Towards Testing of Full-Scale SQL Applications using Relational Symbolic Execution

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

Published in:
Proceedings of 36th International Conference on Software Engineering (ICSE 2014) Workshops: 6th Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA 2014)

Publication date:
2014

Document Version
Peer reviewed version

Towards Testing of Full-Scale SQL Applications using Relational Symbolic Execution

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

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

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

ABSTRACT

Constraint-based testing is an automatic test case generation approach where the tested application is transformed into constraints whose solutions are adequate test data. In previous work, we have shown that this technique is particularly well-suited for testing SQL applications, as the semantics of SQL can be naturally transformed into standard SMT constraints, using so-called relational symbolic execution. In particular, we have demonstrated such testing to be possible in practice with current solver techniques for small-scale applications. In this work, we identify the main challenges and provide research directions towards constraint-based testing of full-scale SQL applications. We investigate the additional research work needed to integrate relational and dynamic symbolic execution, handle properly dynamic SQL, generate tractable SMT constraints for most SQL applications, detect SQL runtime errors and deal with non-deterministic SQL.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution, Testing tools (e.g., data generators, coverage testing)*; F.4.1 [**Mathematical Logic and formal languages**]: Mathematical Logic—*Logic and constraint programming*; H.2.3 [**Database Management**]: Languages—*Query languages*

General Terms

Verification

Keywords

Test data generation, Fault localization, Symbolic execution, SMT solvers, Quantifiers, Databases, SQL

*F.R.S.-FNRS Research Fellow

CSTVA '14, May 31, 2014, Hyderabad, India

1. INTRODUCTION

Symbolic execution [25] is a program analysis that basically executes the program's code over symbolic values instead of concrete ones. This technique has been advocated as an efficient software testing approach [7, 8], as it can be used to provide the tester with a representative set of test inputs (and outputs) for a code under test. Such a testing approach has recently gained a renewed interest, because of the rise of dynamic symbolic execution [18] and of important advances in constraint solving, namely using SMT solvers [14].

Among recent research in the field, symbolic execution has been investigated (e.g. [27, 28, 29, 32, 16, 38, 26, 30, 31]) for testing SQL applications, i.e. programs that mix code with SQL operations to interact with a relational database. Dealing with SQL is a non-trivial extension of known symbolic execution techniques, notably because of the complex structure of relational databases and the complex behavior of SQL statements [29]. In our recent work [27, 28, 29], we have proposed *relational symbolic execution* as an adequate mean to overcome these difficulties. This approach was successfully experimented [29] over a set of *small-scale* SQL applications.

In this paper, we identify the main challenges and provide research directions towards testing of *full-scale* SQL applications, using relational symbolic execution. Section 2 details the relational symbolic execution approach. Section 3 introduces a sample SQL application to test and presents research directions to scale the approach, by discussing the elements in the sample application that our approach cannot currently handle. Finally, section 4 provides some concluding remarks.

2. RELATIONAL TESTING OF SQL APPS

The symbolic execution of an application means processing the application's statements over symbolic values, instead of concrete ones. For each control dependency met in the code, the symbolic execution can proceed along any of the possible paths and generate constraints upon the symbolic values such that when the program's variables have concrete values satisfying these constraints, the real execution would proceed along the selected path. For each path covering a full execution of the application, the constraints collected along can be regrouped in a so-called *path-constraint*. In the context of testing an application, one can use symbolic execution to generate the path-constraints for a set of paths in the code

that satisfy a given coverage criterion [43]. Solving these path-constraints then provides the tester with a representative set of concrete test values for the considered application.

By introducing relational symbolic execution [27, 28, 29], we have extended classical symbolic execution to generate test data for SQL applications. SQL applications are programs that mix classical code with SQL code to read and write, using SQL transactions, into a relational database, subject to data integrity constraints. We have defined a core subset of the Java and SQL languages for writing such SQL applications and written a relational symbolic evaluator [29] for this language, i.e. a set of formal rules to generate the corresponding constraints during the symbolic execution of any application in the language. This evaluator is relational as it is based on a relational view of the tested application. In this view, every table in the database is seen as an application's variable typed as a mathematical relation. Similarly, each SQL statement is seen as a relational operation over these relational variables and the traditional application's variables. The behavior of such relational operations can then be naturally described by SMT constraints [2] mixing quantifiers with uninterpreted function and array theories (AUFLIA logic), as explained in [15]. These behavioral constraints can be combined with constraints enforcing that every relational variable satisfies the database's integrity constraints (like the primary key, foreign key or check constraints), as these last constraints are typically defined in the same constraint logic [12]. This approach was experimented [29] over ten thousands paths in eighteen SQL applications, for a total of five hundred lines of code, including eighty SQL statements. The constraints were successfully and quickly solved using the Z3 solver [13], producing meaningful test data. Alternate approaches are either limited to a reduced part of SQL [16] or they require to transform the SQL code into traditional application code [9, 32], so increasing dramatically the number of paths to be explored [9] and making impossible the detection of unfeasible paths in the original code [29].

In short, our work over relational symbolic execution proves, for a core Java/SQL language, that the full semantics of applications mixing SQL and classical code can be translated into standard SMT constraints, and demonstrates that these constraints can be properly handled by existing solvers, at least for small applications. However, several challenges still need to be overcome to use the approach for more realistic and demanding SQL applications. First, in our current approach, we have always performed relational symbolic execution independently of any concrete execution. Nevertheless, integration with dynamic analysis has been recognized as very beneficial for symbolic execution to work well in practice [7, 8, 18]. Secondly, the approach cannot handle dynamically-crafted SQL code, which is very frequent in practice. Thirdly, general constraint generation rules for the complete SQL language might be difficult to define in practice, because of the complexity and variability of the SQL syntax and semantics. In the the same time, constraint solvers might fall short to solve the constraints generated for the complete SQL semantics over large applications. Fourthly, the approach cannot properly detect SQL runtime errors, which constitute an important marker of faults in SQL applications. Finally, the approach cannot properly handle non-deterministic SQL code, which can be common in practice. The remainder of this paper discusses in details each of these five challenges, which are treated respectively in the five subsections 3.2, 3.3, 3.4, 3.5 and 3.6.

3. ISSUES IN RELATIONAL TESTING

3.1 A Faulty SQL Application to Test

We introduce here the sample faulty SQL application from Figure 1 as a basis for the subsequent discussion over the five main challenges faced by relational symbolic execution. This application describes code as it could have been extracted from library software. It is composed of the Java method *removeBookShelf* and of the SQL DDL code describing the part of the library database touched by the method, i.e. the table *shelf*. In this table, a shelf is described by a technical identifier (an integer), by its label (a string of maximum 50 characters), by the number of books it contains (an integer which must be positive) and by the moment where it is planned to be reordered (which can be *null*). When a book is definitely removed from the library, several Java methods are called in the library's software to update the library's database. Among those, the *removeBookShelf* method is supposed to decrease by one the number of books in the shelf from which the book was removed. If this book was the last one, the shelf is removed from the database. In practice, the *removeBookShelf* method receives the label of the shelf and a connection to the database and retrieves the current number of books in the shelf from the database. If this number is greater than one, it is updated in the database, otherwise the shelf's row is deleted.

Three SQL-related faults are present in this application. First, the method mistakenly supposes that the SELECT query will always return one and only one row. Secondly, the UPDATE statement decreases the number of books by two instead of one. Finally, there is a syntax error in the SQL code of the DELETE statement (doubled WHERE token).

3.2 Static vs Dynamic Symbolic Execution

In our previous work [28, 29], relational symbolic execution was performed independently of any concrete execution. A set of paths in the code was statically selected and symbolically executed to generate test data. The main advantage of such a *static symbolic execution* [25] is that it only requires the database's schema or some parts of it, where a dynamic approach requires to run the code on a functional database. Access to such a database might not be easy at testing time and the repeated calls to the Database Management System (DBMS) will make the testing process slower. Works have tried to alleviate this problem by using a mock database [38].

Nevertheless, *dynamic symbolic execution* (e.g. [19, 6, 35]) has benefited from an important wave of popularity in recent years [7, 8]. [17] formally compares the respective power of static and dynamic symbolic execution. In a nutshell, the dynamic approach is more powerful because it has a natural access to concrete values and can use them to replace the parts of the code that cannot be symbolically executed (e.g. calls to programs whose source is unavailable, or statements that generate undecidable constraints). This last process is called *concretization*. Nevertheless, a similar (and even more powerful) ability can be conceptually integrated, with some practical difficulties, within the static approach, using *higher-order test generation* [17]. As discussed in the next subsections, *concretization* can be beneficial for symbolic execution of SQL, notably to handle some exotic or com-

Figure 1: A faulty SQL application that removes a book from a shelf in a library database.

```
                                                    1  void removeBookShelf (Connection con,int theShelf) throws SQLException {
                                                    2   String condition = " WHERE label ="+theShelf;
CREATE TABLE shelf (                                3   ResultSet rs  = con.createStatement().executeQuery("SELECT numberOfBooks
id INTEGER NOT NULL,                                4                                                      FROM shelf"+condition);
label NVARCHAR2(50) NOT NULL,                       5   rs.next();
numberOfBooks INTEGER NOT NULL,                     6   int booksNumber = rs.getInt("numberOfBooks");
nextReordering TIMESTAMP(2),                        7   if (booksNumber > 1) {
CONSTRAINT sPK                                      8    con.createStatement().execute("UPDATE shelf
                PRIMARY KEY (id),                   9                                   SET numberOfBooks=numberOfBooks-2"
CHECK(numberOfBooks > 0));                         10                                   +condition);
                                                   11   } else {
                                                   12     con.createStatement().execute("DELETE FROM shelf WHERE"+condition);}}
```

plex parts of the language or to make easier the symbolic execution of dynamically-crafted SQL statements. As a consequence, the integration of relational symbolic execution with dynamic symbolic execution and with higher-order test generation should be realized and compared. In the remaining part of this subsection, we briefly show that building a unified relational and dynamic symbolic execution algorithm is conceptually straightforward.

A unified relational and dynamic symbolic execution algorithm would start by populating the database with random but valid content. In order to do so, it could use relational symbolic execution to translate the database's schema into constraints over symbols representing the initial content of the database's tables. Any solution to these constraints would constitute a valid input content for the database. Secondly, the algorithm would run the SQL application by providing it with a connection to the database and with random values for the input parameters. The application's execution would proceed normally, but the code would be instrumented so that the executed statements would be symbolically executed in parallel, thereby generating the corresponding contraints along with the concrete execution, using a relational symbolic evaluator. Once the code completely executed, the input and output values and the database's initial and final contents would be saved as a test case. Then, the algorithm would consider the generated constraints and flip some of them in order to produce constraints that would enforce the execution of an unexplored path. These constraints would be solved to produce new concrete inputs for the database and for the input parameters. The whole process would be repeated with these new inputs. The algorithm would stop when a sufficient [43] number of paths would have been explored.

3.3 Handling Dynamically-Crafted SQL

The relational symbolic evaluator that we proposed in [29] was designed in the context of *static SQL*, whereas the *removeBookShelf* method from Figure 1 uses *dynamic SQL*, making the direct use of the constraint generation rules from [29] impossible for testing this application. Static and dynamic SQL [12] are the two existing interaction paradigms between an application and a DBMS. In static SQL, the syntactic structure of the SQL statement is completely defined statically. The value of the constants in the typed expressions used in this statement can be defined parametrically at compile time as a function of the application's variables. In dynamic SQL, the application builds dynamically a character string containing the SQL statement's code, which is parsed and executed at runtime. Conceptually, symbolic execution is only possible for static SQL, as the symbolic evaluator will always need the syntactic structure of the SQL code, as well as a precise definition of the relation enforced by the SQL code between the application's variables and the database's content, in order to be able to generate constraints. Nevertheless, in practice, the dynamic SQL code can often be normalized, for a given execution path in the code, into an equivalent static SQL code. This normalization is even easier if a possible runtime value for the string variable containing the SQL code is known for the considered path. As a simple illustrating example, if we consider a path in the *removeBookShelf* method that we would like to execute symbolically, the dynamic *SELECT* statement can be replaced by a static SQL statement "**SELECT** numberOfBooks **FROM** shelf **WHERE** label = X" where the value of parameter X is defined as equal to the value of the *theShelf* parameter. The constraint generation rules from [29] can then be applied on this normalized version of the path's code.

However, path normalization of dynamic SQL into static SQL is not always possible, as dynamic SQL allows to use the application's inputs as parts of the syntactic structure of the dynamically-crafted SQL code. In some applications, the whole SQL statement can even be loaded as a string from the database itself. Such cases are particularly problematic, as symbolic execution computes inputs from code analysis and is thus not designed for applications whose code is part of the input. Integration between symbolic execution and partial evaluation [23] could be an interesting research direction for solving this problem. Partial evaluation basically optimizes a piece of code by precomputing statically all the parts of the code that depend on inputs which are known at compile time. In an SQL application, by choosing appropriate concrete values for those parts of the inputs that are used to define the syntactic structure of the dynamic SQL statements, one could use partial evaluation to produce representative specialized versions of the original application that can be properly evaluated symbolically. Interleaving symbolic execution and partial evaluation has already been studied in another context by [5]. Detecting which parts of the application's inputs should be made concrete could benefit from existing work (e.g. [21, 37]) over detection of SQL injections, i.e. well-known malicious code injection techniques exploiting the use of application's inputs in dynamic SQL as an attack vector.

3.4 Building and Solving Constraints for SQL

An important point to notice in the constraints generated using the constraint generation rules that we proposed in [29] is the extensive use of *quantifiers*. These are necessary to capture the full semantics of SQL applied over tables containing relations whose cardinality is unknown and un-

bounded. SQL is indeed essentially syntactic sugar for the operators of relational algebra [11, 12], whose expressiveness has been proven equivalent to a quantified subset of first-order logic, called domain-independent relational calculus [1]. If first-order logic is not decidable in general [10, 40], some SMT solvers (e.g. Z3 [13]) benefit from ongoing developments in heuristics aimed at solving sets of quantified constraints. In our previous work, experiments have shown that the Z3 solver is sufficiently powerful to efficiently solve the constraints generated by relational symbolic execution over small-scale SQL applications, written in our small subset of Java and SQL, allowing only core programming operations and integers as only primary type. We investigate here a more general constraint-based testing of SQL applications.

SQL applications, like the one of Figure 1, can mix various operations over various datatypes, such as strings, binary objects, numeric values and timestamps. Conversions between these types are also common, like at line 2 of the *removeBookShelf* method from Figure 1, where an integer is converted into a string. Symbolic execution of these operations requires the development of solvers able to handle quantified SMT constraints combining in new ways trusted theories, like the integer, real, array or bit-vector theories with new string and timestamp theories. Several works have studied the particular problem of multi-granularity temporal constraint solving (e.g. [3]) and several string constraint solvers have been developed (e.g. [22, 24, 41]). Research is ongoing (e.g. [42, 4, 33, 34, 39, 36]) towards a proper solving of string contraints inter-related with other kinds of constraints, in the context of symbolic execution. As the use of quantifiers and of various operations over various datatypes makes the target logic complex and generally undecidable, symbolic execution of SQL should be tailored to generate very efficient sets of constraints, expressed in parts of the logic that are decidable or optimized for the heuristics used by the solver. Building such a symbolic evaluator is made difficult by the fact that the syntax and semantics of 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 is using relational algebra as an intermediate language for symbolic execution of SQL: the original SQL code would be compiled into a minimal relational algebra, and then the algebraic code would be translated into logical constraints. Algorithms translating SQL statements into equivalent combinations of a core set of relational algebra's operators have already been developed, in the context of DBMS design [12]. Equivalence between relational algebra and logic is well defined since the birth of relational databases [11]. In practice, this idea should be refined, as SQL is more powerful [12] than relational algebra, since it allows non-relational constructs like rows ordering and aggregation, null values, built-in or user-defined function calls, etc. The intermediate language should thus be extended by a minimal set of operators for describing the most common non-relational parts of SQL. Function calls could be symbolically executed as normal procedural code. *Concretization* could be the last-ditch solution to handle exotic or too complex parts of SQL.

3.5 Detecting SQL Runtime Errors

Runtime errors can be frequent in SQL applications [12], as there is no compile-time integration between the application's code and the DMBS's code. When the DBMS encounters an error while processing an application's request, it warns the application by throwing a runtime error. Some of these errors can be caused by a fault in the application, like submitting a syntactically wrong dynamic SQL code, misusing the DMBS's API, accessing an empty cursor, violating the database's integrity constraints or writing values in a different format than specified by the database's schema. Relational symbolic execution should thus be extended to treat every SQL statement (and DBMS's API call) as a kind of switch/case statement dealing with a set of possible runtime errors, typically of the form:

```
switch (SQL Statement) {
case violates the i^th integrity constraint of the schema :
 throw new SQLException("Constraint ... violated");
case contains a syntax error :
 throw new SQLException("Syntax error: ...");
...
default:
 Execute statement. }
```

Two major challenges are, first, to detect, for a given statement, what kinds of runtime error can occur, and, secondly, to infer the constraints that drive the execution towards a particular kind of runtime error being raised or not.

As an example, symbolic execution should detect that the *getInt* method at line 6 of the *removeBookShelf* method from Figure 1 can throw an *SQLException*, if called on an empty *ResultSet*. If we model each datatype in the application by integers and replace the dynamically-crafted SELECT statement by equivalent static SQL, the symbolic evaluator from [29] can be applied to generate constraints driving the execution towards such a faulty *ResultSet* access. Solving these constraints would show that they are satisfiable (for example, if the database is empty at method's start, whatever the other inputs are) and allows thus to detect the fault concerning the *SELECT* statement, which can return less than one row for some inputs. Similarly, satisfiable constraints can be generated for the path leading to a violation of the database's CHECK constraint by the UPDATE statement at line 8 (violation occurs if the selected shelf contains two books), which is a hint at the fault signaled for the UPDATE statement (with a correct code, decreasing the number of books by one, no constraint violation would be possible and the generated constraints would be unsatisfiable). Finally, considering a unified relational and dynamic symbolic execution algorithm, any concrete execution of the method taking the *else* branch at line 11 will throw an *SQLException*, because of the syntax error in the DELETE statement. Statical detection of such syntax errors in dynamic SQL code has been studied in [20].

3.6 Handling Non-Deterministic SQL

Non-determinism creeps in many places in the SQL semantics [12]. A common example is the undefined order in which a SELECT statement can return the selected rows. For example, let us consider two runs of the *removeBookShelf* method from Figure 1 over 66 as input value for parameter *theShelf* and the following content for the table *shelf*:

id	label	numberOfBooks	nextReordering
1	"66"	560	9/9/15 11:00:00
3	"66"	1	8/9/15 9:10:00

Both rows are selected by the SELECT query at line 3 but let us suppose that they are returned in different order between

the two runs. During the first run, the value 560 is returned by the *getInt* method at line 6, triggering the execution of the *then* branch of the *if* statement. During the second run, the value 1 is returned by the *getInt* method, triggering the execution of the *else* branch of the *if* statement. This is problematic for symbolic execution, as it shows that non-deterministic SQL can allow a single application's input to trigger different execution paths in the code and to produce different outputs. Moreover, such a non-deterministic behavior of the whole application should be detected as it can be a hint at an underlying design fault in the code. In this particular case, the fault hinted at is the previously signaled fault concerning the SELECT statement, which can return more than one row for some inputs.

A common way to handle such a non-determinism in symbolic execution is to constrain the order in which the rows are returned by the SELECT statement as if it was one of the method's inputs. In such a way, any solution to a path-constraint would define a particular row order and thus trigger the execution of a single path in the code, producing a single possible output. Moreover, one can subsequently execute the method by varying the row order, while keeping the other inputs constant, to detect if the taken path and the produced output are affected. Other well-known examples of non-deterministic SQL are queries based on the current date in temporal databases, whose result can vary depending on when the query is executed, or insertion of new rows using fields values randomly selected by the DBMS. In both cases, the current date and the randomly selected field values can be considered as inputs of the tested application as well.

4. CONCLUDING REMARKS

Symbolic execution is a particularly well-suited technique for testing SQL applications, as the semantics of SQL has been built over relational algebra and first-order logics, and can thus be naturally expressed using standard SMT constraints. In our previous work, we have shown that such a constraint-based testing of SQL applications was possible with current solver techniques, at least for small-scale applications.

In this work, we have identified the main challenges and provided research directions towards constraint-based testing of full-scale SQL applications. If one-click testing of any SQL application might never be possible, notably because of the complexity and the variability of the still evolving SQL language, automating testing of most SQL applications might be possible with reasonable effort. This will notably require some additional research work to properly handle dynamic SQL (using concretization and partial evaluation), generate tractable constraints (using relational algebra as intermediate language and leveraging the power of new generations of SMT solvers), detect SQL runtime errors (using efficient error analysis and translation into constraints) and deal with non-deterministic SQL (by making the non-deterministic choice an application's input).

5. ACKNOWLEDGMENTS

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

6. REFERENCES

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

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

[3] C. Bettini, X. Wang, and S. Jajodia. Solving multi-granularity temporal constraint networks. *Artificial Intelligence*, 140(1–2):107 – 152, 2002.

[4] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 307–321. Springer, 2009.

[5] R. Bubel, R. Hähnle, and R. Ji. Interleaving symbolic execution and partial evaluation. In *Proceedings of the 8th International Conference on Formal Methods for Components and Objects*, FMCO'09, pages 125–146, Berlin, Heidelberg, 2010. Springer-Verlag.

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

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

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

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

[10] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[11] E. F. Codd. Relational completeness of data base sublanguages. *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.

[12] T. Connolly and C. Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management*. Number vol. 1 in International computer science series. Addison-Wesley, 2005.

[13] L. De Moura and 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, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] L. De Moura and N. Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, Sept. 2011.

[15] A. A. El Ghazi and M. Taghdiri. Relational reasoning via smt solving. In *Proceedings of the 17th international*

conference on Formal methods, FM'11, pages 133–148, Berlin, Heidelberg, 2011. Springer-Verlag.

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

[17] P. Godefroid. Higher-order test generation. *SIGPLAN Not.*, 46(6):258–269, June 2011.

[18] P. Godefroid, D. D'Souza, T. Kavitha, and J. Radhakrishnan. Test generation using symbolic execution. In *FSTTCS*, pages 24–33, 2012.

[19] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[20] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 645–654. IEEE, 2004.

[21] W. G. J. Halfond, A. Orso, and 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, pages 175–185, New York, NY, USA, 2006. ACM.

[22] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. *SIGPLAN Not.*, 44(6):188–198, June 2009.

[23] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[24] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 105–116. ACM, 2009.

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

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

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

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

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

[30] K. Pan, X. Wu, and T. Xie. Database state generation via dynamic symbolic execution for coverage criteria. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, New York, NY, USA, 2011. ACM.

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

[32] K. Pan, X. Wu, and T. Xie. Guided test generation for database applications via synthesized database interactions. *ACM Transactions on Software Engineering and Methodology*, 2013.

[33] G. Redelinghuys, W. Visser, and J. Geldenhuys. Symbolic execution of programs with strings. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, SAICSIT '12, pages 139–148, New York, NY, USA, 2012. ACM.

[34] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[35] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, Sept. 2005.

[36] D. Shannon, I. Ghosh, S. Rajan, and S. Khurshid. Efficient symbolic execution of strings for validating web applications. In *Proceedings of the 2Nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, DEFECTS '09, pages 22–26, New York, NY, USA, 2009. ACM.

[37] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. *SIGPLAN Not.*, 41(1):372–382, Jan. 2006.

[38] K. Taneja, Y. Zhang, and T. Xie. Moda: Automated test generation for database applications via mock objects. In *In Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE 2010), short paper*, 2010.

[39] T. Tateishi, M. Pistoia, and O. Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Trans. Softw. Eng. Methodol.*, 22(4):33:1–33:33, Oct. 2013.

[40] A. Turing. On computable numbers with an application to the " entscheidungsproblem". *Proceeding of the London Mathematical Society*, 1936.

[41] M. Veanes, P. De Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 498–507. IEEE, 2010.

[42] Y. Zheng, X. Zhang, and 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, pages 114–124, New York, NY, USA, 2013. ACM.

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