Testing and Validating End User Programmed Calculated Fields

Víctor Braberman
Diego Garbervetsky
Javier Godoy
ICC, UBA/CONICET
Argentina

Sebastian Uchitel
ICC, UBA/CONICET
Argentina
Imperial College London
UK

Guido de Caso
Ignacio Perez
Santiago Perez
Medallia Inc.
USA

ABSTRACT

This paper reports on an approach for systematically generating test data from production databases for end user calculated field program via a novel combination of symbolic execution and database queries. We also discuss the opportunities and challenges that this specific domain poses for symbolic execution and shows how database queries can help complement some of symbolic execution's weaknesses, namely in the treatment of loops and also of path conditions that exceed SMT solver capabilities.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → *Program analysis*; • **Information systems** → *Data management systems*;

KEYWORDS

Program Analysis, Symbolic Execution, Query Generation

ACM Reference Format:
Víctor Braberman, Diego Garbervetsky, Javier Godoy, Sebastian Uchitel, Guido de Caso, Ignacio Perez, and Santiago Perez. 2018. Testing and Validating End User Programmed Calculated Fields. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 7 pages.
https://doi.org/10.1145/3236024.3275531

1 INTRODUCTION

End users tend to under-test their programs and be overconfident about their correctness [4]. The lack of oracles, partly due to the overhead of producing specifications and also over-reliance on domain expertise, pushes end user program testing to the realm of informal validation rather than verification. In this context, producing test inputs based on real world, relevant data is perceived as a way of improving end user validation.

There has been significant effort in validation of end user spreadsheet programs yet it is acknowledged that there is a large community of non-developers that write database related code to support their jobs [4]. An example of the latter is code that computes values for calculated fields. The value of a calculated field is defined by the values of other fields using a procedure which can be non-trivial to write and hence error prone.

In companies providing business intelligence and analytics services and products, end-user programs for computing calculated fields, are commonplace. In this line of business, end user programmers have a strong understanding of the domain, talk directly to customers or may even be customers, and must develop calculated field programs, test and validate the code and then apply the calculation to every row of the (potentially very large) database.

Calculated field programs are written in general purpose programming languages or Turing complete domain specific programming languages, and are a ripe ground for end user programmer errors. Consequently, effective automated support for validating these programs is of critical importance.

To improve the support it provides to its calculated fields programmers, Medallia Inc. uses a restricted domain specific version of JavaScript for developing calculated fields and developed infrastructure that allows programmers to produce inputs, based on real database registers, that cover their code. Because it is real data, these inputs and their corresponding calculated field output can be more easily validated by business analysts. Furthermore, code that cannot be covered with existing data is a valuable piece of information that leads to improved business analysis.

To support rapid, interactive, provision of feedback in early stages of calculated field program development, code coverage and output coverage (the return type of calculated field is typically an enumerated type) is achieved via random selections from the database. Scanning an entire database of millions of registers for examples for each possible output is simply too expensive.

This paper reports on a joint effort to improve early provision of real (database) test data to calculated field program developers via a novel combination of symbolic execution and database queries. The aim is to use a symbolic execution engine to produce path conditions that can be translated into database queries that retrieve from databases real test data that covers the end user programmed code.

Automated test case generation techniques based on symbolic execution face significant challenges to be applied in practice [3, 8, 12] due to difficulties in reasoning about loops and recursion, and complex path conditions (those beyond SMT [3] capabilities). These difficulties are compounded, in our specific case, by the choice of a dynamic language, JavaScript, for end user programming.

However, in practice, we have found, that end user programs for calculated fields have a number of characteristics that provide a window of opportunity for using symbolic execution. These programs tend to use simple control features, notably with no loops, operate over simple data types but exhibit an intricate pattern of conditional statements that hinders human validation. This setting could be regarded as beneficial for automated test case generation based on symbolic execution engines.

The setting is not as simple as it seems. End user programs for calculated fields call non-user-defined functions that can be of significantly greater complexity, including loops and more complex data types. Such functions can be either show-stoppers for symbolic execution engines attempting to produce path conditions and for translating them into database queries.

A key observation that allows addressing these obstacles is that end user programs for calculated fields used at Medallia call a reduced API that includes only a small set of functions from the standard JavaScript library and from a set of non-user-defined domain specific functions. The latter have been designed in the spirit of a domain specific language, many of which can be dealt with efficiently if appropriately translated and embedded in a database query. Thus, the insight that follows is that the functions that are complicated to handle for a symbolic execution engine can be hidden away and delegated to the database management engine (*e.g.*, as SQL-supported constraints or stored-procedures).

Summarising, we report on a joint effort to support end user validation of JavaScript programs used for calculated fields in data warehouse applications. We discuss a prototype tool chain that given an end user JavaScript program for a calculated field generates a set of unit tests that cover end user code. Each test is created using, when they exist, real values extracted from a production database.

The main components of the tool chain are the PEX dynamic symbolic execution engine [1, 10], a path condition to database query translator to retrieve real test inputs from a production database and a test suite generator to produce unit-tests. We prevent PEX from analysing complex procedure calls within an end user program by annotating these calls as uninterpreted functions. As a result PEX produces path conditions for which the interpreted portion of the path condition is guaranteed (by construction) to be feasible, reducing the total number of infeasible paths. The translator maps procedures marked as uninterpreted into database query functions that implement them efficiently in the database engine. The resulting query exploits existing indexes for the feasible part of the path condition, typically reducing significantly the number of registers to be analysed, and then scans the rest running the more complex procedures on each register until it finds a suitable input.

In the remainder of this paper we first provide examples motivating the difficulties in producing real database test data for calculated field programs, we then describe the approach and discuss its evaluation. Finally we present related work and conclusions.

2 MOTIVATION

In this section we present a series of increasingly complex examples that illustrate the difficulties in providing real database test data for calculated field programs. The examples are set in a fictional company SALES that is analysing the introduction of a reward program. A SALES analyst becomes end-user-programmer and develops code for a calculated field that classifies customers into different tiers (basic, frequent, silver, gold, and platinum).

```
function CF_Points(row) {
  var points = row.points;
  if (points == null) return null;
  if (points > 400000) return 5; // Platinum
  if (points > 200000) return 4; // Gold
  if (points > 100000) return 3; // Silver
  if (points > 50000) return 2; // Frequent
  return 1; // Basic
}
```

Listing 1: A simple calculated field program

A first basic program might simply classify customer into fields based on their accumulated points. The function in Listing 1 shows a possible implementation of this criterion.

To cover all statements in this program it suffices to provide 6 different inputs, one for each return statement. To generate the input for one specific return statement, an input that satisfies the chain of conditions determined by the control path to the statement is required. In other words, an input that satisfies the *path condition* for each return statement is needed.

A sequential scan or random picking from the SALES production database may be used to show the SALES analyst real examples of customers that fall into each tier. Depending on the distribution of customers into the 6 return values and the size of the database, this may be a costly approach.

A symbolic execution engine like PEX [10] or JPF can automatically produce synthetic inputs to cover statements. A by-product that is sometimes offered is the path condition of the program under test. Each condition is a predicate on the input parameters that must hold in order to reach a particular statement. These path conditions can be used to produce database queries that can find efficiently by exploiting database engine infrastructure, including indexes, registers that satisfy each path condition.

As an example, the path condition that corresponds to return value 4 can be used to generate the following database query:

SELECT * FROM row WHERE row.points < 400001 AND 200000 < row.points limit 1;

A further refined and more complex customer classification criterion, possibly elicited after analysing previous criteria, may reward customers who have been in the program for longer. It could include calls to standard library functions to calculate differences between dates as the function `Years(Date lastSeen, Date signupDate)`.

Despite the added complexity and system library calls, a symbolic engine such as PEX is still able to produce inputs that cover all statements for that function:

... ∧ (((4611686018427387903uL & (13835058055282163712uL ^row.lastSeen.dateData)) - (4611686018427387903uL & (13835058055282163712uL ^row.signupDate.dateData))) / 864000000000L) / 365 <3

Note that PEX analyses all the code, including that of the library functions. Thus, the path condition predicates over the internal .NET representation of dates. We do not need nor want the symbolic execution engine to delve into the full detail of operations and data types that can be managed by the database engine.

To tackle this problem we need to tell the symbolic execution engine to handle some specific calls as *uninterpreted*, preventing the engine from reason within these calls. Note that symbolic execution engines typically do not provide annotation mechanisms for identifying uninterpreted functions. For the prototype reported in this paper, some tricks that force the symbolic engine to consider some

```
1   function CF_PointPromotion(row) {
2     ...
3     if (relevantDays < 3)  return CF_Points(row);
4     if (IsPlatinum(row) == 1)   return 5;
5     ...
6   }
7   function IsPlatinum(row){
8     return IsPlatinum(row, CF_ComputePointsReqForCountry(row));
9   }
10  function IsPlatinum(row, pointsForPlatinum) {
11    var points = row.lastPoints.split(';');
12    if (points.Length < row.numDays) return 0;
13    var isPlatinum = true;
14    for(i = 0; i < row.numDays; i++) {
15      var calcPoint = int.Parse(points[i]);
16      isPlatinum = isPlatinum && (calcPoint > pointsForPlatinum);
17    }
18    if (isPlatinum) return 1;
19    return 0;
20  }
21  function CF_ComputePointsReqForCountry(row) {
22    if (row.countryCode==null) return null;
23    if (row.countryCode==1 || row.countryCode==44) return 150000;
24    if (row.countryCode==54 || row.countryCode==55) return 250000;
25    return null;
26  }
```

Listing 2: Calculated field with a loop

calls as uninterpreted were devised (see Section 3). By declaring the function `Years` as uninterpreted the path condition will look as $\ldots \wedge Years(lastSeen, signupDate) < 3$.

If uninterpreted functions have straightforward mappings to functions supported by the database query language, the path condition can then be translated into a database query.

A further refinement of the reward program may be to offer upgrade to customers that acumulated points above some baseline for a certain number of days in a row. The required amount of points varies according to the country of origin. In the code below, the function `IsPlatinum` reads a text field containing a semicolon separated list of points (*e.g.*, `lastPoints` = "140; 23; 526; ...; 410") and checks that the first `numDays` values are above a threshold determined by the country. There are also some extra conditions that are shown in Listing 2. This calculated field program goes beyond the capabilities of symbolic execution engines, which cannot produce an input nor a path condition that covers line 4. One way to overcome this issue is to postpone the analysis of the complex method `IsPlatinum` and leave this job to the database engine. This, of course can only be done if the database query language supports an equivalent function. If function `IsPlatinum(row, pointsForPlatinum)` is marked as uninterpreted then PEX is able to obtain a path condition for line 4:

$$\text{row.lastPoints} \neq \text{null} \wedge \text{row.numDays} \neq \text{null} \wedge 1 < \text{row.numDays} \wedge \\ \text{row.countryCode} \neq \text{null} \wedge \text{row.countryCode} = 1 \wedge \text{IsPlatinum(row, 150000)} = 1$$

Execution of queries such as the above will typically include cutting down the number of records very efficiently. Using the conditions on `countryCode`, `points` and `numDays`, the database query would first find a reduced number of candidates records and then will need to check the condition `IsPlatinum` on it reduced set of records. This is a significantly more efficient mechanism than random picking or sequential scan over the entire database. For a database with 100 million records a sequential scan (and execution of a simple function such as the one above on each register) can be 5 orders of magnitude slower than path condition generation, translation and query execution.

3 APPROACH

In this section we report on an approach aimed at assisting end user programmer validation of programs used for computing calculated fields in data warehouse applications.

In Figure 1 we present a flow chart that sketches how calculated fields are processed to tests using inputs from real database data. The prototype consumes a JavaScript program that computes a calculated field and produces i) a set of test inputs taken from database records that cover statements of the calculated field program, and ii) a set of synthetic test inputs that cover the statements for which no input in database exists. Programs are first pre-processed to mark uninterpreted functions. PEX [1, 10] , a symbolic execution engine, is then used to produce path conditions to cover all statements. Path conditions are translated into database queries which are then run to retrieve real test input data. Finally, the test inputs are used to produce executable tests. PEX also produces synthetic inputs. For every path condition that produced a query whose result was empty, the synthetic input for that path condition generated by PEX is used to generate additional tests to augment coverage.

3.1 A Domain Specific Language

Before discussing the main procedures of Figure 1 we discuss a pre-condition of the approach: the existence of a domain specific language (DSL) for calculated field development. This language may be de-facto or formally defined and enforced.

In Medallia, the DSL restricts end user programmers to writing functions that have as an input a database register and output the value of particular fields (*i.e.*, the calculated field). Programmers are not allowed to access the database itself, hence calculated fields that consider aggregation from several rows results are not allowed.

In addition, a library of domain specific functions is provided. These functions raise the level of abstraction of the database fields, encapsulate business rules, and can be called by end user programs. An example of such functions in Section 2 is `IsPlatinum`.

Finally, the language defines the functions that are to be considered uninterpreted calls by the symbolic execution engine. These functions calls will appear verbatim in path conditions and need to be translatable into expressions that the database engine can handle in a query. For example `Years(lastSeen,signupDate)` can be defined to be the expression `date_part('year', age(date1, date2))` while `IsPlatinum(row,num)` may be implemented directly as a stored procedure.

The choice of functions to be considered uninterpreted is crucial, highly domain dependent, and may need to be revised over time. As the goal is to build database queries directly from path conditions, it is undesirable to include in path conditions low level constraints arising from code that is already functionally supported by the database query engine. Examples of these are math operations such as `round` and date operations. Marking these functions as uninterpreted will make the symbolic execution engine consider them black boxes, making them appear as symbolic expressions in the path condition. The uninterpreted manipulation of `Years` in Section 2 exemplifies this case.

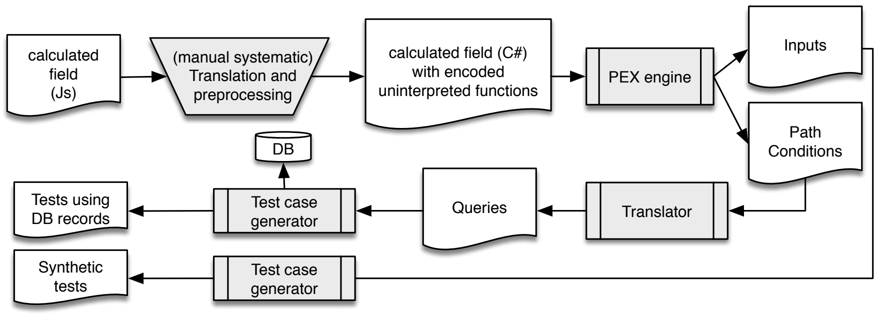

Figure 1: Real Data Test Case Generation

It is also worth considering function calls as uninterpreted when end-user programs implement stable functionality that has complex control structures or data types that can break the symbolic execution engine. In these cases it may be preferable to translate them to expressions that the database query engine can manage. The uninterpreted manipulation of `IsPlatinum` in Section 2 is an example. Another is code that can be efficiently implemented in terms of optimised access to internal tables.

3.2 Calculated Field Program Preprocessing

Preprocessing involves marking uninterpreted functions calls in the calculated field program according to the definition of the DSL. Although conceptually simple, our choice of symbolic execution engines introduced two accidental challenges: programming language translation and forcing symbolic execution to not interpret marked functions calls.

As mentioned previously, JavaScript is the language used by Medallia's end user programmers. We considered symbolic execution tools for JavaScript, including Jalangi[9], concluding they were insufficiently robust. We also considered using Symbolic JPF [7], which is for Java and supports user defined uninterpreted function calls. However, the tool does not correctly support conditions over null values. We finally opted for PEX which despite supporting .NET rather than JavaScript and not supporting user marked uninterpreted calls, has the advantage of being a well established industrial strength tool (*e.g.*, [1]) that is available through Visual Studio Suite. PEX is actually a concolic execution engine: it combines concrete and symbolic execution to produce inputs covering a determined set of statements of a given procedure.

Having chosen PEX, the first accidental issue that the prototype's preprocessing must address is the translation of JavaScript into C#, which for calculated field end user programs is straightforward and can be systematically done manually. Given a end user JavaScript program and a list of uninterpreted functions, translation starts at the program entry point and recursively translates each JavaScript method into a corresponding C# method. This process ends at calls to uninterpreted functions, which are not translated.

The second issue is that PEX does not support user specification of calls that are to be treated as uninterpreted. To overcome this limitation, we convert function calls that should be uninterpreted into C# n-dimensional arrays. The trick behind this encoding is to use PEX's ability to handle arrays symbolically. Given an uninterpreted method $uninterpreted(a_1, ..., a_n)$, we encode it in an n-dimensional array of integer types $uninterpreted[a_1\_self, \ldots, a_n\_self]$. The idea is to use the *ghost* variables $a_1\_self, \ldots, a_n\_self$ of integer type as proxies of the original arguments of the uninterpreted function (of arbitrary types). Using this encoding, PEX can produce an input and a path condition by instantiating the array and ghost variables with arbitrary values.

3.3 Path Condition Generation

Once the calculated field programs are preprocessed and translated to C# we are ready to run PEX and produce path conditions. The interpreted portion of the conditions is guaranteed to be feasible, reducing the total number of infeasible paths. The uninterpreted part is evaluated on the database in the next phase.

For instance, the verbatim path condition returned by PEX for the `Years` function in the previous section is:

```
... && Years != (int[,])null && lastSeen_self >= Years.GetLowerBound(0)
&& lastSeen_self < Years.GetLowerBound(0) + Years.GetLength(0)
&& signupDate_self >= Years.GetLowerBound(1)
&& signupDate_self < Years.GetLowerBound(1) + Years.GetLength(1);
```

The following is a path condition for `CF_PointPromotion`:

```
row.lastPoints ≠ null ∧ 1 <row.numDays ∧
((row.countryCode == null ∧ IsPlatinum(row._self, 400000) == 1)
∨ (row.countryCode == 1 ∧ IsPlatinum(row._self, 150000) == 1)
∨ ...∨ (row.countryCode ≠ 1 ∧ row.countryCode ≠ 44 ∧
    row.countryCode ≠ 54 ∧ row.countryCode ≠ 55 ∧
    IsPlatinum(row._self, 400000) == 1)
∨ (row.countryCode == 55 ∧ IsPlatinum(row._self, 250000) == 1))
```

Note that PEX actually returns multiple path condition that cover a statement, to account for different paths that reach the statement. As we choose in this paper statement coverage, we integrate the path conditions with a disjunction.

3.4 Path Condition to Query Translation

Converting path conditions into database queries requires that all functions identified in the DSL as uninterpreted have a mapping mechanism from function calls to equivalent query expressions.

First we reverse from each path condition the encoding for uninterpreted function calls introduced in the previous section. For example in the case of array `Years[lastSeen_self,lastSeen_self]`, the function call `Years(lastSeen, lastSeen)` is extracted. It also applied minor syntax transformations such as replacing expressions `e.hasvalue != false` by $e! = null$ and removes side-effect conditions produced by the encoding for uninterpreted calls. Note that treatment of null values is different in SQL than in JavaScript and C#. Finally, uninterpreted calls are replaced by their equivalent query expressions.

3.5 Test Case Generation

This process produces a unit tests by executing queries against a database and using the selected records as inputs for unit tests.

Each query, when executed by the database query engine, exploits existing indexes for the feasible part of the path condition, typically reducing significantly the number of registers to be analysed, and then scans the rest running the more complex procedures on each register until it finds a suitable input.

Should the query result be empty, this means that there are no database records that satisfy the original path condition. In these cases, it may be possible to obtain from PEX for a synthetic input instead. The only potential problem is that the synthetic input is, by construction, consistent with the interpreted part of the path condition but may be inconsistent with the uninterpreted function call expression.

Generated inputs, real or synthetic, are then used to produce the test cases. The prototype uses Mocha [2] to execute synthesised tests and measure coverage of calculated field programs.

4 EVALUATION

The purpose of the prototype described above is to understand the feasibility of applying advanced test case generation techniques for systematically covering with real data end user calculated field programs at Medallia.

We used 9 anonymised programs and database consisting of 21000 records extracted from an internal production database. The aim of the evaluation was to assess to what extent these programs, considered representative of those developed at Medallia by end users, could be covered by the prototype.

The programs operate over simple data types such as integer, floats and strings and outputting enumerated types (in 7 out of 9 cases), integers (1/9) and strings (1/9). They lack loops and recursion but exhibit intricate patterns (*e.g.*, nested conditional statements, early program exit with return statements) that hinder human validation. Other characteristics of the programs include the use of many checks for null values stored in fields, integer, float and string manipulation in conditional statements using JavaScript standard functions such as indexOf(), toUpperCase(), parseInt(), and Round(). In addition, the programs included multiple invocations to functions of a Medallia library some of which essentially encapsulate lookups on auxiliary tables, others supporting operations such as regular expression search over strings.

We first developed the list of functions to be treated as uninterpreted. The list was defined to include all functions in the Medallia library plus operations over float, string and date types. A mapping for each one to equivalent database query expressions was developed. For basic types, the mapping is very straightforward (see `Years` in Section 3.1). For Medallia library functions supporting lookups, for instance, SQL join expressions were used.

Having then translating manually all programs into C#, tests data was generated. We measure statement and branching coverage of the calculated field programs.

We report coverage using two different notions of test unit: each calculated field program can be thought of as a testing unit, as it is common that end user programmers develop or change one program leaving the rest unchanged. However, many of these programs actually call other calculated field programs which were developed by other end user programmers, and in many cases these programs need to be changed consistently together. Thus we also consider as a test unit the entire set of calculated field programs.

Table 1: Analysed programs.

Anonynous CF	Stms	Covered Stms	%	Tests
CF_1	10	10	100.00	5
CF_2	6	6	100.00	4
CF_3	8	8	100.00	5
CF_4	11	10	90.91	10
CF_5	4	4	100.00	2
CF_6	7	7	100.00	4
CF_7	11	11	100.00	5
CF_8	6	6	100.00	4
CF_9	11	11	100.00	4
Total	74	73	98.65	43

When considering the set of calculated field programs as a test unit, we computed coverage achieved by executing all tests generated for each calculated field program individually. This amounts to 44 path conditions converted to database queries. All queries but one returned results, which led to 43 test cases covering 98% statement coverage. Although we did not aim for branch coverage, the tests achieved 92%.

We also report (see Table 1) the coverage of test units comprising only one calculated field program at a time. The tool covered all statements in each individual calculated field program with the exception of one statement which could not be covered due to the non-existence of records in the database satisfying the corresponding query. The symbolic execution engine, however, was able to produce a synthetic input to cover the statement.

Our experimentation was run on an anonymised version of a production database. To gain insight on the domain relevance of test cases that the approach can produce, we provided the experimental subjects and database queries to independent end user programmers at Medallia. They de-anonymised field names in code and queries and then ran the queries on the production database. While validating the resulting test cases, the end user programmers identified one mismatch between informal requirements and the calculated field program. The mismatch had to that point not been identified. Furthermore, the mismatch was subtle enough that initially end users assumed that there was a problem of the prototype test case generator.

Both the high coverage and the success in producing findings regarding existing calculated field programs are promising results that provide preliminary evidence that symbolic execution combined with database queries is a technically feasible solution for Medallia's end user calculated field programs.

5 RELATED WORK

Testing of end user programs has attracted much attention, much of it addressing spreadsheet programs [4]. We are not aware of any work supporting testing and validation of end user calculated fields programs written in general purpose programming languages. Other kinds of end user programs that are related to databases have been studied though, most notably those that include explicit database queries in the code and assume a fixed database state. Those approaches aim at practical code or query coverage by generating program inputs [5, 6] or by reducing database state (e.g., [11]).

Although, in some cases, part of their internal workings resemble ours (e.g., construction of auxiliary queries based on intermediate information) there are some key differences: firstly, we pursue detecting database records covering statements of an imperative program, there is no SQL-query or query-manipulating code in the programs we target. Secondly, our test-cases retrieval-queries end up executing user defined code or some equivalent database query code. Finally, a distinctive aspect of our approach is the identification of uninterpreted functions to simplify symbolic analysis and transfer complexity to the database engine.

6 LESSONS LEARNED

The project has left us with some lessons learned that may be of use in a more general setting: when coverage needs to be achieved using test cases based on real inputs taken from large databases, a viable alternative to sequential scan of the data or random picking can be to use symbolic execution to generate efficient queries over the data set.

We identified three main pitfalls when attempting to generating database queries from path conditions. In some cases the code is too complex for the symbolic execution engine to output a path condition (e.g., code with complex iterative or recursive structure). Alternatively, the path condition may be expressed in terms of low level constraints that include how types are represented in the program and which cannot be mapped to the database. Finally, a path condition may be over-constrained, due to the complexity of the code, leading to a query that does not retrieve any data.

A viable approach to overcome these limitations is to identify code blocks that can be declared as uninterpreted predicates over the data set. Note that, importantly, it must be possible to implement these predicates efficiently using the database query engine (e.g, using native SQL operators or developing within the database engine appropriate user defined functions). Identifying uninterpreted code facilitates the symbolic engine's task as it can avoid complex loops and low level representation issues.

The resulting high level path condition can be translated into a query to select from the database an appropriate input. Note that a downside of introducing uninterpreted code blocks is that the symbolic engine cannot guarantee the feasibility of a resulting path condition, hence the approach may generate more database queries than necessary.

7 CONCLUSIONS

We have reported on a project to improve testing and validation of end user programs that compute values for calculated fields. The approach is based on symbolic execution to systematically achieve coverage and database queries to obtain test inputs based on real data. Identification of calls that should not be interpreted during symbolic reasoning and resolved by the database engine is a key part of the approach. The test case generation prototype has shown that such a tool chain can fully cover statements of real calculated field end user programs using real data.

Our approach hints that for some domains, the challenge for symbolic execution engines is not scale, loops, nor SMT support for more complex theories (all three of which are main thrusts of the community). In some domains, increased applicability of symbolic execution may be achieved by providing support for user defined uninterpreted functions and better supporting more end user oriented programming languages such as JavaScript.

This work has led us to an additional observation that may inform research in test case generation based on dynamic symbolic execution: in some domains, rather than trying to produce and solve difficult path conditions, an efficient search over real values in a database may not only allow covering code that otherwise remains uncovered but also can provide tests that use real data, aiding test comprehension and validation. In other words, there may be benefits to produce a more abstract path conditions, avoiding the interpretation of functions, that can be handled by a database engine. If the database is sufficiently populated to always have at least one record that satisfies each query and sufficiently efficient to process queries, a method for systematically constructing test cases that use realistic data for end user programs can be deployed.

Future work aimed at the construction of a fully automated test case generator for Medallia will include incorporating and improving JavaScript symbolic execution tools of the likes of Jalangi[9], including the addition of support for user defined uninterpreted functions. More generally we believe there are opportunities in investigating the use of database engines to replace limitations of SMT solvers for test case generation.

ACKNOWLEDGEMENTS

This work was partially supported by ANPCYT PICT 2014-1656, 2015-3638, 2015-1718, CONICET PIP 2015-0931CO, 2014-0688CO. Research partly supported by the European Unions Horizon 2020 research and innovation programme under the Marie Skodowska-Curie grant agreement No 778233.

REFERENCES

[1] Visual Studio Intellitest. https://docs.microsoft.com/en-us/visualstudio/test/intellitest-manual/.
[2] Mocha javascript test framework, 2017. https://mochajs.org/.
[3] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.
[4] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck. The state of the art in end-user software engineering. *ACM Comput. Surv.*, 43(3):21:1–21:44, Apr. 2011.
[5] 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.
[6] K. Pan, X. Wu, and T. Xie. Program-input generation for testing database applications using existing database states. *Automated Software Engg.*, 22(4):439–473, Dec. 2015.
[7] C. S. Păsăreanu and N. Rungta. Symbolic pathfinder: Symbolic execution of java bytecode. ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.
[8] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, USA*, pages 317–331. IEEE CS, 2010.
[9] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of javascript. In *ESEC/FSE 2013*, pages 615–618. ACM, 2013.
[10] N. Tillmann and J. De Halleux. Pex–white box test generation for. net. *Tests and Proofs*, pages 134–153, 2008.
[11] J. Tuya, C. d. l. Riva, M. J. Suarez-Cabal, and R. Blanco. Coverage-aware test database reduction. *IEEE Trans. Softw. Eng.*, 42(10):941–959, Oct. 2016.

[12] X. Xiao, S. Li, T. Xie, and N. Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In E. Denney, T. Bultan, and A. Zeller, editors, *2013 28th IEEE/ACM, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 246–256. IEEE, 2013.