\text{-}expr} \rangle_2\texttt{)}$
| $\texttt{(}\langle \mathit{id} \rangle\texttt{ == null)}$

$\langle \mathit{expr} \rangle$ ::= $\langle \mathit{int\text{-}expr} \rangle \mid \langle \mathit{list\text{-}expr} \rangle$

$\langle \mathit{int\text{-}expr} \rangle$ ::= $\langle \mathit{id} \rangle$
| $\langle \mathit{integer} \rangle$
| $\texttt{(}\langle \mathit{int\text{-}expr} \rangle_1\ \{\texttt{+} \mid \texttt{-}\}\ \langle \mathit{int\text{-}expr} \rangle_2\texttt{)}$
| $\texttt{(}\langle \mathit{id} \rangle\texttt{.get(}\langle \mathit{int\text{-}expr} \rangle\texttt{).intValue())}$
| $\texttt{(}\langle \mathit{id} \rangle\texttt{.size())}$

$\langle \mathit{list\text{-}expr} \rangle$ ::= $\langle \mathit{id} \rangle$
| $\texttt{null}$
| $\texttt{new ArrayList<Integer>()}$

2.2.6. Interacting with the outside world

The scanner parameter of the method can be used to get integer data from the "outside world" (user prompt, network access, reading from a file, etc.). This interaction is assumed to always succeed, without any technical problem. We have thus the following new alternative for the $\langle \mathit{stmt} \rangle$ non-terminal:

$\langle \mathit{stmt} \rangle$ ::= $\{\texttt{int}\}^?\ \langle \mathit{id} \rangle\texttt{ = in.nextInt();}$

2.2.7. 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. We have thus the following alternatives for existing non-terminals:

$\langle \mathit{stmt} \rangle$ ::= $\{\texttt{ ResultSet }\}^?\ \langle \mathit{id} \rangle\texttt{ = con.createStatement().executeQuery(" }\langle \mathit{select\text{-}query} \rangle\texttt{ ");}$
| $\langle \mathit{id} \rangle\texttt{.next();}$

$\langle \mathit{int\text{-}expr} \rangle$ ::= $\langle \mathit{id} \rangle_{\mathit{tab}}\texttt{.getInt(" }\langle \mathit{id} \rangle_{\mathit{att}}\texttt{ ")}$

$\langle \mathit{cond} \rangle$ ::= $\texttt{(}\langle \mathit{id} \rangle\texttt{.next(}\langle \mathit{int\text{-}expr} \rangle\texttt{))}$

as well as the following new non-terminals:

$\langle \mathit{select\text{-}query} \rangle$ ::= $\texttt{SELECT }\{\langle \mathit{id} \rangle_i\texttt{,}\}{*}\langle \mathit{id} \rangle_n\texttt{ FROM }\langle \mathit{id} \rangle_{\mathit{tab}}\ \{\texttt{ WHERE }\langle \mathit{db\text{-}cond} \rangle\ \}^?$

$\langle \mathit{db\text{-}cond} \rangle$ ::= $\texttt{(}\langle \mathit{db\text{-}cond} \rangle_1\ \{\texttt{AND} \mid \texttt{OR}\}\ \langle \mathit{db\text{-}cond} \rangle_2\texttt{)}$
| $\texttt{(NOT }\langle \mathit{db\text{-}cond} \rangle\texttt{)}$
| $\texttt{(}\langle \mathit{id} \rangle\ \{\texttt{<} \mid \texttt{=} \mid \texttt{>}\}\ \langle \mathit{db\text{-}int\text{-}expr} \rangle\texttt{)}$

$\langle$*db-int-expr*$\rangle$ ::= $\langle$*id*$\rangle$
  | $\langle$*integer*$\rangle$
  | `(`$\langle$*db-int-expr*$\rangle_1$ {`+` | `-`} $\langle$*db-int-expr*$\rangle_2$`)`
  | `"+(` $\langle$*int-expr*$\rangle$ `)+"`

Integer expressions appearing in SQL code ($\langle$*db-int-expr*$\rangle$) can be parameterised by integer constants computed dynamically by the Java method. For example, in:

```
void example (Connection con, Scanner in) throws SQLException {
 int n = 0;
 ResultSet r= con.createStatement().executeQuery("SELECT id FROM table WHERE id="+(n)+"");
 r.next();
 int p = r.getInt("id");
}
```

The character string that will be effectively sent to and processed by the DBMS is:

```
SELECT id FROM table WHERE id=0
```

The parametric constants can depend on the method's inputs, like in:

```
void example (Connection con, Scanner in) throws SQLException {
 int n = in.nextInt();
 ResultSet r= con.createStatement().executeQuery("SELECT id FROM table WHERE id="+(n)+""); }
```

2.2.8. Writing data into the database

Data can be written into the database using simple SQL INSERT, UPDATE or DELETE statements. If the execution of 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 method and its execution is stopped. Such exceptions should be caught using a try/catch structure.

$\langle$*stmt*$\rangle$ ::= `con.createStatement().execute("` $\langle$*db-write*$\rangle$ `");`
  | `try { con.createStatement().execute("` $\langle$*db-write*$\rangle$ `"); }`
    `catch (SQLException e)`
    `{` $\langle$*stmt*$\rangle$* `};`

$\langle$*db-write*$\rangle$ ::= `INSERT INTO` $\langle$*id*$\rangle$ `VALUES (` { $\langle$*int-expr*$\rangle_i$ `,` }* $\langle$*int-expr*$\rangle_n$ `)`
  | `UPDATE` $\langle$*id*$\rangle_{tab}$ `SET` $\langle$*id*$\rangle_{att}$ `=` $\langle$*db-int-expr*$\rangle$ { `WHERE` $\langle$*db-cond*$\rangle$ }$^?$
  | `DELETE FROM` $\langle$*id*$\rangle$ { `WHERE` $\langle$*db-cond*$\rangle$ }$^?$

2.2.9. Transaction management

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

$\langle$*stmt*$\rangle$ ::= `con.commit() ;`
  | `con.rollback() ;`

3. A symbolic execution for unit testing of transactional database programs

In the forthcoming section, we present our relational symbolic execution algorithm, able to generate adequate path-constraints, coupled with the necessary database schema integrity constraints, for any OLTP use case implementation, written in the language defined in the previous section.

3.1. Symbolic execution and path exploration

Originally introduced in [5], symbolic execution has been used as the core principle of many test data generation techniques. In some of these techniques (see e.g. [36] for an overview), symbolic execution is performed for a set of paths in the control-flow graph, according to some coverage criterion [4], by inspecting *statically* the tested code, i.e. independent of any concrete input values. More recently, test data generation techniques that combine symbolic execution with a *dynamic* path exploration process have also been proposed (see e.g. [11, 37, 38]). The code is first executed on concrete inputs to produce concrete outputs, but it is instrumented so that symbolic execution is performed in parallel, thereby generating the path-constraint corresponding to the concrete execution. By flipping the value of some of the logical formulas in this path-constraint, one can produce another path-constraint, whose solution describes new concrete inputs triggering the execution of a different path. The process is then repeated with these new inputs until a set of inputs and outputs has been generated for a set of paths covering a sufficient part of the code, according to some coverage criterion [4]. The point of using such a dynamic path exploration process is to provide seamlessly symbolic execution with concrete values. These values can then be used as a last resort to replace the statements for which constraints cannot be generated (like proprietary API calls) or handled by the solver (as they belong to a complex, exotic and/or undecidable logic). This process is called *concretisation* and is deeply discussed and evaluated in [39].

The relational symbolic execution algorithm described in this section receives as inputs the SQL DDL code of the database, the Java code of the method under test and an execution path through this method. It produces as output a constraint system mixing classical constraints with relational constraints. Solutions to this system are such that when the method is executed with respect to any of these solutions, its execution will follow the given path. Coupling this algorithm with any existing approach, either static or dynamic, to explore a set of paths to test in the method's control flow graph will enable generating test data for these paths. The set of paths for which test data are computed, as well as the process used to explore these paths, are thus parameters of the approach that we propose. This also enables the approach to be used within the context of different code coverage criteria [4].

3.2. Algorithm inputs and outputs

The inputs of our algorithm are an OLTP use case implementation, written in the language defined in the previous section, as well as an execution path through the Java method defined in this implementation. This execution path is supposed to be a finite path in the method's control flow graph [4]. It defines which branches were taken at each of the encountered if statements, how many times the body of each encountered loop was executed (this number must be finite), which assertions were eventually violated and, for each encountered try/catch statement, whether the catch clause was executed.

Our algorithm translates the path into a constraint system, combining the *path-constraint* with the *database schema integrity constraints*, expressed in SMT-Lib v2 [29], a widely adopted language used as the standard language for many SMT solvers. The generated constraints fit into the SMT-Lib AUFLIA logic, i.e. they can involve quantified array, uninterpreted functions and linear integer arithmetic constraints.

Solving the constraint system generated by the algorithm for a complete path enables finding values for the inputs (and, as we will see, also for the outputs) of the analysed Java method. The inputs include the content of each database table at the start of the method's execution, the value of every list received as argument by the method, and a value for the part of the input stream that is consumed during the method's execution. The outputs include the content of each database table at the end of the method's execution, the final value of each of the argument lists of the method, and possibly the value returned by the return statement. If the constraint system produced for a given path has no solution, this means that the path is infeasible. As the produced constraints are written in a quantified logic that is not generally decidable [28], it can happen that for a given path the solver may neither be able to find a solution for the generated constraint system, nor be able to establish that such a solution does not exist. This is coherent with the problem being not computable in general.

3.3. Algorithm principle

The algorithm performs a symbolic execution of the path received as input. First, it creates symbols to represent each of the inputs of the Java method and the initial content of each of the database tables. It generates also constraints over the symbols representing the initial values of the database tables. These constraints state that, initially, each table contains data that conform to the database schema integrity constraints.

Secondly, the algorithm analyses one by one the method's statements in the order specified by the path. Each time a statement sets or changes the value of a method variable or database table, the algorithm creates a symbol representing the newly defined value and generates constraints linking this new symbol to the symbols created previously. These constraints describe how the new value can be computed from the values of the database tables and method variables before the statement's execution. Moreover, every time the value of a database table is changed, constraints are also added to state that the new value satisfies the database schema integrity constraints. Finally, every time an if, while, assert or try/catch statement is encountered, the algorithm generates an additional constraint over the symbols such that when the method is executed with respect to values satisfying this constraint, the execution is guaranteed to take the considered path.

3.4. Execution example

In the following paragraphs, we illustrate the execution of the algorithm over the sample code given in Figure 1 (page 7). We detail each step of the symbolic execution process over the path where the while loop is executed once, the else branch of both if statements is taken, and the catch clause of the try/catch is executed (lines 1-6, 8-15, 18-21, 3, 22).

At each step, we present the rules used by our algorithm to generate the corresponding SMT-Lib symbols and constraints. It should be noted that SMT-Lib syntax is inspired by the S-expressions from Lisp, where classical expressions like 2 + 2 + 2 are written as (+ 2 (+ 2 2)). All the rules that are part of the complete set of rules defining our symbolic execution algorithm for the subset of Java/SQL defined in the previous section are presented during one of these steps, and/or described formally in a set of tables available at the end of this section.

The first step executed by our algorithm is to generate SMT-Lib symbols and constraints for the SQL DDL code of the database schema. For the database schema described in Figure 1, the generated SMT-Lib code is detailed in the frame below. For the reader's convenience, the corresponding Java/SQL code will be reminded as a preliminary comment in the SMT-Lib code.

First of all, the algorithm generates new symbol types for the kind of objects stored in each table defined by the schema (the lines prefixed by (0) in the SMT-Lib code below). It will then generate symbols and constraints describing the input content of each of these tables. The used modelling is inspired by the one proposed in [28] for relational types. First, a symbol is created (1) to represent the initial set of objects in each table. Typed as a boolean function, it returns true for each object present in the input content of the table. Symbols typed as integer functions are then generated (2) to associate to each object in the table one of its attribute values. Finally, constraints are generated to enforce on this input content all the check constraints (3), primary key constraints (4), and foreign key constraints (5) defined in the schema.

Note that the original SQL table and attribute names, as well as the original Java variable names, are used as SMT-Lib symbols, suffixed by the natural number 1 (e.g. *book*1 or *id*1 in the SMT-Lib code below), which represents the fact that the current symbols represent the initial values of the represented tables, attributes or variables. Subsequent values of a same table, attribute or variable will be represented by the same symbol suffixed with successive numbers. Moreover, as detecting corner cases linked to arithmetic overflow is not a priority of our work, we have used SMT-Lib symbols typed as mathematical integers to represent the fixed-width integers used the code.

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

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

; New types for tables
(0) (declare-sort book)
(0) (declare-sort shelf)
; Input content of table Book
(1) (declare-fun book1 (book) Bool)
(2) (declare-fun shelfId1 (book) Int)
(2) (declare-fun code1 (book) Int)
(4) (assert (forall ((a book) (b book))
        (=> (and (and (book1 a) (book1 b)) (= (code1 a) (code1 b))) (= a b))))
```

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

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

The second step executed by our algorithm is to define a new SMT-Lib symbol type (called BoundedList) for lists of integers. All the symbols that will be subsequently generated to represent the value of a variable typed as a Java list will be part of this new SMT-Lib type. A BoundedList symbol represents a record composed of three fields: the *isNull* field is typed as boolean, the *size* field is typed as integer and the *elements* field is typed as array of integers. If the *isNull* field is true, then the symbol represents the Java `null` value. Otherwise, the field *size* represents the size of the list, and the field *elements* represents an array whose indexes 0 to ($size - 1$) contain the elements of the list in the right order.

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

It should be noted that the BoundedList symbol type is defined using an algebraic datatype declaration, where "mk-bounded-list" will be the ad-hoc constructor for list objects. If datatype declarations are handled by several SMT solvers, the related syntax has not been standardised in SMT-Lib. In this work, we use the datatype notation available in the Microsoft Z3 SMT solver [30].

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

```
; INPUT PARAMETER: List<Integer> newBooks
(declare-const newbooks1 BoundedList)
(assert (=> (not (isNull newbooks1)) (>= (size newbooks1) 0)))
```

It should be noted that the second constraint enforces a general property of BoundedList objects, described earlier. However, it will be enforced one by one for each BoundedList object. This is an optimisation compared to using a more general constraint, quantified over the set of all possible BoundedList objects. In order to solve such a constraint, the solver would indeed be required to instantiate correctly the quantifier by itself, making the constraint more costly to solve.

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

```
; int i = 0;
(1) (declare−const i1 Int)
(2) (assert (= i1 0))
```

```
; List<Integer> addedBooks = new ArrayList<Integer>();
(1) (declare−const addedbooks1 BoundedList)
(2) (assert (not (isNull addedbooks1)))
(2) (assert (= (size addedbooks1) 0))
```

The next statement in the path is a while statement. As the path specifies that the loop body must be executed, a constraint is generated to specify that the loop condition at this point of time should be true:

```
; ENTERING LOOP: while ( !(newBooks == null) & (i < newBooks.size()) )
(assert (and (not (isNull newbooks1)) (< i1 (size newbooks1))))
```

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

```
; int error = 0;
(declare−const error1 Int)
(assert (= error1 0))
```

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

```
; int theShelf = in.nextInt();
(declare−const theshelf1 Int)
```

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

```
;  ResultSet shelves
(1) (declare−const shelves1Size Int)
(2) (declare−fun shelves1List (Int) shelf)
```

Constraints are then generated to specify that a row is part of the ResultSet if and only if it is part of the current content of the table on which the select query is executed and that it enforces the WHERE condition of the select query. In order to do so, the modelling proposed in [28] for constraining the content and cardinality of relations, in a way so that the constraints can be effectively solved by Z3, is used. Following [28], new constraints are added (1) to define a function *shelves1InvertedList* which is the inverse of *shelves1List*.

This function is used (1) to ensure that *shelves1List* defines a one to one correspondence between the integers $0 \leqslant i < shelves1Size$ and the elements in the ResultSet. Helper code (2) is added to ensure an efficient pattern-based quantifier instantiation [40] by the solver, using the `:pattern` keyword [41].

```
;  = con.createStatement().executeQuery("SELECT id FROM shelf WHERE id="+theShelf);
(1)  (declare−fun shelves1InvertedList (shelf) Int)
(2)  (declare−fun shelves1Trigger (Int) Bool)
(1)  (assert (and (>= shelves1Size 0)
                (=> (= shelves1Size 0)
                      (forall ((c shelf)) (not (and (shelf1 c) (= (id1 c) theshelf1 )))))))
(1)  (assert (forall ((c shelf))
   (=> (and (shelf1 c) (= (id1 c) theshelf1))
        (and (>= (shelves1InvertedList c) 0) (< (shelves1InvertedList c) shelves1Size )))))
(1)  (assert (forall ((c shelf))
   (=> (and (shelf1 c) (= (id1 c) theshelf1))
        (= c (shelves1List (shelves1InvertedList c))))))
(1)  (assert (forall ((i Int))
   (=> (and (>= i 0) (< i shelves1Size))
        (= i (shelves1InvertedList (shelves1List i))))))
(1)  (assert (forall ((i Int))
    (! (=> (and (>= i 0) (< i shelves1Size))
            (and (shelf1 (shelves1List i)) (= (id1 (shelves1List i)) theshelf1)))
(2)     :pattern (shelves1Trigger i))))
(2)  (assert (=> (>= 0 shelves1Size) (shelves1Trigger 1)))
(2)  (assert (forall ((i Int))
    (! (=> (and (>= i 0) (< i shelves1Size))
            (shelves1Trigger (+ i 1)))
    :pattern (shelves1Trigger i))))
```

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

For each ResultSet object, the algorithm records the number of times the `next()` method has been called on this object. This value represents the index increased by one of the row pointed by the cursor of the ResultSet at the current execution state of the path. When the boolean value returned by the '`next()`' method is used in an if or while condition, this value states if the number of rows in the ResultSet is greater or equal to the number of times the '`next()`' method has been called so far on this ResultSet. In this case, `shelves.next()` will return true if the ResultSet `shelves` contains at least one row (as `shelves.next()` has been called once on the ResultSet):

```
; TAKING ELSE BRANCH OF: if ( ! shelves.next() )
(assert (>= shelves1Size 1))
```

Symbolic execution for update creates a new symbol (1) typed as an integer function, that will replace the previous symbol associating the attribute value to each object in the table. As this new symbol is the second one to represent the value of the attribute *numberOfBooks*, it is named *numberOfBooks2*. A couple of constraints (2)(3) is then generated to relate the old and new attribute values in the table: one for the rows that do not match the WHERE condition (2), and one for those that do (3). Finally, constraints are added to specify that no integrity constraint was violated during the update. In this case, a constraint (4) is added to state that the updated attribute values still enforce the check constraint defined in the database schema.

```
; con.createStatement().execute(
;  "UPDATE shelf SET numberOfBooks=numberOfBooks+1 WHERE id ="+theShelf);
(1) (declare−fun numberOfBooks2 (shelf) Int)
(2) (assert (forall ((p shelf))
        (=> (or (and (shelf1 p) (not (= (id1 p) theshelf1))) (not (shelf1 p)))
            (= (numberOfBooks2 p) (numberOfBooks1 p)))))
(3) (assert (forall ((p shelf))
        (=> (and (shelf1 p) (= (id1 p) theshelf1))
            (= (numberOfBooks2 p) (+ (numberOfBooks1 p) 1)))))
(4) (assert (forall ((a shelf)) (> (numberOfBooks2 a) 0)))
```

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

```
; TAKING THE CATCH BRANCH OF:
;  try { con.createStatement().execute(
;  "INSERT INTO book VALUES ("+newBooks.get(i)+","+theShelf+")");
; } catch (SQLException e) {
(1) (assert (or (exists ((a book)) (and (book1 a)
                                   (= (code1 a) (select (elements newbooks1) i1))))
            (forall ((a shelf)) (=> (shelf1 a)
                                   (not (= (id1 a) theshelf1 ))))))
(2) (assert (not (isNull newbooks1)))
(2) (assert (>= i1 0))
(2) (assert (< i1 (size newbooks1)))
```

The content of the catch block is then symbolically executed:

```
; error = 1;
(declare−const error2 Int)
(assert (= error2 1))
```

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

```
; TAKING ELSE BRANCH OF: if (error==0)
(assert (not (= error2 0)))
```

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

The assignment statement is then symbolically executed:

```
; i = i + 1;
(declare−const i2 Int)
(assert (= i2 (+ i1 1)))
```

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

```
; ESCAPING LOOP: while ( !(newBooks == null) & (i < newBooks.size()) )
(assert (not (and (not (isNull newbooks1)) (< i2 (size newbooks1)))))
```

As a return statement is met, the symbolic execution can be stopped and the generated SMT-Lib constraint model can be returned. The Z3 solver can now be asked to find a valuation for the defined symbols satisfying the constraints. As the algorithm records what symbols represent the initial, respectively final, values of a variable or table, the input and output values of the method (for the considered path) can easily be extracted from the solution to the constraint system.

For our example, the set of 29 constraints was solved by Z3 in 24ms (1.8 GHz Intel Core i5, 8GB Ram) and the test data that were obtained from the solution to the constraint system are summarised in the following tables:

<table>
<tr><th colspan="3">Inputs</th></tr>
<tr><th>Name</th><th>Symbol(s)</th><th>Value</th></tr>
<tr><td>TABLE
shelf</td><td>CONTENT:
shelf1,
ATTRIBUTES:
id1
numberOfBooks1</td><td><table><tr><td>id</td><td>n.Books</td></tr><tr><td>6</td><td>1</td></tr><tr><td>12</td><td>1</td></tr></table></td></tr>
<tr><td>TABLE
book</td><td>CONTENT:
book1
ATTRIBUTES:
code1
shelfId1</td><td><table><tr><td>code</td><td>s.Id</td></tr><tr><td>4</td><td>12</td></tr></table></td></tr>
<tr><td>newBooks</td><td>newbooks1</td><td>[4]</td></tr>
<tr><td>in.nextInt()</td><td>theshelf1</td><td>[6]</td></tr>
</table>

<table>
<tr><th colspan="3">Outputs</th></tr>
<tr><th>Name</th><th>Symbolic(s)</th><th>Value</th></tr>
<tr><td>TABLE
shelf</td><td>CONTENT:
shelf1,
ATTRIBUTES:
id1
numberOfBooks1</td><td><table><tr><td>id</td><td>n.Books</td></tr><tr><td>6</td><td>1</td></tr><tr><td>12</td><td>1</td></tr></table></td></tr>
<tr><td>TABLE
book</td><td>CONTENT:
book1
ATTRIBUTES:
code1
shelfId1</td><td><table><tr><td>code</td><td>s.Id</td></tr><tr><td>4</td><td>12</td></tr></table></td></tr>
<tr><td>newBooks</td><td>newbooks1</td><td>[4]</td></tr>
<tr><td>addedBooks</td><td>addedbooks1</td><td>[]</td></tr>
</table>

3.5. Constraint Generation Rules

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

In each of these tables, any declaration of a new symbol leverages a generator providing a fresh symbol identifier, i.e. which has still not been used in the SMT-Lib code generated so far. This is denoted by (for a function declaration):

```
(declare-fun freshSym Type)
```

and by (for a constant declaration):

```
(declare-const freshSym Type)
```

All the subsequent references to freshSym in the table represent this newly declared symbol. The generated fresh symbols are named according to the naming rule detailed along the example given in the previous subsection.

Finally, it should be noted that assertions are handled as if statements. For example :

```
assert x == 0;
```

is handled as:

```
if (!(x == 0))
        End of computation with AssertionError
```

Assertions are particularly useful for symbolic execution as they let the programmer express additional constraints otherwise non-obvious to the solver.

Table 1: SMT-Lib Translation Abbreviations List

Abbreviation	**Meaning**
$smt2Of(x)$	Java condition/expression x translated into a corresponding SMT-Lib condition/expression.
$smt2Of(x,t,r)$	SQL condition/expression x evaluated for row r in table t translated into a corresponding SMT-Lib condition/expression.
name(x)	**if** (x refers to a database table name) **then** The symbol that represents the current content of table x **else if** (x refers to a database attribute name) The symbol that represents the current values of attribute x **else if** (x refers to a Java variable name) The symbol that represents the current content of the Java variable x
att_i	Name of the i_{th} attribute in the list of attributes of table $\langle id \rangle$
pk	Name of the primary key attribute of table $\langle id \rangle$.
pk^{pos}	Position of primary key in the list of attributes of table $\langle id \rangle$
fk_i^{tab}	Name of the table referenced by the i_{th} foreign key in the list of foreign keys of table $\langle id \rangle$
fk_i^{pk}	Name of the primary key attribute of the table referenced by the i_{th} foreign key in the list of foreign keys of table $\langle id \rangle$
fk_i^{pos}	Position of the foreign key attribute, declared by the i_{th} foreign key in the list of table $\langle id \rangle$, in the list of attributes of table $\langle id \rangle$
ifk_i^{tab}	Name of the table where is declared the i_{th} foreign key referencing table $\langle id \rangle$ in the whole schema
ifk_i^{att}	Name of the foreign key attribute declared by the i_{th} foreign key referencing table $\langle id \rangle$ in the schema
co_i^{pos}	Position of the attribute constrained by the i_{th} check constraint declared in table $\langle id \rangle$
co_i^{right}	Inverted right part of the i_{th} check constraint declared in table $\langle id \rangle$ (i.e. inverted right part of "a $>$ 0" is "$<$ 0")

Table 2: SMT-Lib constraints generation rules for INSERT statements

`INSERT INTO` $\langle id \rangle$ `VALUES` ($\langle$*int-expr*$\rangle_1$, ... , $\langle$*int-expr*$\rangle_i$, ... , $\langle$*int-expr*$\rangle_n$)

if (*no exception thrown in path for this INSERT*) {

; Inserted primary key value does not already exist
$(\textbf{assert}(\textbf{forall}((\mathrm{a}\langle id\rangle))(=>(\mathsf{name}(\langle id\rangle)\ \mathrm{a})(\textbf{not}(=\ (\mathsf{name}(\mathsf{pk})\ \mathrm{a})\ smt2Of(\langle int\text{-}expr\rangle_{\mathsf{pk}^{pos}}))))))$

; Inserted values constrained by the i^{th} foreign key reference existing rows
$(\textbf{assert}(\textbf{exists}((\mathrm{a}\ \mathsf{fk}_i^{tab}))(\textbf{and}(=\ (\mathsf{name}(\mathsf{fk}_i^{pk})\ \mathrm{a})\ smt2Of(\langle int\text{-}expr\rangle_{\mathsf{fk}_i^{pos}}))\ (\mathsf{name}(\mathsf{fk}_i^{tab})\ \mathrm{a}))))$

; Symbol for new table content
$(\textbf{declare–fun}\ \mathsf{freshSym}\ (\langle id\rangle)\ \textbf{Bool})$

; Constraints describing new table content
$(\textbf{assert}\ (\textbf{forall}\ ((\mathrm{a}\ \langle id\rangle))\ (=>\ (\mathsf{name}(\langle id\rangle)\ \mathrm{a})\ (\mathsf{freshSym}\ \mathrm{a}))))$
$(\textbf{assert}\ (\textbf{exists}\ ((\mathrm{a}\ \langle id\rangle))\ (\textbf{and}\ (=\ (\mathsf{att}_i\ \mathrm{a})\ smt2Of(\langle int\text{-}expr\rangle_i))\ (\mathsf{freshSym}\ \mathrm{a}))))$
$(\textbf{assert}\ (\textbf{forall}\ ((\mathrm{a}\ \langle id\rangle))$
$(=>(\textbf{and}(\textbf{not}\ (\mathsf{name}(\langle id\rangle)\ \mathrm{a}))(\textbf{not}\ (=\ (\mathsf{att}_i\ \mathrm{a})\ smt2Of(\langle int\text{-}expr\rangle_i))))(\textbf{not}\ (\mathsf{freshSym}\ \mathrm{a})))))$

; No duplicate inserted row
$(\textbf{assert}\ (\textbf{forall}\ ((\mathrm{a}\ \langle id\rangle)\ \ (\mathrm{b}\ \langle id\rangle))$
$(=>(\textbf{and}(\textbf{and}\ (\mathsf{freshSym}\ \mathrm{a})\ (\mathsf{freshSym}\ \mathrm{b}))\ (=\ (\mathsf{pk}\ \mathrm{a})\ (\mathsf{pk}\ \mathrm{b})))\ (=\ \mathrm{a}\ \mathrm{b}))))$

} **else** {
// Logical disjunction between every possible constraint
// violation given the database schema and this insert:

; The inserted primary key value already exists in the table
$(\textbf{exists}\ ((\mathrm{a}\ \langle id\rangle))\ \ (\textbf{and}\ (\mathsf{name}(\langle id\rangle)\ \mathrm{a})\ (=\ (\mathsf{name}(\mathsf{pk})\ \mathrm{a})\ \ smt2Of(\langle int\text{-}expr\rangle_{\mathsf{pk}^{pos}}))))$

; i^{th} inserted foreign key value does not reference a row:
$(\textbf{forall}\ \ ((\mathrm{a}\ \mathsf{fk}_i^{tab}))(=>\ (\mathsf{name}(\mathsf{fk}_i^{tab})\ \mathrm{a})\ (\textbf{not}\ (=\ (\mathsf{name}(\mathsf{fk}_i^{pk})\ \mathrm{a})\ smt2Of(\langle int\text{-}expr\rangle_{\mathsf{fk}_i^{pos}})))))$

; An inserted attribute violates the i^{th} check constraint:
$(\textbf{not}\ (\mathsf{co}_i^{right}\ \ smt2Of(\langle int\text{-}expr\rangle_{\mathsf{co}_i^{pos}})))$

}

Table 3: SMT-Lib constraints generation for UPDATE statements

`UPDATE` $\langle id \rangle$ `SET` $\langle id \rangle_{att}$ `=` $\langle$*db-int-expr*$\rangle$ `WHERE` $\langle$*db-cond*$\rangle$

if (*no exception thrown in path for this UPDATE*) {

; Symbol for new attribute values
(**declare−fun** freshSym ($\langle id \rangle$) **Int**)
; Constraints describing new attribute values
(**assert** (**forall** ((a $\langle id \rangle$)) (=> (**or** (**and** (name($\langle id \rangle$) a) (**not** *smt2Of*($\langle$*db-cond*$\rangle$,$\langle id \rangle$,a)))
(**not** (name($\langle id \rangle$) a))) (= (freshSym a) (name($\langle id \rangle_{att}$) a)))))
(**assert** (**forall** ((a $\langle id \rangle$)) (=> (**and** (name($\langle id \rangle$) a) *smt2Of*($\langle$*db-cond*$\rangle$,$\langle id \rangle$,a))
(= (freshSym a) *smt2Of*($\langle$*db-int-expr*$\rangle$,$\langle id \rangle$,a)))))
; Update on attribute constrained by i^{th} foreign key not leave pending references
(**assert** (**forall** ((a $\langle id \rangle$)) (=> (name($\langle id \rangle$) a)
(**exists** ((b fk_i^{tab})) (**and** (name(fk_i^{tab}) b) (= (freshSym a) (name(fk_i^{pk}) b)))))))
; Update on attribute constrained by primary key not leaves duplicate attribute values
(**assert** (**forall** ((a $\langle id \rangle$) (b $\langle id \rangle$))
(=> (**and** (**and**(name($\langle id \rangle$) a) (name($\langle id \rangle$) b)) (= (freshSym a) (freshSym b))) (= a b))))
; Update on primary key referenced by i^{th} foreign key does not leave pending references
(**assert** (**forall** ((a ifk_i^{tab})) (=> (name(ifk_i^{tab}) a)
(**exists** ((b $\langle id \rangle$))(**and** (name($\langle id \rangle$) b) (= (name(ifk_i^{att}) a) (freshSym b)))))))
; Update on attribute constrained by i^{th} check constraint does not violate the constraint
(**assert** (**forall** ((a $\langle id \rangle$)) (co_i^{right} (freshSym a))))

} **else** {
// Logical disjunction between every possible constraint
// violation given the database schema and this update:

; Update on primary key leads to duplicate attribute values
(**exists** ((a $\langle id \rangle$) (b $\langle id \rangle$)) (**and**(**and** (name($\langle id \rangle$) a) (**and** (name($\langle id \rangle$) b) (**not** (= a b))))
(**or** (**and** *smt2Of*($\langle$*db-cond*$\rangle$,$\langle id \rangle$,a) (**and** *smt2Of*($\langle$*db-cond*$\rangle$,$\langle id \rangle$,b)
(= *smt2Of*($\langle$*db-int-expr*$\rangle$,$\langle id \rangle$,a) *smt2Of*($\langle$*db-int-expr*$\rangle$,$\langle id \rangle$,b))))
(**and** (**not** *smt2Of*($\langle$*db-cond*$\rangle$,$\langle id \rangle$,a)) (**and** *smt2Of*($\langle$*db-cond*$\rangle$,$\langle id \rangle$,b)
(= (name($\langle id \rangle_{att}$) a) *smt2Of*($\langle$*db-int-expr*$\rangle$,$\langle id \rangle$,b)))))))
; Update on primary key referenced by the i^{th} foreign key leaves pending references
(**exists** ((a $\langle id \rangle$) (b $\langle id \rangle$)) (**and** (**and** (name($\langle id \rangle$) a) (name(ifk_i^{tab}) b))
(**and** (**and** (**not** (= (name($\langle id \rangle_{att}$) a) *smt2Of*($\langle$*db-int-expr*$\rangle$,$\langle id \rangle$,a)))
(= (name($\langle id \rangle_{att}$) a) (name(ifk_i^{pk}) b))) *smt2Of*($\langle$*db-cond*$\rangle$,$\langle id \rangle$,a))))
; Update on attribute constrained by i^{th} foreign key leaves pending references
(**exists** ((a $\langle id \rangle$)) (**and** (**and** (name($\langle id \rangle$) a) *smt2Of*($\langle$*db-cond*$\rangle$,$\langle id \rangle$,a))
(**not** (**exists** ((b name(fk_i^{tab}))) (= (name(fk_i^{pk}) b) *smt2Of*($\langle$*db-int-expr*$\rangle$,$\langle id \rangle$,a))))))
; Update on attribute constrained by i^{th} check constraint violates the constraint
(**exists** ((a $\langle id \rangle$)) (**and** (**and** (name($\langle id \rangle$) a) *smt2Of*($\langle$*db-cond*$\rangle$,$\langle id \rangle$,a))
(**not** (co_i^{right} *smt2Of*($\langle$*db-int-expr*$\rangle$,$\langle id \rangle$,a)))))

}

Table 4: SMT-Lib constraints generation for DELETE statements

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

Table 5: SMT-Lib constraints generation rules for add(int) and remove(int) statements

$\langle id \rangle$`.add(` $\langle \textit{int-expr} \rangle$ `);`
(**declare–const** freshSym BoundedList) (**assert** (**not** (isNull name($\langle id \rangle$)))) (**assert** (**not** (isNull freshSym))) (**assert** (= (size freshSym) (+ (size name($\langle id \rangle$)) 1))) (**assert** (= (elements freshSym)   (**store** (elements name($\langle id \rangle$)) (size name($\langle id \rangle$)) *smt2Of*($\langle \textit{int-expr} \rangle$)))))
$\langle id \rangle$`.remove(` $\langle \textit{int-expr} \rangle$ `);`
(**declare–const** freshSym BoundedList) (**assert** (**not** (isNull name($\langle id \rangle$)))) (**assert** (**not** (isNull freshSym))) (**assert** (>= (size name($\langle id \rangle$)) 1)) (**assert** (= (size freshSym) (– (size name($\langle id \rangle$)) 1))) (**assert** (>= *smt2Of*($\langle \textit{int-expr} \rangle$) 0)) (**assert** (< *smt2Of*($\langle \textit{int-expr} \rangle$) (size name($\langle id \rangle$)))) (**assert** (**forall** ((i **Int**)) (=> (**and** (>= i 0) (< i *smt2Of*($\langle \textit{int-expr} \rangle$))))   (= (**select** (elements name($\langle id \rangle$)) i) (**select** (elements freshSym) i))))) (**assert** (**forall** ((i **Int**)) (=> (**and** (>= i *smt2Of*($\langle \textit{int-expr} \rangle$))) (< i (size freshSym)))   (= (**select** (elements name($\langle id \rangle$)) (+ i 1)) (**select** (elements freshSym) i)))))

4. Experimental evaluation

In the upcoming section, we introduce a test generation tool based on the symbolic execution algorithm detailed in the previous section. This test generation tool is experimented over a number of sample OLTP use case implementations. First, these experimentations enable us to compare the ability of the Alloy and Z3 solver to handle the constraints produced by relational symbolic execution. Secondly, they enable us to estimate the practical scalability of relational symbolic execution with current solver technologies. Finally, they enable us to compare the performance of relational symbolic execution against an alternate tool, based on normalisation of SQL into native code.

4.1. Test generation tool based on our relational symbolic execution algorithm

In order to evaluate experimentally the effectiveness and efficiency of our relational symbolic execution approach, we have built a test generation tool that implements it. This tool integrates our relational symbolic execution algorithm with a control-flow graph exploration process (based on a static inspection of the tested code) and the Microsoft Z3 solver as a back-end solver.

Our tool was implemented in Java and works as follows. Given some code implementing an OLTP use case, written in the language detailed in section 2, this code is parsed and a Java representation of it is built as an abstract syntax tree (AST). The used parser was generated with the Cup parser generator[2] and an LALR version of the grammar defined at section 2. By traversing the generated AST, the tool performs a depth-first search of the Java method's control-flow graph, considering all the paths that execute the body of each loop in the method at most K times, where K is a parameter of the tool. Consequently, our tool satisfies a finitely applicable variant of the common path-coverage criterion [4], similar to the loop count-K criterion originally proposed in [42].

In parallel to this exploration of the control-flow graph, the tool applies the relational symbolic execution algorithm proposed in the previous section. The constraints are produced but also solved on-the-fly, using the Z3 solver, along the path exploration. This means that as soon as new constraints are generated by symbolic execution for a newly traversed statement, the set of constraints generated so far are directly solved. If these constraints are unsatisfiable, the path prefix explored so far in the graph is infeasible and the depth-first search process immediately backtracks. This is a common optimisation in symbolic execution, enabling the early detection of infeasible path prefixes, so that the path exploration can immediately prune out all the paths starting with such a prefix. When the constraints corresponding to a feasible path are solved, test input and output data (including database content) are extracted from the solution and recorded as constituting one *test-case*. Once all the paths have been explored, a so-called *test-suite* comprising all the generated test-cases is returned to the user. The tool also keeps a separate list of the paths for which the solver heuristics fail solving the constraints.

During our experiments, the tool was run on a dual core Intel Core i5 processor at 1.8GHz (256 KB L2 cache per core and 3 MB L3 cache) with 8GB of dual channel DDR3 memory at 1600 MHz. The runtime environment was the Oracle JVM 1.6.0_45 under a 32-bit edition of Microsoft Windows 8.1. The version 4.3.0 of the Microsoft Z3 solver was used.

[2] `http://www2.cs.tum.edu/projects/cup/`

4.2. Sample OLTP use case implementations used for experimental evaluation

The sample code used for our experimental evaluation is composed of eighteen Java methods, performing SQL operations over relational databases. These methods can be divided into four groups:

- The first group contains nine methods. Each of them was crafted to systematically evaluate the correct symbolic execution of one of the different Java and SQL constructs handled by our algorithm. As such, the methods in this group exercise the different behaviours of the integer and list operators, conditional and loop statements, SQL statements and transaction management primitives.
- The second group contains three methods crafted as implementations of OLTP use cases. The first method in this group performs repeated manipulations of integers and lists to compare and save some data. The second method performs many interleaved reads and writes in a database containing four tables, representing regular or prospect customers that make purchases of products. The third method mixes SQL statements with traditional Java code and uses SQL transactions. The manipulated database contains two tables that represent authors writing theatre plays.
- The third and fourth groups contain Java methods extracted respectively from UnixUsage[3] and RiskIt[4], the two open-source software that have been used in [23], as a basis for evaluating their proposed test generation tool, called SynDB, based on SQL normalisation into native code. UnixUsage is a monitoring application for Unix, manipulating a database with eight tables and thirty one attributes. RiskIt is an insurance application, manipulating a database with thirteen tables and fifty-seven attributes.

Together, the methods from our testbed constitute a set of five hundred lines of code, containing notably eighty SQL statements (including SELECT, INSERT, UPDATE, DELETE statements, as well as transaction management code), over databases containing up to thirteen tables (subject to primary key, foreign key and check constraints).

Detailed statistics for each of these methods can be found in table 6, including the value of K used in our tool to limit the loop exploration depth for each method. Given this value of K, the number of test case to generate (i.e. the number of feasible paths in the K-bounded control-flow graph) is provided for each method. The methods from groups 1 and 2 have a small number of feasible paths because they were constructed in this way on purpose, for assessing the soundness of our test generation tool. The methods from groups 3 and 4 have a small number of feasible paths because because they have rather simple control-flow graphs.

Finally, the code of these methods, as well as the generated test data, can be found on the web[5].

4.3. Evaluating the Alloy and Z3 solvers for relational symbolic execution

4.3.1. Performance comparison over common samples

Our first objective with the experiments was to compare the ability of the Alloy and Z3 solvers to solve effectively and efficiently the constraints produced by relational symbolic

[3]`http://sourceforge.net/projects/se549unixusage`

[4]`https://riskitinsurance.svn.sourceforge.net`

[5]`https://staff.info.unamur.be/mmr/scp/`

Table 6: Statistics for the selected samples

Name	LOCs	Number of SQL statements	Number of tables	K Depth	Number of Test Cases
CRAFTED SAMPLES					
- Language Unit Testing					
Integers	7	0	0	3	1
Lists	9	0	0	3	1
If and While	30	0	0	1	1
Conditions	30	0	0	1	1
Select	23	7	2	3	1
Insert	14	7	2	3	1
Update	20	11	2	3	1
Delete	6	3	2	3	1
Transactions	13	2	2	3	2
- Realistic OLTP Methods					
Handling Data	45	1	1	5	23
Clients and Products	56	18	4	5	4
Play Catalog	72	4	2	2	63
SAMPLES EXTRACTED FROM OPEN-SOURCE SOFTWARE					
- UnixUsage					
courseNameExists	7	1	8	1	2
getCommandsByCategory	10	1	8	1	2
getCourseIDByName	10	1	8	1	2
getDepartmentIDByName	11	1	8	1	2
- RiskIt					
createNewUser	91	7	13	2	3
deleteUsers	55	16	13	2	2
ALL METHODS	509	80	up to 13	up to 5	113

execution. In order to do so, we used the Alloy version of our tool from [26] and the SMT-Lib version presented here to generate test cases for the methods of the first and second groups from our testbed.

The Alloy solver basically solves the constraints by setting upper bounds on the scope of the possible integers, as well as on the cardinality of the possible different rows which can appear in the solution for each different table. This makes finite the set of possible solutions, which can then be exhaustively explored, by transforming the constraints into an equisatisfiable SAT instance, solved by a dedicated SAT solver. As a consequence, if no solution is found with the default minimal bound value, the testing tool should repeat the process recursively with a higher value as bound, until a time-out is reached. When this time-out occurs, the tool considers the underlying path as infeasible. The time-out value should thus be long enough to enable finding a proper model for feasible paths. At the same time, it should not be too long to detect infeasible paths in minimal time. Finding an optimal time-out value appeared to be very difficult in practice, as it depends on the size and complexity of the constraints to solve.

In the left part of table 7, we synthesise the results obtained with the Alloy version of the tool, over the feasible paths of the methods from groups one and two. For each method, the table provides the minimal bound values enabling Alloy to find inputs for all the feasible paths in the method. It also provides the total number of constraints generated for these feasible paths and the minimal time within which the solver was able to solve these constraints. The methods named "Integers", "Update", "Delete" and "Clients and Products" involve either large integer values or repeated actions on a single table. As a consequence, the constraints generated for these methods require large enough bound values to be solved, which had in practice to be found manually, using a costly trial and error approach. Moreover, the results show that once the bounds are increased, the size of the SAT instance generated by the Alloy solver quickly explodes, while the solving time blows up from dozens of milliseconds to several minutes. Finding inputs for the "Handling Data" and the "Play Catalog" methods, revealed to be intractable in reasonable time, with the Alloy solver and without any manual help.

In accordance with [28], where Z3 is used as a complement to the Alloy solver for proving instances unsatisfiable, the Z3 solver was able to detect almost instantaneously the unsatisfiability of the generated constraints. As a consequence, the SMT-Lib version of the tool was able to detect properly and efficiently the infeasible path prefixes in the control-flow graphs of the methods from groups one and two. As we used a more recent version of Z3 than in [28], involving notably a new model-based quantifier instantiation (MBQI) technique for solving quantified constraints, we could hope that the solver would also be able to find efficiently satisfying models for the generated constraints and thus produce inputs for feasible paths. The obtained results are detailed in the right part of table 7. The Z3 solver was able to find inputs for each feasible path in milliseconds, always outperforming the Alloy solver, even when minimal bound values were used in the Alloy version. As an example, the Alloy version took more than thirty eight minutes and a half to find test inputs for the four feasible paths of method "Clients and Products", when the Z3 version was able to do it in three seconds and a half. Finding inputs for the "Handling Data" and the "Play Catalog" methods, which was intractable using Alloy, became possible in less than two minutes using Z3.

To quantify the impact of using Z3 4.3.0 (with MBQI), instead of the version 2.16

Table 7: Z3 and Alloy statistics for the methods of the first and second groups

Name	Feasible Paths with Alloy			Feasible Paths with Z3	
	Cardinality Bound	**Constraints**	**Solver Time**	**Constraints**	**Solver Time**
CRAFTED SAMPLES					
- Language Unit Testing					
Integers	1 but 5 int	6	129ms	4	16ms
Lists	4	10	186ms	45	16ms
If and While	1	17	46ms	15	15ms
Conditions	1	19	37ms	19	16ms
Select	3	41	221ms	61	47ms
Insert	1	23	40ms	22	16ms
Update	7 but 8 int	39	6m 6s 299ms	41	63ms
Delete	1 but 8 int	16	51s 638ms	15	15ms
Transactions	1	38	137ms	44	30ms
- Realistic OLTP Methods					
Handling Data	-	-	-	2154	641ms
Clients and Products	8 but 16 clients	502	38m 37s 506ms	612	3s 470ms
Play Catalog	-	-	-	4343	1s 361ms

(without MBQI) used in [28], we have re-run the experiments with the Z3 version used in [28]. These new experiments showed that Z3 2.16 can detect infeasible path prefixes with the same efficiency as Z3 4.3.0, but is not able to find models for the constraints generated for feasible paths. [28] showed that Z3 2.16 could be used as a complement to the Alloy solver for proving Alloy instances unsatisfiable. Our work completes this work by indicating that more recent versions of Z3 could advantageously and totally replace the Alloy solver both for unsatisfiable and satisfiable constraints.

4.3.2. Practical scalability limits

Our second objective with the experiments was to estimate the practical scalability limits of current solver technology for the generated constraints. In order to do so, we have measured the number of generated constraints and the constraint solving time for relational symbolic execution (using Z3) of the following method, containing a single execution path, traversing a linear block of SQL statements. These statements involve SQL SELECT, as well as INSERT, UDPATE and DELETE statements, and the latter act on fields subject to primary key, foreign key and check constraints.

```
CREATE TABLE t1 (
idt1 INTEGER NOT NULL,
 fieldt1 INTEGER NOT NULL,
CONSTRAINT t1PK PRIMARY KEY (idt1),
CHECK(idt1 > 0));
CREATE TABLE t2 (
idt2 INTEGER NOT NULL,
 fieldt2 INTEGER NOT NULL,
CONSTRAINT t2PK PRIMARY KEY (idt2),
CONSTRAINT t2FK FOREIGN KEY (fieldt2) REFERENCES t1(idt1),
CHECK(idt2 > 0));
```

```
void test (Connection con,Scanner in) throws SQLException {
int i = 1;
con.createStatement().execute("INSERT INTO t1 VALUES ("+i+","+i+")");
con.createStatement().execute("INSERT INTO t1 VALUES ("+(i+1)+","+(i+1)+")");
con.createStatement().execute("INSERT INTO t2 VALUES ("+i+","+i+")");
con.createStatement().execute("INSERT INTO t2 VALUES ("+(i+1)+","+(i+1)+")");
int input1 = in.nextInt();
ResultSet result1 = con.createStatement().executeQuery("SELECT idt1
                                                          FROM t1
                                                          WHERE fieldt1="+i);
result1.next();
con.createStatement().execute("DELETE FROM t2 WHERE idt2="+input1);
con.createStatement().execute("UPDATE t2
                                SET fieldt2 = 1+fieldt2
                                WHERE idt2 < "+(2+result1.getInt("idt1")));
input1 = in.nextInt();
con.createStatement().execute("DELETE FROM t1 WHERE idt1="+input1); }
```

Then, the measurement was repeated for a similar path, but where the block of SQL statements was executed twice in turn: a first time normally and a second time with a value of i increased by 2. The statements of the second round were executed directly on the database resulting from the first round so that they can modify the rows it contains as well. The process was then repeated with four rounds and so on. The obtained measurements are reported in figure 2 and 3. These graphs show the number of constraints and the constraint solving time, as a function of the number of SQL statements executed in the path. The number of constraints evolves linearly with the number of SQL statements.

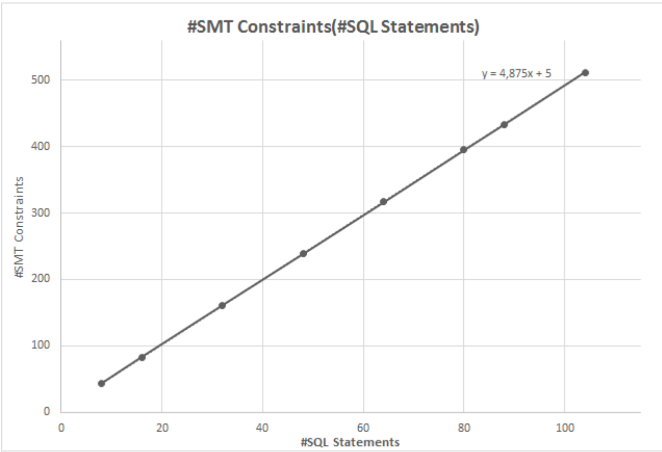

Figure 2: Constraint number as a function of the number of SQL statements in the path

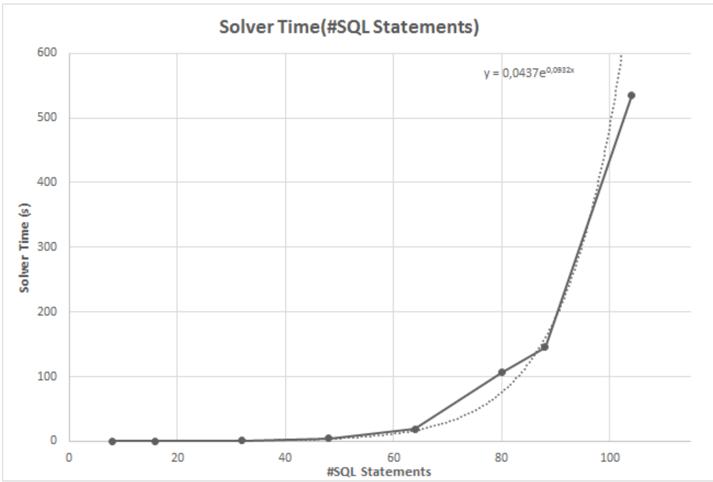

Figure 3: Solving time as a function of the number of SQL statements in the path

The constraint solving time increases exponentially with the number of SQL statements. The solving time starts to rise around 60 statements and becomes important above 100 statements, on our Intel Core i5 at 1.8GHz.

4.4. Benchmarking relational symbolic execution against normalisation

Our last objective with the experiments was to compare the performance of relational symbolic execution with the SynDB tool proposed in [23]. In this tool, the tested code is preprocessed to translate the SQL code into normalised native program code, before applying a classical symbolic execution process. In order to perform this comparison, we have selected four simple Java methods from the testbed used by [23] and used our tool on them. These methods are those of the third group from our testbed.

Our tool generated tests with the same statement coverage as provided by [23]. In order to propose a fair comparison, the total generation time for each method was measured on a Pentium 4 configuration (3.06GHz, 1GB RAM, Microsoft Windows XP 32-bit), similar (but admittedly not identical) to the one used in [23]. The obtained results are synthesised in the first part of table 8. Results for the normalisation-based tool are extracted from [23]. While the results are of course not fully conclusive as both tests were run on different (but comparable) machines, we can nevertheless observe our tool to be between one and two orders of magnitude faster than [23], generating in each case tests in a few seconds compared to more than one minute and a half.

A first reason for such a performance difference between both tools is that [23] normalises the SQL code into native code before trying to generate tests. This requires time and significantly increases the number of paths to be explored, as each SQL statement is translated into native code that introduces new branches and cycles in the control-flow graph. Secondly, [23] is built upon the Microsoft Pex tool for constraint-based testing, which makes use of a dynamic path exploration, requiring to effectively run the code for each generated test case. Our research prototype tool makes use of a static path exploration and does not need to run the code, which also has an impact on our performance comparison. However, this impact is limited by the fact that the methods are very small, involving between 7 and 11 simple statements, with only one simple SQL (SELECT) statement, one loop and no conditional per method. Finally, it should be noted that our tool handles the string attributes appearing in the methods as integer ones. This is possible as the tested methods do not use string-specific operators.

4.5. Additional experiments over open-source code

UnixUsage and RiskIt provide a useful source of open-source code for evaluating our testing tool. As a consequence, we have also extracted some methods involving not only SELECT statements, as in [23], but also SQL statements writing into the database, in order to assess our tool. These methods are those of the fourth group from out testbed.

Test data generation for these methods required extending our symbolic execution algorithm to handle some new parts of Java and SQL, used in RiskIt, or to simulate them the currently handled sublanguage. In particular, the management of tables with no primary key or with multiple-attribute primary key was integrated in the algorithm and string management operations were simulated using either integers or lists of integers. As the second part of table 8 shows, correct test data were generated in just a few seconds and, moreover, manual reviewing of the automatically generated test data enabled detecting a possible fault in the code of RiskIt, where a runtime error is thrown when the method `createNewUser` is called on inputs where the inserted job does not reference any existing occupation or industry.

Table 8: Analysis time for the methods of the third and fourth groups

Name	**Analysis via SynDB [23]**		**Analysis via Relational Symbolic Execution**	
	Covered Statements	**Analysis Time**	**Covered Statements**	**Analysis Time**
SAMPLES EXTRACTED FROM OPEN-SOURCE SOFTWARE				
- *UnixUsage*				
courseNameExists	7	1m 41s	7	4s 984ms
getCommandsByCategory	10	1m 30s	10	1s 454ms
getCourseIDByName	10	1m 33s	10	2s 891ms
getDepartmentIDByName	11	1m 43s	11	1s 578ms
- *RiskIt*				
createNewUser	-	-	91	11s 294ms
deleteUsers	-	-	55	5s 39ms

5. Synthesis of research contributions

In this work, we proposed an approach for enabling the direct symbolic execution of SQL code into constraints. This is a non-trivial extension of traditional symbolic execution because of the complex structure of relational databases and the complex behaviour of SQL statements. Given a code unit in a database program, mixing traditional code with SQL statements, each database table manipulated by the unit is modelled as a variable typed as a relation and each SQL statement as a relational operation over both these relational variables and the traditional variables of the unit. A classical symbolic execution process can then be applied to produce path-constraints, mixing relational and classical constraints over symbols representing the values of both the classical and relational inputs of the unit. These path-constraints can be unified with the data integrity constraints from the database schema. Any solution to the resulting constraint system for a path describes input data for the code unit, including a valid database content, with respect to which the code can be executed and is guaranteed to follow the same path for which the constraints were generated.

A symbolic execution algorithm based on this principle has been completely detailed for a precise subset of Java and SQL. This language enables writing Java methods that use SQL statements and transactions to read and write in a relational database; the latter typically subject to data integrity constraints. The algorithm has been designed with the aim of testing rather small methods, having simple operations and not involving complex computation, as they are typically written in programs acting in an OLTP context. Given the schema of the database, the code of the method and an execution path in this method, the algorithm performs the symbolic execution of the path and produces the corresponding constraints in the standard SMT-Lib language.

The algorithm has been implemented into a testing tool and used to generate test data for a number of methods, including some open-source code. The generated constraints have been solved using the Microsoft Z3 solver. The experiments show that the technique is able to generate test data for all the considered methods, seamlessly and in reasonable time. In particular, the results dramatically improve the scope of the approach, compared to the strategy based on the Alloy solver, proposed in our previous work [25, 26]. These results provide both an experimental confirmation and new elements to the research presented in [28], where Z3 was used to prove the unsatisfiability of Alloy constraints. Our experiments can indeed be seen as some kind of a case study for [28], which confirms the conclusions of [28] about proving unsatisfiable instances. Moreover, our results show that versions of Z3 more recent than the one used in [28], including new model-based quantifier instantiation techniques, can outperform the Alloy solver also in model finding for satisfiable instances. Finally, our measurements showed that the approach may face scalability issues outside the context of methods implementing OLTP use cases, if it is used as such over pieces of code whose typical execution scenarios involve the processing of a large number of SQL statements.

6. Related work

6.1. SQL normalisation into native code

An early work that has considered test data generation for programs interacting with a relational database is [22]. The paper suggests to transform the tested code unit by inserting new classical variables representing the database structure, and translating all SQL statements into native code acting on these variables. A translation to C++ is provided for some relational operators and for some other SQL mechanisms, like row sorting. Classical white-box testing approaches can then be applied to the normalised code unit. A conceptually similar but entirely automated technique is proposed in [23], where the off-the-shelf Microsoft Pex tool, based on symbolic execution and a dynamic path exploration process, is applied to the normalised version of code units from database programs written in C#. This latter approach is validated over 39 Java methods (translated to C# using an automated compiler) extracted from UnixUsage and RiskIt. These samples involve 32 LOCs per method on average, with a maximum of 108 LOCs, and each method mixes Java statements with one or a few SELECT SQL statements.

Conceptually, normalising SQL code into native code and then applying classical symbolic execution on the result is an alternate approach to ours, where the SQL code is directly compiled into relational constraints during symbolic execution. Replacing a single SQL statement by a piece of native code, simulating its execution by a DBMS, is time-consuming and may significantly increase the number of paths to be explored, compared to the original code [22]. In contrast, in relational symbolic execution, the code must not be preprocessed and the execution paths to be considered are limited to paths in the *original* code. Our experimental benchmarks, comparing the SynDB tool from [23], based on normalisation, with our tool, tend to indicate that a direct translation of SQL into constraints provokes a strong performance improvement. Moreover, even if the symbolic execution of INSERT, UPDATE and DELETE statements is conceptually possible using normalisation, [23] only validates their approach over code containing SELECT statements. Relational symbolic execution has on the contrary be experimented in the presence of SQL code writing into the database, as well as in the presence of SQL transactions.

On the other hand, the SynDB tool proposed in [23] also provides some important features that are not handled by ours.

First, SynDB relies on Microsoft Pex, as an off-the-shelf back-end symbolic execution tool. Pex handles natively a large part of the C# language, where our tool is restricted to a small part of Java. Moreover, Pex provides some support for character strings, which constitute an important datatype in SQL, not handled by our tool. Thirdly, Pex uses a dynamic path exploration process (coupled with heuristics for a smarter covering of large control-flow graphs), making concrete values for the program variables available if necessary. This notably gives a direct access to the concrete SQL code that can be, in some programs, crafted dynamically as a character string, to be parsed and processed by the DBMS. As a consequence, the tool from [23] can account, at least to some extent, for such dynamically crafted SQL statements, where our tool only considers SQL statements whose structure is completely defined statically.

Secondly, SynDB deals with more complex SELECT queries than our tool, involving cross joins as well as aggregate functions. At the same time, it proposes to handle SQL queries involving nested sub-queries by unnesting them into equivalent simple queries. Our tool could also preprocess the tested code to unnest queries into simpler ones, using the same translation algorithms as in SynDB. Constraints would then be derived directly from the unnested queries. The constraint generation rules used in our tool could be generalised to handle additional SQL constructs like cross joins and aggregate functions, but this has currently not been implemented. A cross join between tables could be represented by an SMT-Lib predicate over a record type, gathering together the corresponding joined sorts. Aggregate functions could be represented by (appropriately constrained) uninterpreted functions. As a brief and preliminary experimentation of the efficiency of such generalised rules, we have generated constraints for the following piece of code, acting over the same library database as in Figure 1:

```
ResultSet maxCodeShelf = con.createStatement().executeQuery(
                                                    "SELECT max(code) AS maxCode
                                                     FROM shelf,book
                                                     WHERE shelfId = id
                                                     GROUP BY shlefId");
if (maxCodeShelf.next() & maxCodeShelf.getInt("maxCode") == 1)
    ...
```

The code contains one SELECT query involving a cross join and an aggregate function. The constraints were appropriately generated for the path taking the then branch of the if statement and solved in about 10ms by Z3.

6.2. SQL queries as independent input relations

[24] is, to our best knowledge, the very first work to have proposed a convincing approach, based on symbolic execution, for testing database programs. In that work, after defining the formal syntax and semantics of a study language for writing code units in database programs, a test generation algorithm is developed for this language. The principle is to consider the result of each (simple) SQL SELECT query executed in the code as an independent input of the tested unit, typed as a relation. The size of each of these input relations, as well as the content of each of their cells, is transparently accessed by the code in the unit. As a consequence, the test generation algorithm will produce path constraints involving constraints over these sizes and contents. The algorithm will also add additional constraints to the path constraints, enforcing that each row in an input relation makes the condition of the WHERE clause from its corresponding SELECT true. The resulting path constraints are solved using a sound but approximate ad-hoc procedure, involving a solver for strings, which is described in the paper. Following this approach, the JCUTE symbolic execution tool for Java is provided with the ability to generate database inputs in the presence of simple SELECT queries in the tested code. Experimentation is discussed for a 16 lines of code Java method, involving one SQL query and extracted from open-source software.

The tool detailed in [24] has important features that are not present in our tool. First, and contrary to our tool, it deals with null values and, at least partly, with character strings. Secondly, it relies on the JCute symbolic execution tool. JCute handles a large part of the Java language, including multi-threading, and uses a dynamic path exploration process. As with Pex, this makes the tool from [24] able to benefit from concretisation as well as to account, at least to some extent, for dynamically crafted SQL statements.

However, [24] also suffers from several conceptual limitations compared to relational symbolic execution. Firstly, the generated path constraints do not describe the actual content of the database during code execution, like in relational symbolic execution, but only the results of the SQL queries executed by this code, as if they were non-interdependent code inputs. As a consequence, the algorithm may generate incomplete path constraints, missing the fact that the result of a query in a path may depend on the result of a query executed earlier in this path, as both queries are executed over a single database. Secondly, [24] only provides a conceptual intuition but no proper algorithm component for enforcing the data integrity constraints defined in the database schema. [23] advocates that such a lack leads to the possible generation of invalid test data and to a poor code coverage. Finally, [24] provides no procedure, nor any conceptual intuition, about how the proposed algorithm could practically handle the presence of data modification or transaction management primitives in the tested code, which are both crucial components of database programs.

6.3. Other related work

As a part of a multi-step approach for generating servlet call sequences to test web applications, [43] briefly suggests to execute symbolically SQL queries into relational constraints written in Alloy. The paper proposes to transform SQL queries into a relational algebra, and provides an example that encodes one relational algebra query into Alloy constraints. While the relational symbolic execution proposed in [43] is conceptually similar to ours, it does not account for the symbolic execution of data modification statements with constraint integrity verification, nor for transaction primitive handling. Moreover, no formal systemization of the constraint generation process, nor any evaluation of this process, is proposed. Finally, the target language for the generated constraints is the Alloy framework, whose poor adequacy for this particular problem has been demonstrated in this paper.

On a related but complementary level, a substantial amount of work (e.g. [44, 45, 46, 47, 33, 48, 49, 50, 51, 52, 20, 53, 54, 55, 56]) has been done on how to generate test database content exhibiting some desirable properties, given only the database schema and possibly some queries to be executed over the database. The main difference between our work and these approaches is that they essentially work without considering the control flow of the programs manipulating the database.

Microsoft Qex [52, 20] is probably the one of these techniques which is the closest to our work, as both approaches are based on the translation of the SQL semantics into SMT-Lib constraints, solved using Z3. However, the two approaches also exhibit important differences. Firstly, Qex only considers input generation for a single SQL SELECT query in isolation and not for pieces of code involving several queries and modification statements, but the handled queries can be more complex than in our tool. Secondly, the two techniques translate the SQL semantics into constraints differently: our tool use predicates and quantifiers to represent relations, while Qex use fixed-size lists of tuples. Qex involves thus a mechanism similar to Alloy, where the solving process must be repeated on increasing size for the input relations, in order to find a solution. Such a mechanism may not be problematic for testing OLTP code units, touching only a limited number of tuples in the database. However, this approach does not make it possible to prove formally that a path or path prefix is infeasible, which is done efficiently by our tool.

Some database programs are developed to work with data already stored in an existing database. Some papers [57, 58] study the particular situation where classical test inputs must be generated for code manipulating in parallel an existing database with a known input content.

Other work [59, 60, 61] considers *mutation testing* of database programs, where our approach performs *structural testing*. In mutation testing, the quality of the test data is no more measured in terms of code coverage, as in structural testing, but in terms of program fault detecting ability (see [4] for a discussion). Some works have also focused over testing of non-functional aspects of database programs, like security testing [62].

7. Threats to validity

7.1. Internal validity

The performance comparison between our relational symbolic execution tool and the SQL normalisation tool from [23] revealed a very strong advantage for relational symbolic execution, over similar hardware configurations. However, the comparison was performed on a limited number of small samples and considering tools with different architectures. We think that the impact of the last element is minor, given the small size of the tested methods, and cannot explain, on its own, how important the measured performance gap is. However, this element, as well as the lack of a comparison over a large number of various complex pieces of code, threaten our ability to conclude, from our experimental work, that a direct translation of SQL into relational constraints intrinsically improves the testing time, compared to compiling SQL into native code before generating constraints.

7.2. External validity

The tool we have developed handles a limited subset of the Java/SQL syntax and was developed in the context of transactional business use cases, i.e. small pieces of code acting on a limited number of rows in the database. Several elements threaten the generalisability of our approach to handle any piece of code involving SQL.

Conceptually, our approach can be integrated into any existing symbolic execution tool, to provide this tool with the ability to handle SQL code. An integration into a state-of-art tool, able to handle not all, but a wide range of programs in a mainstream programming language, would strongly improve the practical scope of our technique. However, such an integration needs to be evaluated in practice.

The handling of a larger part of SQL may require the generation of constraints which could be hard or impossible to manage by solver technology. Moreover, as SQL is a large and complex language, fragmented into several dialects, developing an universal SQL symbolic execution engine, handling any piece of SQL code, will be a very hard task.

Our tool provides no automated way to handle SQL code that has been dynamically crafted by the program, which can be frequent in practice. Alternate tools have shown that we could exploit the concretisation mechanism from a state-of-art symbolic execution tool to alleviate the problem, thanks to the availability of the concrete values for the dynamically crafted SQL code. However, such an approach fails handling cases where the syntactic structure of the crafted SQL depends on the inputs of the tested code.

Finally, current solver technologies have shown to face a scalability issue when required to solve the constraints generated by our tool for a sufficiently large number of SQL statements. Whether this problem can be solved or at least alleviated, by optimising the constraint generation or by new solving capabilities, remains an open research question.

8. Future work

For further work, we have first identified three main research directions. These directions sketch a path towards a broader evaluation of the approach over many various real pieces of code, in an industrial context:

Integration with Pex Our tool has demonstrated how a classical symbolic execution mechanism for a typical programming language could be empowered with the ability to generate constraints in the presence of SQL code. As this integration provided promising results, it should now be repeated with a state-of-art tool based on symbolic execution. Microsoft Pex [18] is a particularly appropriate candidate for integration with our work, as it is based as well on the Z3 solver. In addition, such an integration could enable a deeper comparison with the SynDB tool from [23], which is also based on Pex. The Pex tool is not open-source but provides an extension interface. To the best of our knowledge, available open-source constraint-based testing tools are KLEE [14] (used as a core component of the S2E platform [63]) and CREST [64] (for C code) as well as Symbolic Path Finder [17] (for Java byte-code).

Dynamic SQL Handling By integrating our approach in a tool like Pex, the resulting tool would benefit for free from a dynamic path exploration process coupled with heuristics to handle large number of paths. In presence of SQL code that has been built dynamically as a character string in the tested code, the use of such a dynamic path exploration would make the concrete values of the assembled string elements directly available. Like in alternate approaches to ours, this runtime information should be used to recover the complete structure of the executed SQL code, making us able to translate it into constraints.

However, such an approach will fail if the code unit's inputs are used as parts of the syntactic structure of the dynamically-crafted SQL code. Nevertheless, by choosing appropriate concrete values for those parts of the inputs defining the syntactic structure of the dynamic SQL, we could produce representative specialised versions of the original code unit, which could be properly evaluated symbolically. Interleaving symbolic execution with such a partial evaluation [65] has already been studied in another context by [66]. Detecting which parts of the code unit's inputs should be made concrete could benefit from existing work (e.g. [67, 68]) over detection of SQL injections attacks.

Wider SQL Handling In the perspective of testing programs involving many complex SQL statements, a tighter integration with constraint solving techniques would be beneficial to offer a better scalability and a larger scope to the approach. The constraint generator should be tailored so to generate constraints optimised for the internal algorithms of the solver. Conversely, the development of solving algorithms or heuristics tailored to the kind of constraints produced by the symbolic execution of large pieces of complex SQL code should also be considered.

In particular, SQL enables various operations to be performed over data belonging to various datatypes, such as strings, binary objects, numeric values and timestamps. Symbolic execution of such operations will produce complex constraints over such datatypes. If modern SMT solvers like Z3 can already handle many of these constraints, work should be done to locate the common parts of SQL which will put current solvers into trouble and research should be performed to possibly build a workaround. Solver development is a particularly dynamic research domain. Notably, research is ongoing (e.g. [69, 70]) towards a proper solving of string contraints inter-related with other kinds of constraints, in the context of symbolic execution. Similarly, recent works (see e.g. [71]) have considered (partial) solving of non-linear integer arithmetic constraints. Other works have also studied the particular problem of multi-granularity temporal constraint solving (see e.g. [72]).

However, building optimal constraint generation rules for the whole of SQL is made difficult by the fact that SQL is large and complex, and can vary strongly in practice between different DBMS's versions and manufacturers. A research direction for overcoming these difficulties would be to use a relational algebra as an intermediate language for constraint generation, similarly to the approach proposed in [43]. The symbolic execution engine would compile the original SQL code into a minimal relational algebra, and then the algebraic operators would be translated into logical constraints, using a minimal set of translation rules optimised for the solver. Algorithms translating SQL statements into equivalent combinations of a core set of relational algebra's operators are well-known, in the context of query processing in DBMS design [73]. In practice, this idea should be refined, as SQL allows non-relational constructs like rows ordering and aggregation, null values etc. The intermediate language should thus be extended by a minimal set of operators for describing common and tractable non-relational parts of SQL. Concretisation could be the last-ditch solution to handle exotic or too complex parts of SQL.

Programs to be tested which manipulate an existing database are common in practice. Whether and how our technique could select test data from an existing database, instead of generating them from scratch is an interesting matter of further research.

Constraints typically admit many different solutions. However, our tool uses the arbitrary solution returned as first by the solver. A possible improvement could be to use an optimal solution according to some criterion, like for example, a minimal number of rows in the database. In order to do so, it has recently been announced[6] that Z3 could be provided with the ability to return the solution which optimises a given objective function. Evaluating and integrating this mechanism with our approach is a topic for future research.

Finally, being somehow parametrised with respect to the paths that should be considered, our approach can be used with respect to any traditional code coverage criterion based on the notion of execution path [4]. Nevertheless, several works [74, 75, 76, 77, 60] propose coverage criteria particularly tailored towards testing of database programs. Integrating such criteria into our constraint-based approach is a topic of ongoing research.

Acknowledgment

This work has been funded by the Belgian Fund for Scientific Research (F.R.S.-FNRS). The authors would like to thank P.-Y. Schobbens, V. Englebert and G. Perrouin for useful discussions, C. Aerts for test hardware availability and the anonymous referees for their constructive remarks both on current and earlier versions of this paper.

[6]see "Objective Functions in Z3", Nikolaj Bjorner, keynote at CSTVA 2014, Hyderabad, India.

Bibliography

References

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

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

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

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

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

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

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

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

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

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

[11] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, SIGPLAN Not. 40 (6) (2005) 213–223.

[12] B. Dutertre, L. de Moura, A fast linear-arithmetic solver for DPLL(T), Proceedings of the 18th International Conference on Computer Aided Verification (2006) 81–94.

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

[14] C. Cadar, D. Dunbar, D. Engler, KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs, Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (2008) 209–224.

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

[16] K. Sen, G. Agha, CUTE and jCUTE: concolic unit testing and explicit path model-checking tools, in: Proceedings of the 18th international conference on Computer Aided Verification, CAV'06, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 419–423.

[17] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, M. Pape, Combining unit-level symbolic execution and system-level concrete execution for testing NASA software, in: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA '08, ACM, New York, NY, USA, 2008, pp. 15–26.

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

[19] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.

[20] M. Veanes, N. Tillmann, J. de Halleux, Qex: Symbolic SQL query explorer, Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (2010) 425–446.

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

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

[23] K. Pan, X. Wu, T. Xie, Guided test generation for database applications via synthesized database interactions, ACM Trans. Softw. Eng. Methodol. 23 (2) (2014) 12:1–12:27.

[24] M. Emmi, R. Majumdar, K. Sen, Dynamic test input generation for database applications, in: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA, ACM, New York, NY, USA, 2007, pp. 151–162.
[25] M. Marcozzi, W. Vanhoof, J.-L. Hainaut, Test input generation for database programs using relational constraints, in: Proceedings of the Fifth International Workshop on Testing Database Systems, DBTest '12, ACM, New York, NY, USA, 2012, pp. 6:1–6:6.
[26] M. Marcozzi, W. Vanhoof, J.-L. Hainaut, A relational symbolic execution algorithm for constraint-based testing of database programs, in: Source Code Analysis and Manipulation (SCAM), IEEE 13th International Working Conference on, ACM, 2013, pp. 179–188.
[27] D. Jackson, Software Abstractions: Logic, Language, and Analysis, The MIT Press, 2006.
[28] A. A. El Ghazi, M. Taghdiri, Relational reasoning via SMT solving, in: Proceedings of the 17th international conference on Formal methods, FM'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 133–148.
[29] C. Barrett, A. Stump, C. Tinelli, The SMT-LIB Standard: Version 2.0, in: A. Gupta, D. Kroening (Eds.), Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK), 2010.
[30] L. De Moura, N. Bjørner, Z3: An efficient SMT solver, in: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 337–340.
[31] Y. Ge, L. Moura, Complete instantiation for quantified formulas in satisfiability modulo theories, in: Proceedings of the 21st International Conference on Computer Aided Verification, CAV '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 306–320.
[32] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, 1st Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
[33] C. Binnig, D. Kossmann, E. Lo, Multi-RQP: Generating test databases for the functional testing of OLTP applications, Proceedings of the 1st International Workshop on Testing Database Systems (2008) 5:1–5:6.
[34] Oracle, JDBC overview: `http://www.oracle.com/technetwork/java/overview-141217.html`, Website (3 2014).
[35] ISO, ISO/IEC 9075 : 2011 information technology, database languages, SQL 9075 : 2011 information technology, database languages, SQL (2011).
[36] J. Edvardsson, A survey on automatic test data generation, in: Proceedings of the Second Conference on Computer Science and Engineering in Linköping, ECSEL, 1999, pp. 21–28.
[37] K. Sen, D. Marinov, G. Agha, Cute: a concolic unit testing engine for C, SIGSOFT Softw. Eng. Notes 30 (5) (2005) 263–272.
[38] C. Cadar, D. Engler, Execution generated test cases: How to make systems code crash itself, in: Proceedings of the 12th International Conference on Model Checking Software, SPIN'05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 2–23.
[39] P. Godefroid, Higher-order test generation, SIGPLAN Not. 46 (6) (2011) 258–269.
[40] L. Moura, N. Bjørner, Efficient e-matching for SMT solvers, in: Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction, CADE-21, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 183–198.
[41] C. Barrett, A. Stump, C. Tinelli, The SMT-LIB Standard: Version 2.0, in: A. Gupta, D. Kroening (Eds.), Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK), SMT-Lib, 2010.
[42] W. E. Howden, Methodology for the generation of program test data, Computers, IEEE Transactions on 100 (5) (1975) 554–560.
[43] X. Fu, Relational constraint driven test case synthesis for web applications, in: G. Salaún, X. Fu, S. Hallé (Eds.), Proceedings Fourth International Workshop on Testing, Analysis and Verification of Web Software, Antwerp, Belgium, 21 September 2010, Vol. 35 of Electronic Proceedings in Theoretical Computer Science, Open Publishing Association, 2010, pp. 39–50.
[44] N. R. Lyons, An automatic data generating system for data base simulation and testing, SIGMIS Database 8 (4) (1977) 10–13.
[45] M. Robbert, F. J. Maryanski, Automated test plan generator for database applications systems, SIGSMALL/PC Notes 17 (3-4) (1991) 29–35.
[46] C. Binnig, D. Kossmann, E. Lo, Reverse query processing, in: Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on, IEEE Computer Society Press, 2007, pp. 506–515.
[47] C. Binnig, D. Kossmann, E. Lo, Testing database applications, in: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD '06, ACM, New York, NY,

USA, 2006, pp. 739–741.
[48] J. Zhang, C. Xu, S. C. Cheung, Automatic generation of database instances for white-box testing, in: Proceedings of the 25th International Computer Software and Applications Conference, COMPSAC '01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 161–165.
[49] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, E. J. Weyuker, An agenda for testing relational database applications: Research articles, Softw. Test. Verif. Reliab. 14 (1) (2004) 17–44.
[50] Y. Deng, P. Frankl, D. Chays, Testing database transactions with agenda, in: Proceedings of the 27th International Conference on Software Engineering, ICSE '05, ACM, New York, NY, USA, 2005, pp. 78–87.
[51] D. Willmor, S. M. Embury, An intensional approach to the specification of test cases for database applications, in: Proceedings of the 28th international conference on Software engineering, ICSE '06, ACM, New York, NY, USA, 2006, pp. 102–111.
[52] M. Veanes, P. Grigorenko, P. Halleux, N. Tillmann, Symbolic query exploration, in: Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering, ICFEM '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 49–68.
[53] C. de la Riva, M. J. Suarez-Cabal, J. Tuya, Constraint-based test database generation for SQL queries, in: Proceedings of the 5th Workshop on Automation of Software Test, AST '10, ACM, New York, NY, USA, 2010, pp. 67–74.
[54] J. Tuya, M. J. Suarez-Cabal, C. de la Riva, Full predicate coverage for testing SQL database queries, Softw. Test. Verif. Reliab. 20 (3) (2010) 237–288.
[55] C. Binnig, D. Kossmann, E. Lo, M. T. Özsu, QAGen: generating query-aware test databases, in: C. Y. Chan, B. C. Ooi, A. Zhou (Eds.), SIGMOD Conference, ACM, 2007, pp. 341–352.
[56] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, S. Khurshid, Query-aware test generation using a relational constraint solver, in: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 238–247.
[57] C. Li, C. Csallner, Dynamic symbolic database application testing, in: Proceedings of the Third International Workshop on Testing Database Systems, DBTest '10, ACM, New York, NY, USA, 2010, pp. 7:1–7:6.
[58] K. Pan, X. Wu, T. Xie, Generating program inputs for database application testing, in: Proc. 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society Press, 2011.
[59] J. Tuya, M. J. Suarez-Cabal, C. de la Riva, SQLmutation: A tool to generate mutants of SQL database queries, in: Proceedings of the Second Workshop on Mutation Analysis, MUTATION '06, IEEE Computer Society, Washington, DC, USA, 2006, p. 1.
[60] C. Zhou, P. Frankl, Mutation testing for Java database applications, in: Proceedings of the International Conference on Software Testing Verification and Validation, ICST '09, IEEE Comp. Soc., Washington, DC, USA, 2009, pp. 396–405.
[61] K. Pan, X. Wu, T. Xie, Automatic test generation for mutation testing on database applications, in: Automation of Software Test (AST), 8th International Workshop on, IEEE Computer Society Press, 2013, pp. 111–117.
[62] D. Appelt, C. D. Nguyen, L. C. Briand, N. Alshahwan, Automated testing for SQL injection vulnerabilities: An input mutation approach, in: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2014, ACM, New York, NY, USA, 2014, pp. 259–269.
[63] V. Chipounov, V. Kuznetsov, G. Candea, The s2e platform: Design, implementation, and applications, ACM Trans. Comput. Syst. 30 (1) (2012) 2:1–2:49.
[64] J. Burnim, K. Sen, Heuristics for scalable dynamic test generation, in: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 443–446.
[65] N. D. Jones, C. K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
[66] R. Bubel, R. Hähnle, R. Ji, Interleaving symbolic execution and partial evaluation, in: Proceedings of the 8th International Conference on Formal Methods for Components and Objects, FMCO'09, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 125–146.
[67] W. G. J. Halfond, A. Orso, P. Manolios, Using positive tainting and syntax-aware evaluation to counter SQL injection attacks, in: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, ACM, New York, NY, USA, 2006, pp. 175–185.
[68] Z. Su, G. Wassermann, The essence of command injection attacks in web applications, SIGPLAN

Not. 41 (1) (2006) 372–382.
[69] Y. Zheng, X. Zhang, V. Ganesh, Z3-str: A Z3-based string solver for web application analysis, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, ACM, New York, NY, USA, 2013, pp. 114–124.
[70] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, M. Deters, A DPLL(T) theory solver for a theory of strings and regular expressions, in: A. Biere, R. Bloem (Eds.), CAV, Vol. 8559 of Lecture Notes in Computer Science, Springer, 2014, pp. 646–662.
[71] D. Jovanovic, L. M. de Moura, Solving non-linear arithmetic (2012) 339–354.
[72] C. Bettini, X. Wang, S. Jajodia, Solving multi-granularity temporal constraint networks, Artificial Intelligence 140 (1–2) (2002) 107 – 152.
[73] T. Connolly, C. Begg, Database Systems: A Practical Approach to Design, Implementation, and Management, no. vol. 1 in International computer science series, Addison-Wesley, 2005.
[74] W. G. J. Halfond, A. Orso, Command-form coverage for testing database applications, in: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 69–80.
[75] G. M. Kapfhammer, M. L. Soffa, A family of test adequacy criteria for database-driven applications, in: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-11, ACM, New York, NY, USA, 2003, pp. 98–107.
[76] M. J. Suárez-Cabal, J. Tuya, Using an SQL coverage measurement for testing database applications, in: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '04/FSE-12, ACM, New York, NY, USA, 2004, pp. 253–262.
[77] M. J. Suarez-Cabal, J. Tuya, Structural coverage criteria for testing SQL queries, Journal of Universal Computer Science 15 (3) (2009) 584–619.