SEDGE: Symbolic Example Data Generation for Dataflow Programs

Kaituo Li*, Christoph Reichenbach†, Yannis Smaragdakis‡, Yanlei Diao*, Christoph Csallner§
*Computer Science Department, University of Massachusetts, Amherst, USA
†Institute of Informatics, Goethe University Frankfurt, Germany
‡Department of Informatics, University of Athens, Greece
§Computer Science and Engineering, University of Texas at Arlington, USA

***Abstract*—Exhaustive, automatic testing of dataflow (esp. map-reduce) programs has emerged as an important challenge. Past work demonstrated effective ways to generate small example data sets that exercise operators in the Pig platform, used to generate Hadoop map-reduce programs. Although such prior techniques attempt to cover all cases of operator use, in practice they often fail. Our SEDGE system addresses these completeness problems: for every dataflow operator, we produce data aiming to cover all cases that arise in the dataflow program (e.g., both passing and failing a filter). SEDGE relies on transforming the program into symbolic constraints, and solving the constraints using a symbolic reasoning engine (a powerful SMT solver), while using input data as concrete aids in the solution process. The approach resembles dynamic-symbolic (a.k.a. "concolic") execution in a conventional programming language, adapted to the unique features of the dataflow domain.**

In third-party benchmarks, SEDGE achieves higher coverage than past techniques for 5 out of 20 PigMix benchmarks and 7 out of 11 SDSS benchmarks and (with equal coverage for the rest of the benchmarks). We also show that our targeting of the high-level dataflow language pays off: for complex programs, state-of-the-art dynamic-symbolic execution at the level of the generated map-reduce code (instead of the original dataflow program) requires many more test cases or achieves much lower coverage than our approach.

I. Introduction

Dataflow programming has emerged as an important data processing paradigm in the area of big data analytics. Dataflow programming consists of specifying a data processing program as a directed acyclic graph. Internal nodes of the graph represent operations on the data, for example, using relational algebra primitives such as filter, project, and join, or functional programming primitives such as "map" applications of user-defined local functions and "reduce" operations that collect values over sets of data. The edges in the graph represent data tables or files passed between operators (nodes) in the graph. Many recently proposed data processing languages and systems, such as Pig Latin [18], DryadLINQ [13], and Hyracks/ Asterix [1] resemble dataflow programming on datasets of enormous sizes. A user can develop dataflow programs by either writing the programs directly using the above languages or compiling queries written in declarative languages such as SQL and Hive [24].

When a user writes a dataflow program, he/she will typically employ example data or test cases to validate it. Validating with large real data is impractical, both for reasons of efficiency (running on large data sets takes a long time) and for reasons of ease-of-validation (it is hard to tell whether the result is what was expected). One alternative is to sample the real data available. The sample data need to thoroughly exercise the program, covering all key behavior of each dataflow operator. This is very hard to achieve via random sampling, however. For instance, equi-joining two sample data tables of small size is likely to produce an empty result, if the values being joined are distributed arbitrarily.

Another alternative is to synthesize representative data. Such data synthesis is complicated by the complexity of dataflow language operators as well as by the presence of user-defined functions. Current state-of-the-art in example data generation for dataflow programs [18] is of limited help. Such techniques can generate high-coverage data for dataflow programs with simple constraints. However, for dataflow programs with complex constraints, e.g., with numerous filters, arithmetic operations, and user-defined functions, the generated data are incomplete due to shortcomings in constraint searching and solving strategies.

In this paper, we address the problem of efficient example data generation for complex dataflow programs by bringing *symbolic reasoning* to bear on the process of sample data generation. We present the first technique and system for systematically generating representative example data using dynamic symbolic execution (DSE) [8], [12], [25] of dataflow programs. Our concrete setting is the popular Pig Latin language [19]. Our DSE technique analyzes the program while executing it using sampled data, determines whether the sampled input data are complete (i.e., achieve full coverage), and, if not, attempts to synthesize input tuples that result in the joint sampled and synthesized data being a complete example data set for the program.

We have implemented this approach in SEDGE, short for *Symbolic Example Data GEneration*. SEDGE is a reimplementation of the example generation part in the Apache Pig dataflow system, which currently implements the closest comparable past research, by Olston et al. [18].

Illustration: For a simple demonstration, consider an application scenario in computational astrophysics. We surveyed

11 queries in the Sloan Digital Sky Survey[1] for analyzing star and galaxy observations, and rewrote them using the Pig Latin language. The most complex query contains 34 filters and 2 joins. For ease of exposition, we show a simple example query in Listing 1 by combining features from two actual queries and will use it as a running example in the paper. (For more details on the real queries, see the evaluation section.)

```
A = LOAD 'fileA' using PigStorage()
  AS (name:chararray, value:int);
B = LOAD 'fileB' using PigStorage()
  AS (u:double, class:int);
C = FILTER A BY value < 100 AND value >= 0;
D = FILTER B BY math.POW(u,2.0) > 0.25;
E = JOIN C ON value, D ON class;
```

Listing 1. An example Pig Latin program

The program begins by loading tables A and B from files containing measurements. Both kinds of measurements need to be filtered. The first filter keeps only measurements in a certain value range and the second filter removes low u values. The tuples that survive the filtering get joined.

Imagine that we execute the program for a small number of sampled input tuples from `fileA` and `fileB`. If we want to achieve perfect coverage on random sampling of actual data alone, we are unlikely to be successful if the sample is small. The data from the two tables need to pass filters and (even more unlikely) have their `value` and `class` fields coincide. This is a case where targeted test data generation can help.

Past techniques for example data generation cannot handle this example well. Olston et al.'s technique [18] will synthesize data by considering operators one-by-one in reverse order in the Pig Latin program. It will attempt to create data to satisfy the JOIN first, without concern for the FILTER conditions that the same data have to satisfy. This will likely fail to satisfy even the first FILTER operator: the range 0-99 will have to be hit purely by chance. The problem for the Olston technique is that `value` is not a free variable once the JOIN constraint is satisfied: it is limited to the values that the system arbitrarily chose in order to have the JOIN operator produce output.

Even more importantly, the second FILTER operator is hard to process. It contains a user-defined function, `math.POW`. Although this function is simple, it will still befuddle an automatic test data generation system. Furthermore, an essential part of dataflow programming is the ability to use user-defined functions freely, however complex these functions may be. The large volume of work on automatic data generation in other settings (e.g., SQL databases [3]) does not address user-defined functions.

Our approach overcomes such problems by modeling the entire program in a powerful reasoning engine, handling complex conditions, and dealing with user-defined functions with the aid of concrete values observed over sample data. We process the program using a domain with (symbolic) variables, such as *value*, *class*, etc. A symbolic variable "*columnname*" represents the value of one column of an input table for a set of tuples. We start with a concrete execution of the program using small samples of real input data. During such concrete execution we observe, first, which program cases are covered, and, second, what are the values of user-defined functions for real data. E.g., a tuple $(3.3, 32)$ of table B will register the value pair $(u : 3.3, \textit{math.POW(u,2.0)} : 10.89)$ for the user-defined function. This value will later help when trying to solve symbolic constraints.

After the concrete execution, our approach uses symbolic reasoning to cover program cases that were not already covered by the concrete execution. The approach performs a symbolic execution of the program, gathering constraints along each path to the sources.

We use the Z3 SMT solver [10] to solve the constraints. Concrete values for user-defined functions are supplied to the solver. That is, the user-defined function is treated as a black-box and the solver is supplied extra constraints of the form $u = 3.3 \Rightarrow \textit{math.POW}(u, 2.0) = 10.89$. These can aid the solver in producing satisfying assignments. Essentially, we try to make an educated guess: Whenever we do not know how to generate example data for a constraint that depends on a user defined function, we can always simplify this constraint by replacing the symbolic representation of the user-defined function with concrete values.

Contributions: In brief, the contributions of our work are as follows:

- We detail a translation of dataflow operators into symbolic constraints. These constraints are subsequently solved using an SMT solver.
- We adapt the technique of dynamic-symbolic execution to the domain of dataflow languages. By doing so, we exploit the unique features of this domain, thus enabling high coverage. Specifically, we exploit the absence of side-effects in order to perform a multiple-path analysis: observations on the values of a user-defined function on different execution paths can help solve constraints involving the user-defined function.
- As a result of the above, we produce an example data generation technique that achieves higher coverage than past literature, managing to produce data that exercise all operators of all programs that we examined. We show extensive measurements to confirm our approach's advantage. Our technique achieves full coverage in all benchmark programs with a boost in performance for most benchmarks.

II. Background and Context

We next discuss some pertinent background on dataflow programming as well as on concepts and mechanisms introduced in closely related past work.

A. Dataflow Program

A dataflow program is a directed bipartite graph, separating computations (i.e., operators) in one partition and computational intermediate results (i.e., data tables) in the other partition. In other words, it is a graph in which data tables *flow into* operators, and operators flow into data tables. A

[1] http://skyserver.sdss.org/public/en/help/docs/realquery.asp

data table is a collection of tuples with possible duplicates. A tuple is typically a sequence of atomic values (integer, long, float, chararray, etc.) or complex types (tuple, bag, map). An operator usually has some input tables and one output table. We say that a data table is an input table of an operator in a dataflow program if the data table flows into the operator. Similarly, we say that a data table is an output table of an operator in a dataflow program if the operator flows into the data table. If operator A's output table is one of operator B's input tables, A is said to be an *upstream neighbor* of B and B is said to be a *downstream neighbor* of A. An operator without any upstream neighbor is called a *leaf* operator, and an operator without any downstream neighbor is called a *root* operator—the root operator generates the final output.

B. Pig Latin

Pig Latin is a well-known dataflow programming language and the language front-end of the Apache Pig infrastructure for analyzing large data sets. The Pig compiler translates Pig Latin programs into sequences of map-reduce programs for Hadoop. A Pig Latin program is a sequence of statements where each statement represents a data transformation. In a Pig Latin statement, an operator processes a set of input tables and produces an output table. Following are the core operators of Pig Latin [19].

1) LOAD: Read the contents of input data files.
2) FILTER: Discard data that do not satisfy a built-in logic predicate or a user-defined boolean function.
3) COGROUP: Divide one or more sets (more accurately "bags", but we use the term "set" informally) of input tuples into different groups according to some specification. Each resultant output tuple consists of a group identifier and a nested table containing a set of input tuples satisfying the specification.
4) GROUP: A special case of COGROUP when only one set of input tuples is involved.
5) TRANSFORM: Apply a transformation function to input tuples. Transformation functions include projection, built-in arithmetic functions (e.g., incrementing a numeric value), user-defined functions, and aggregation. An aggregation is implemented by first invoking COGROUP or GROUP, and then doing transformation group by group. For example, *Average* $(\cdot)$ is an aggregation that averages the values in each group of input tuples.
6) JOIN: Equijoin tuples from two input tables.
7) UNION: Vertically glue together the contents of two input tables into one output table.
8) FOREACH: Apply some processing to every tuple of the input data set. FOREACH is often followed by a GENERATE clause to pick a subset of all available fields.
9) DISTINCT: Remove duplicate tuples from the input data set.
10) SPLIT: Split out the input data set into two or more output data sets. A condition argument determines the partition that each tuple of the input data goes into.
11) STORE: Write the output data set to a file.

C. Equivalence Class Model

Our work tries to maximize branch coverage in Pig Latin programs. An interesting question is what constitutes full coverage of a Pig Latin operator. In some cases the answer is clear: the FILTER operator, for instance, is well-covered when its input contains tuples that satisfy the filter condition and tuples that fail the filter condition. In other cases, the definition of coverage is not as simple. For instance, do we consider a UNION operator sufficiently covered if all its output tuples come from a single input table (i.e., if one of its input tables is empty)? The choice is arbitrary but the more reasonable option seems to be to require that both inputs of a UNION operator be non-empty. Furthermore, whether an operator is covered may be more convenient to discern in some cases by observing its input and in others by observing its output. For instance, a JOIN is well-covered when its output is non-empty, while a UNION is well-covered when its inputs are both non-empty.

To specify the coverage of operators we inherit the definition of *equivalence classes* from Olston et al. [18]—the research work that has formed the basis of the example generation functionality in Apache Pig. Each Pig Latin operator yields a set of equivalence classes for *either its input or its output tuples*. Equivalence classes partition the actual set of tuples—each tuple can belong to at most one equivalence class per operator. To generate example data with 100% coverage, the input or output table of each operator (when the program is evaluated with the example data) must contain at least one tuple belonging to each of the operator's equivalence classes.

We summarize the equivalence class definitions for the operators of Pig Latin below. The definitions are from Olston et al.'s publication [18] and the implementation in Apache Pig.[2]

- LOAD/STORE/FOREACH/TRANSFORM: Every input tuple is assigned to the same class E_1. (I.e., the operator is always covered, as long as its input is non-empty.)
- FILTER: Every input tuple that passes the filter is assigned to a class E_1; all others are assigned to a class E_2. (The intention is to show at least one record that passes the filter, and one that does not pass.)
- GROUP/COGROUP: Every output tuple is assigned to the same class E_1. For every output tuple, the nested table for every group identifier must contain at least two tuples. (The purpose of E_1 is to illustrate a case where multiple input records are combined into a single output record.)
- JOIN: Every output tuple is assigned to the same class E_1. (The intention is to illustrate a case of two input records being joined.)
- UNION: Every input tuple from one input table is assigned to E_1, tuples from the other input table are assigned to E_2. (The aim is to show at least one record from each input table being placed into the unioned output.)
- DISTINCT: Every input tuple is assigned to the same class E_1. For at least one input tuple to DISTINCT, there must

[2]http://pig.apache.org/

be a duplicate, to show that at least one duplicate record is removed.

- SPLIT: Every input tuple that passes condition i is assigned to class E_{i1}; input tuples that do not pass i are assigned to a class E_{i2}. The number of equivalence classes of a SPLIT depends on how many conditions the SPLIT has. If a SPLIT has n conditions, it yields $2n$ equivalence classes. (The aim is to show, for each split condition, at least one record that passes the condition, and one that does not pass.)

D. Quantitative Objectives

We use two metrics to describe the quality of example data and follow earlier terminology [18]:

1) Completeness: The average of per-operator completeness values. The completeness of an operator is the fraction of the equivalence classes of the operator for which at least one example tuple exists. An ideal algorithm should find example data for every equivalence class of every operator in a Pig Latin program.
2) Conciseness: The average of per-operator conciseness values. The conciseness of an operator is the ratio of the number of operator equivalence classes to the total number of different example tuples for the operator (with a ceiling of 1). An ideal algorithm should use as few example tuples as possible to illustrate the semantics of an operator.

The completeness metric is clearly a metric of coverage, as defined earlier. Specifically, it corresponds to *branch coverage* in the program analysis and software engineering literature. Branch coverage counts the percentage of control-flow branches that get tested.

III. SEDGE Design

Our system, SEDGE, uses a three-step algorithm to generate example data in Pig Latin programs.

(1) Downstream Propagation: execute programs using sampled real data, record values of user-defined functions—see Section III-B;
(2) Pruning Pass: eliminate redundant data so that each covered equivalence class only contains a single member;
(3) Upstream Pass: generate constraints and synthesize data for equivalence classes that the sampled test data do not explore by performing DSE.

The last pass (upstream pass) is the key new element of our approach and is described next.

A. Constraint Generation

The essence of our approach is to represent equivalence classes symbolically and to produce symbolic constraints that describe the data tuples that belong in each equivalence class. Solving the constraints (i.e., producing data that satisfy them) yields our test inputs. Our constraint generator steps through the dataflow graph to compute all equivalence classes for each Pig Latin operation, starting at root (i.e., final) operators. We assume that each root operator is of the form STORE W, without loss of generality (the analysis enters dummy nodes of this form when they are implicit). Similarly we assume that all variable names in our program are unique.

We represent the set of constraints (one for each equivalence class) of a statement V = ... as $\mathcal{C}(V)$. We consider two kinds of equivalence classes: *terminating* equivalence classes, which represent paths of tuples that end at a given operator (e.g., filtered out), and *binding* equivalence classes, which represent paths through which tuples continue downstream.

For illustration, consider our running example, reproduced here for ease of reference.

```
A = LOAD 'fileA' using PigStorage()
  AS (name:chararray, value:int);
B = LOAD 'fileB' using PigStorage()
  AS (u:double, class:int);
C = FILTER A BY value < 100 AND value >= 0;
D = FILTER B BY math.POW(u,2.0) > 0.25;
E = JOIN C ON value, D ON class;
```

Here, our root node consumes variable E. Our analysis considers E as if it were flowing upstream from a STORE operation. We invent a symbolic name, P, for the (single) constraint induced by the STORE. Its constraint is satisfied by all tuples:

$$\mathcal{C}(E) \supseteq \{P\}, \text{where } \forall t : P(t)$$

Note the use of $\supseteq$. A dataflow node could receive constraints from several downstream neighbors so our constraint inference is using subset reasoning: we know that $\mathcal{C}(E)$ includes at least P, but it could include other constraints as well. (In this example it does not.)

$\mathcal{C}(E)$ is then propagated to the JOIN statement that constructs E. Throughout this section, given a statement Q and its downstream neighbor with constraints P, P' denotes the refined constraints by conjoining P and the new constraints needed to flow a tuple upstream out of the downstream neighbor. JOINs require tuples to agree on particular fields (`value` and `class`, here), so we enforce this property by encoding it via P' in our constraints:

$$\begin{array}{ll} \text{for all} & P \in \mathcal{C}(E) : \exists e_x. \\ & \mathcal{C}(C) \supseteq \{P'_{\texttt{value}}\} \\ & \mathcal{C}(D) \supseteq \{P'_{\texttt{class}}\} \\ & \text{where} \quad P'_{\texttt{value}}(t) \equiv P(t) \wedge t.\texttt{value} = e_x \\ & \text{and} \quad\ P'_{\texttt{class}}(t) \equiv P(t) \wedge t.\texttt{class} = e_x \end{array}$$

("for all $P \in \mathcal{C}(E) : \mathcal{CS}$" here means that we iterate over all P in $\mathcal{C}(E)$ and generate constraints $\mathcal{CS}$ for each such P.)

Continuing the propagation process, we pass the above constraints on to the FILTER operators of our example. For instance, consider the statement D = FILTER ..., which eliminates all elements for which math.POW(u,2.0) > 0.25 does not hold. This statement first introduces a binding equivalence class for the constraints flowing upstream via $\mathcal{C}(D)$. The statement also introduces a single terminating equivalence class ($P_\neg$) to capture the case of tuples that do not pass the

Pig Latin code	Equivalence class constraints	Cardinality constraints
STORE A	$\mathcal{C}(A) \supseteq \{P\}$, where $\forall t : P(t)$	$\#\mathcal{T}(P) \geq 1$
A = FILTER B BY Q	$\mathcal{C}(B) \supseteq \{P_\neg\}$, where $P_\neg(t) \equiv \neg[Q]_b(t)$. for all $P \in \mathcal{C}(A) : \mathcal{C}(B) \supseteq \{P'\}$ where $P'(t) \equiv P(t) \wedge [Q]_b(t)$	$\#\mathcal{T}(P'_\neg) \geq 1$ $\#\mathcal{T}(P') \geq \#\mathcal{T}(P)$
A = UNION B, C	for all $P \in \mathcal{C}(A) : \mathcal{C}(B) \supseteq \{P\}$ and $\mathcal{C}(C) \supseteq \{P\}$	
A = JOIN B BY x, C BY y	for all $P \in \mathcal{C}(A) : \exists A_f . \mathcal{C}(B) \supseteq \{P'_x\}$ and $\mathcal{C}(C) \supseteq \{P'_y\}$ where $P'_x(t) \equiv P(t) \wedge t.x = A_f$ and $P'_y(t) \equiv P(t) \wedge t.y = A_f$	$\#\mathcal{T}(P'_x) \geq \#\mathcal{T}(P)$ $\#\mathcal{T}(P'_y) \geq \#\mathcal{T}(P)$
A = DISTINCT B	for all $P \in \mathcal{C}(A) : \exists A_t . \mathcal{C}(B) \supseteq \{P'\}$ where $P'(t) \equiv P(t) \wedge t \approx A_t$	$\#\mathcal{T}(P') \geq 1 + \#\mathcal{T}(P)$
A = GROUP B BY x	for all $P \in \mathcal{C}(A) : \exists A_f . \mathcal{C}(B) \supseteq \{P'\}$ where $P'(t) \equiv P(t) \wedge t.x = A_f$	$\#\mathcal{T}(P') \geq 1 + \#\mathcal{T}(P)$

Fig. 1. Summary of representative translations from Pig Latin statements into equivalence classes, manifested as constraints. The above constraints are all *binding* constraints, except for the terminating $P_\neg$ in FILTER, and for P' in DISTINCT which is both terminating and binding. In the above, $[\cdot]_b$ translates boolean Pig expressions into our term language, and $\mathcal{T}(P)$ is the set of sample tuples for constraint P. Every rule introduces fresh symbolic names for equivalence classes, we use fresh variables A_f to refer to individual values, and A_t to refer to tuples.

filter:

$\mathcal{C}(B) \supseteq \{P_\neg\}$
where $P_\neg(t) \equiv \neg(\text{math.POW}(t.\text{u},2.0) > 0.25)$
and for all $P \in \mathcal{C}(D)$
$\mathcal{C}(B) \supseteq \{P'\}$
where $P'(t) \equiv P(t) \wedge \text{math.POW}(t.\text{u},2.0) > 0.25$

In our representation, we have preserved the user-defined function math.POW as an example of a function that the theorem prover cannot handle directly (see Section III-B).

We handle the other FILTER statement similarly and reach the LOAD statement which completes the analysis. The resulting $\mathcal{C}$ sets contain symbolic names for all equivalence classes and our symbolic constraints can be used to define members of these classes.

Figure 1 gives the general form of our reasoning for representative constructs (also including DISTINCT statements and cardinality constraints, discussed below). For all operators for which our first two analysis passes observed insufficient coverage, we collect constraints using the above scheme to generate the constraints P that represent each insufficiently covered equivalence class. For each P we attempt to add elements to its corresponding set of samples $\mathcal{T}(P)$. We synthesize such tuples t as follows:

1) Pass P to the theorem prover and query for witnesses for the existentially qualified fields. If there are no witnesses, abort; either the equivalence class is empty/not satisfiable due to conflicting requirements, or the theorem prover lacks the power to synthesize a representative tuple.
2) Otherwise, extract the witnesses into tuple t'.
3) For any field f required by the type constraints over t in P, extract t_2.f from randomly chosen t_2 from our observed samples. Combine t' with all the t_2.f into t''.
4) For any still-missing fields (i.e., if no matching t_2 exists), fill the field with randomly synthesized data, yielding t.
5) If $t \in \mathcal{T}(P)$ already, repeat the previous two steps as needed, otherwise insert t into $\mathcal{T}(P)$.

As the last steps (and Table 1) show, there is another dimension in our sample generation, namely generating the right amount of sample data. Specifically, recall that our binding equivalence class for

```
F = GROUP B BY x
```

requires at least two tuples. To capture this constraint, we permit constraints on the cardinality of our sets of witness tuples, notation $\#\mathcal{T}(P') \geq 2$, where predicate P' represents the binding equivalence class in the above. All such constraints are greater-than-or-equal constraints, and we always pick the minimum cardinality that satisfies all constraints.

Another subtlety of our constraint notation comes from the DISTINCT statement, as in

```
G = DISTINCT B
```

This statement eliminates duplicate tuples. Since set semantics have no notion of duplicates, we extend all of our tuples with a unique *identity* field that does not occur in the Pig program. We write $t_1 \approx t_2$ iff the tuples t_1 and t_2 have the same fields, ignoring the *identity* field.

To support aggregation operations in sample synthesis, we further permit reasoning about our sampled tuples. For example, Pig Latin allows us to write

```
A = LOAD ...
sum = SUM(A.x)
B = FILTER A BY count == sum
```

We translate aggregations such as sum = SUM(A.x) into aggregations over our sets of samples. Whenever we synthesize samples for one of A's binding equivalence classes, e.g., represented by P, we simply set sum $= \sum_{t \in \mathcal{T}(P)} t.\text{x}$. The translation is analogous for other aggregators (AVG, MAX, etc.). Aggregators enforce $\#\mathcal{T}(P) \geq 2$.

B. User-defined Function Concretization

In earlier sections, we classified our approach as *dynamic-symbolic*, following other similar work in different settings [8], [12], [21], [25]. The important aspect of a dynamic-symbolic execution approach to test generation is that dynamic (i.e., concrete) observations are used to help the symbolic solving process. The foremost aspect where this benefit is

```
public class HASH extends EvalFunc<Integer>
    {
  public Integer exec(Tuple input) {
    if (input == null || input.size() == 0)
      return null;
    Integer y = (Integer) input.get(0);
    int hash = y*(y+3);
    return hash % 60;
  } // ..
}
```

Fig. 2. The implementation of function HASH.

apparent in our setting is when dealing with user-defined functions (UDFs). A user-defined function is any side-effect-free operator that has a definition external to the language. In the Pig Latin world, this typically means a Java function used to process values, e.g., in a FILTER. What a dynamic-symbolic execution engine can do is to treat a UDF as a black-box function. Inside a constraint, a use of a UDF is replaced by a set of function values from the concrete semantics, under the assumption that some invocations of UDFs (and return values thereof) have already been observed.

Consider the example Pig Latin program shown in Listing 2. Our objective is to generate complete example data with one tuple passing and one tuple not passing the FILTER. This program's key step is the application of the UDF HASH to perform filtering, which takes an integer as argument and returns its hash value.

```
A = LOAD 'fileA' using PigStorage()
  AS (x:int, y:int);
B = FILTER A BY x == HASH(y) AND x > 50;
```

Listing 2. Example Pig Latin program calling user-defined function HASH.

A simplified implementation of HASH is shown in Figure 2. In this implementation, HASH extends the EvalFunc class (which is required by Pig Latin to construct Java user-defined functions[3]).

Assume that we run the program with two input tuples (33, 42) and (47, 19) that do not pass the FILTER (since 33 and 47 are not the hash values of 42 and 19, respectively). On these two executions, we obtain two evaluations of HASH: y = 42, HASH(y) = 30 and y = 19, HASH(y) = 58. For our technique to have 100% completeness, we need to generate example data for A(x,y) such that x == HASH(y) && x > 50. Using the two evaluations of HASH, we construct two simplified versions of the constraint: x == 30 && y == 42 && x > 50 and x == 58 && y == 19 && x > 50. In the simplified constraints the function call HASH(y) has been *concretized* to the observed values (30 and 58, respectively). The second simplified constraint is satisfiable while the first is not. Using the satisfying assignment, we derive a new example input (58, 19) for A(x,y).

Thus, our approach *records* concrete values for UDFs during the downstream pass, *concretizes* constraints using recorded concrete data, and *solves* them via automatic constraint solvers, in the upstream pass. We use uninterpreted functions to encode a concretized constraint. An uninterpreted function (UF) [5], [7] is a black box with no semantic assumptions other than the obligation that it behave functionally: equal parameters yield equal function values. To encode UDFs as uninterpreted functions for our constraint solver, we supply concrete observations as implications, using the if-then-else operator (*ite*) over boolean formulas and concrete values.

Consider the example of Listing 2 again, in which we need to find assignments to (x,y) to satisfy the constraint x == HASH(y) && x > 50. We supply the constraint solver Z3 with concrete observations on the HASH UDF via the following commands:

```
(declare-const x Int)
(declare-const y Int)
(define-fun HASH ((x!1 Int)) Int
    (ite (= x!1 42) 30
    (ite (= x!1 19) 58
     0)))
(assert (not (= (HASH y) 0)))
(assert (= (HASH y) x))
(assert (> x 50))
```

The first two `declare-const` commands declare two integer variables. The `define-fun` command creates a UF that takes a parameter representing an integer and returns a constant value. x1! is the argument of the UF. We have observed two invocations of the function HASH, HASH applied to y == 42 yields 30, and HASH applied to y == 19 yields 58. To complete the definition of the UF, we need to relate unknown parameter values with a default return value, which in this case we arbitrarily choose to be zero. Still, we assert that HASH(y) is not zero to avoid accidental satisfaction. Finally we provide the constraint x == HASH(y) && x > 50 that we want to solve. Using three `assert` commands, the system pushes three formulas into Z3's internal constraint stack. We solve the concretized constraints by asking Z3 to produce a satisfying assignment for variables in the constraints.

Of course, when the observations of the UDF are not sufficient to obtain the desired coverage, Z3 will deem a concretized constraint to be unsatisfiable or unknown. To increase the chance of finding a satisfying assignment for an abstract constraint, we also try a second constraint solver, CORAL [4], when Z3 returns unsatisfiable or unknown for a concretized constraint. The distinction between Z3 and CORAL concerns the kind of formulas that they can solve: Z3 can derive models and check satisfiability of formulas in decidable theories, while CORAL can deal with numerical constraints involving undecidable theories. As a consequence of supporting undecidable theories, CORAL can solve constraints involving UDFs in the form of common math functions (e.g., power function) directly without concretization. If neither concretization-and-employing-Z3 nor calling CORAL can solve a constraint, SEDGE will be unable to obtain perfect coverage.

Note that our approach to solving UDFs reasons about all observed values of the UDF in parallel. These UDF

[3]See Pig's implementation guide for user-defined functions at http://pig.apache.org/docs/r0.9.2/udf.html

observations may be produced in different paths through the program, including executions of different test cases. Still, the observations can be used together (i.e., we can assume that all of them hold) because of the lack of side-effects in a dataflow program. In contrast, in dynamic-symbolic execution of an imperative program, only values (of user-defined functions) observed during the current dynamic execution can be leveraged at a given constraint solving point.

IV. Implementation

The SEDGE system has required non-trivial implementation effort, in the support of different data types, the interfacing with the Z3 constraint solver, and the integration of string generation capabilities.

A. Symbolic Representation of Values

SEDGE maintains an intermediate level of abstract syntax trees for communication between Z3 and Pig Latin constraints. Each node of the tree denotes a symbolic variable occurring in the Pig Latin constraints. The high-level idea is that SEDGE maps an execution path to a conjunction of arithmetic or string constraints over symbolic variables and constants. Each symbolic variable has a name and a data type, such as int and long, mapping to a field of a table in a Pig Latin script with the same name and data type. SEDGE then invokes Z3 to find a solution to that constraint system. If the constraint solver finds a solution, SEDGE maps it back to input tuples (tuples from LOAD). SEDGE supports mapping all Pig Latin data types into a symbolic variable, with support for overflow and underflow checked arithmetic.

1) int: Integers are represented as 32-bit signed bit-vectors, since the Z3 constraint solver has better support for bit-vector arithmetic than for integer arithmetic. Arithmetic calculations over integers are thus simulated with arithmetic calculations over 32-bit-vectors. The simulation is accurate and takes into account the Java (Apache Pig is written in Java) representation of values of type int as 32-bit-vectors. Additional constraints are created to check that the bit-wise computation does not overflow and underflow. Standard library conversion functions (e.g., *java.lang.Long.parseLong(String)*) are used to translate back from 32-bit-vectors into integers.

2) long: Similar to *int*, long integers are represented by 64-bit signed bit-vectors, since Z3 has no builtin support for long integers.

3) float/double: Floating point numbers are represented by real numbers in the form of fractions of long integers. No current constraint solvers have good support for floating-point arithmetic. Calculations with floating point numbers are thus approximated by real-valued calculations. A real number in the form of fractions of long integers can be translated to a floating point number by first representing the fraction using *BigFraction* from *Apache Common Math Library*,[4] and invoking *BigFraction.floatValue()* (or *BigFraction.doubleValue()*) to get the fraction as a float (or double, respectively).

[4]http://commons.apache.org/math/

4) chararray: A character array is represented by *java.lang.String*, which is also the inner representation of a character array in Pig Latin.

5) bytearray: We do not support byte arrays directly. We try to identify the type that the byte array can convert to at runtime and cast it.

6) boolean: A boolean variable is represented by an integer with 3 values: -1 for *FALSE*; 0 for *UNDEF*; 1 for *TRUE*.

Arithmetic and string constraints are typically expressed over fields of simple types as listed above. Therefore, we do not define complex types (tuple, bag, map) for symbolic variables.

B. Arithmetic and String Constraint Solving

As mentioned earlier, SEDGE uses Z3 [10] to solve arithmetic constraints. Since Z3 provides a C interface and SEDGE is implemented using Java, to have access to Z3's C API from Java, we employ SWIG.[5] We wrap Z3's C API using Java proxy classes and generate JNI [16] wrapper code automatically. A problem in wrapping C programs for Java is that values may be returned in function parameters in C, but in Java values are typically returned in the return value of a function. SEDGE uses typemaps in SWIG, a code generation rule that is attached to a specific C data type, to overcome the problem. Given the data type *D* of a value returned in function parameters, SEDGE constructs a structure *S* containing a member variable of type *D*. It also registers a typemap such that 1) any occurrence of a function parameter of type *D* in a function call in Z3 is converted into *S*, 2) the return parameter *S*'s value can be read after returning from the function in Java.

For string constraints, the main new element of our implementation concerns reasoning about string constraints containing regular expressions. Our approach is based on Xeger[6] a Java library for generating a sample string for a regular expression. Xeger builds a deterministic finite automaton (DFA) for a string constraint in the form of a regular expression, and follows the edges of the DFA probabilistically, until it arrives at an accepting state of the DFA. Xeger is suboptimal for two reasons: first, it may keep visiting the same state until a "stack overflow" error happens; second, it does not support union, concatenated repetition, intersection, concatenation, or complement of regular expressions. To avoid the "stack overflow" error, our approach keeps a map from state ID to the number of times a state has been entered and reduces the probability of re-entering that state proportionally. To support operations such as union and intersection, we add an intermediate step between building the DFA and following DFA edges to return a new deterministic automaton for the appropriate regular expression. For example, when generating a satisfying assignment that maps a string to a value so that the constraint matches '.*apache.*' and '.*commons.*', we intersect the automata representing the strings.

[5]http://www.swig.org/

[6]http://code.google.com/p/xeger/

V. EVALUATION

In this section, we evaluate the implementation of SEDGE by running a wide spectrum of actual Pig Latin programs. We measured both the completeness of generated example tuples and the run-time of example generation for SEDGE and for the current state-of-the-art: the original Pig example data generator (abbreviated to "Olston's system" in our discussion). Compared to Olston's system, our experiments confirm that SEDGE achieves higher completeness. In most experiments, SEDGE also incurs a lower running time.

A. Benchmark Programs

To evaluate our system, we applied it to two benchmark suites:

- We use the entirety of the *PigMix* benchmark suite, consisting of 20 Pig programs designed to model practical Pig problems.
- We use eleven sample SQL queries (the first ten in the list and an 11-th selected for being complex) from the Sloan Digital Sky Survey (SDSS) set[7] and hand-translated them directly into Pig code. The complex query contains 34 FILTER operations and 2 JOIN operations.

The PigMix benchmark provides Pig Latin programs for testing a set of features such as "data with many fields, but only a few are used" and "merge join". The SDSS sample queries typically search for an astronomical object based on some criteria. For example, Program 5 of the SDSS set is below:

```
A = LOAD Galaxy3 using PigStorage()
  AS (colc_g : float, colc_r : float,
      cx : float, cy : float);
B = FILTER A BY
    (-0.642788 * cx + 0.766044 * cy >=0.0)
  AND (-0.984808 * cx - 0.173648 * cy <0.0);
```

Listing 3. SDSS program 5

The program finds galaxies in a given area of the sky, using a coordinate cut in the unit vector cx, cy, cz. As can be seen, these benchmark programs are typically short, with only a handful of them exhibiting interesting complexity.

For each query set, we used the input data that accompany the relevant queries. PigMix ships with a tuple synthesizer that generates such data. The SDSS benchmark suite is designed for the digital sky survey data from the SDSS data release 7. We selected a random sampling of tuples from the database of this benchmark, in which the total amount of data is 818 GB, and the total number of rows exceeds 3.4 billion.

B. Methodology and Setup

Since the importance of tuple synthesis varies not only by benchmark but also by the size of the tuples supplied to the first analysis pass (in the ideal case, tuple synthesis is entirely unnecessary), we ran our benchmarks for sample input tuple sizes of 10, 30, 100, 300, and 1000 tuples using our system and Olston's system. For each sample input tuple size, we executed each benchmark program 10 times with different randomly sampled input tuples. All experiments were performed on a four core 2.4 GHz machine with 6 GB of RAM.

We configured our system to compare directly to Olston's system, which is implemented as the "illustrate" command in Pig Latin. Unfortunately, the current implementation of Olston's system has some limitations not mentioned in the published paper. We wanted to evaluate against the approach and not against the implementation. To that end, we addressed such limitations or tweaked the benchmark programs so that the problems do not manifest themselves. The first issue is that the downstream pass would discard all sample input containing null fields. In the upstream pass, if all input tuples happen to contain null fields and thus all of them are discarded, there would be a *NullPointerException*. We sidestep the null field issue by ensuring that at least one sample input does not contain null fields. In addition, the system can only handle 32 FILTER conditions at most, as it encodes pertinent equivalence classes for FILTER conditions as individual bits in a (32-bit) integer index variable. This problem affects one PigMix benchmark program intended for scalability testing. The benchmark has a very large set (500+) of FILTER conditions. We sidestepped the problem by making changes to the PigMix program so that the resulting program has only 31 FILTER conditions. Also, when reasoning about a JOIN in the upstream pass, a *NullPointerException* is thrown if no data are observed in the input side of a JOIN (typically because one of its upstream neighbors is a highly selective operator). We address this issue by skipping the JOIN if its input has no data and then attempt to continue upstream propagation. Moreover, by mistakenly setting a non-tuple field to a tuple in a method involved in upstream propagation, a type casting error arises, which impedes the ability of Olston's system to reason over the JOIN and FOREACH operations if their downstream neighbor is a FILTER operator. We disallow assigning the non-tuple field to a tuple in the problematic method.

C. Results

We ran each experiment 10 times and averaged the completeness of 10 runs (since the completeness may theoretically vary due to different random choice of initial samples). The size of the input data has little effect. The results are almost the same for all sample input sizes.

Figures 3 and Figure 4 show the average completeness for each Pig Latin program in the PigMix and SDSS sets, respectively, for a sample input size of 100 tuples. Every bar corresponds to one program (with the exception of program *L12* in Figure 3, in which there are three subprograms and example data were generated for three different root operators corresponding to the three subprograms) for a total of 20 programs.

As can be seen, we improve on completeness for 5 out of 20 PigMix benchmark programs and 7 out of 11 SDSS benchmark programs. Although the benchmark programs are small and much of their coverage is achieved with random sampling of real inputs, they demonstrate clearly the benefits of our

[7] http://skyserver.sdss.org/public/en/help/docs/realquery.asp

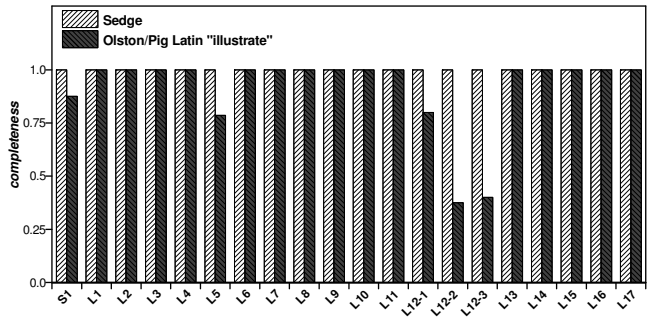

Fig. 3. Completeness of sample data generation for the PigMix benchmarks

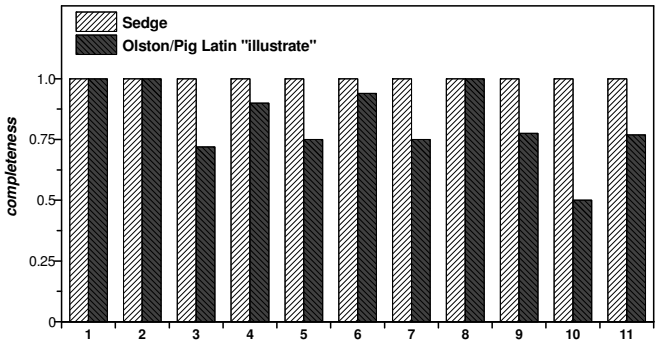

Fig. 4. Completeness of sample data generation for the SDSS benchmarks

approach. *Practically every program in the two benchmark sets that has any kind of complexity (either more than one operator in the same path, or a user-defined function, or complex filter conditions) is not fully covered by Olston's approach.* For example, Olston's system cannot generate data that fail the FILTER in the presence of grouping, projecting, UDF invocation in the following program (program *S1* in Figure 3).

```
A = LOAD '$widerow' using PigStorage()
  AS (name: chararray, c0: int, c1: int,
      ..., c31: int);
B = GROUP A BY name;
C = FOREACH B GENERATE group, SUM(A.c0) as
    c0, SUM(A.c1) as c1, ..., SUM(A.c31) as
    c500;
D = FILTER C BY c0 > 100 AND c1 > 100 AND c2
     > 100 ... AND c31 > 100;
```

Listing 4. PigMix program S1

In fact, SEDGE achieves perfect coverage (i.e., full completeness) for all benchmark programs. Compared to Olston's approach our improved coverage is due to stronger constraint solving ability (for programs *4,5,6,7* in Figure 4), to UDF handling ability (for programs *9,10* in Figure 4) and also to inter-related constraints and global reasoning (for programs *S1,L5,L12-1,L12-2,L12-3* in Figure 3 and programs *3,11* in Figure 4).

We also recorded how long it took SEDGE and Olston's system to finish example generation. We include the infrastructure bootstrap time on each benchmark program. Both SEDGE and Olston's system need to prepare the Hadoop execution environment for new executions. SEDGE needs to load its constraint solver Z3 and CORAL as well.

As can be seen in Figure 5 and Figure 6, SEDGE is faster on average than Olston's system in 18 out of 20 PigMix benchmark programs and 9 out of 11 SDSS benchmark programs. For the rest of benchmark programs, SEDGE incurs a little higher running time than Olston's system. From these numbers we can infer that, although we have to conduct path exploration and constraint solving, there are even time savings in most cases due to avoiding the step of pruning redundant tuples after the upstream pass (because our approach does not generate redundant data).

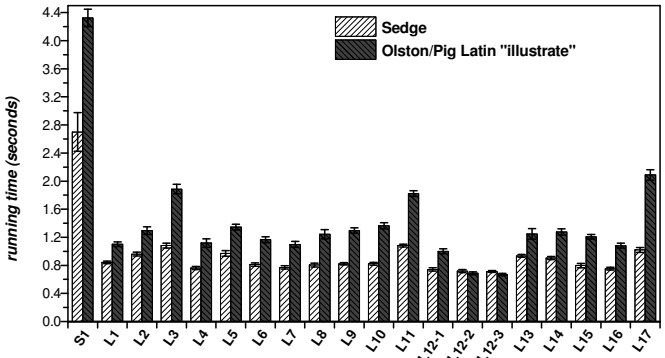

Fig. 5. Running time of sample data generation for the PigMix benchmarks

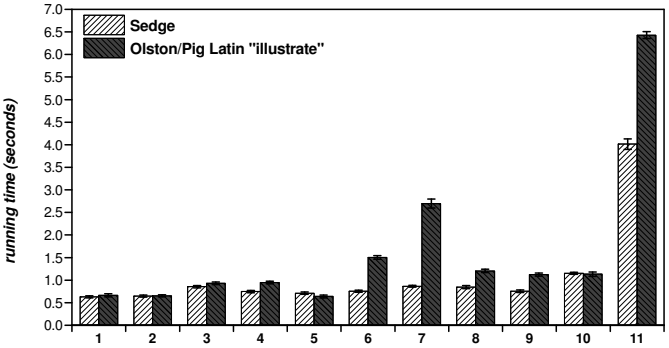

Fig. 6. Running time of sample data generation for the SDSS benchmarks

VI. DISCUSSION: WHY HIGH-LEVEL DSE

A natural qualitative comparison is between a Dynamic Symbolic Execution (DSE) engine at the level of the Pig Latin language and DSE engines for imperative languages, since Pig Latin code is eventually compiled into imperative code that uses a map-reduce library. The expected benefits from our approach are a) simplicity; b) conciseness of the generated test cases (i.e., the same coverage with fewer tests); and c) completeness: an imperative DSE engine may have trouble solving constraints over the logically more complex generated code, rather than the original Pig Latin code. Furthermore, an imperative DSE engine cannot take advantage of the lack of side-effects in order to better concretize user-defined functions, as discussed in Section III-B.

We compared SEDGE with the Pex [25] state-of-the-art DSE engine in a limit study. Pex accepts C# input, hence we hand-

translated Pig Latin programs into C# programs.[8] The resulting C# programs are single-threaded without any call to the map-reduce API, in order to test the applicability of Pex in the ideal case. (The inclusion of the map-reduce library complicates the control-flow of the imperative program even more and can easily cause the DSE engine to miss a targeted branch of test execution, leading to low coverage of generated test cases [28].)

For our translation, we inspected the Java code generated by the Pig compiler and made a best-effort attempt to replicate it in C#, without map-reduce calls. We translated 13 programs from our Pig Latin benchmark suites. Since Pex has no knowledge of the original input (it accepts concrete values only when passed into the test method as parameters with primitive types) we enable just the 3rd pass (upstream pass) of SEDGE, for a fair comparison (i.e., SEDGE also does not benefit from sampled real data—this also disadvantages SEDGE as it removes the advantage of better UDF handling).

The results confirm our expectation. The conciseness of the test suite generated by Pex is low since Pex needs to examine a lot of irrelevant low-level branches or constraints that are not necessary for equivalence class coverage of the high-level Pig Latin control flow. For example, for step A in the Pig Latin program in Listing 3, Pex generates 30 tuples within 11 tables, of which 3 tuples pass the filter in step B, while SEDGE generates 2 tuples within exactly 1 table, of which 1 tuple passes the filter in step B. The conciseness of the test suite generated by Pex is 0.05, while the conciseness of the test suite generated by SEDGE is 0.75. Furthermore, for specific complex constructs we also get much higher completeness, although quite often Pex also gets perfect coverage. In our experience, for Pig Latin programs containing FILTER statements after (CO)GROUP or JOIN statements, the test suites yielded by Pex lack in completeness. For instance, in the SDSS program with 34 FILTER operations and 2 JOIN operations, the Pex completeness is only 0.09.

VII. RELATED WORK

Dataflow languages such as Pig can been seen as a compromise between declarative languages, such as SQL, and imperative languages, such as C and Java. That is, Pig combines the declarative feature of straightforward parallel computation with the imperative feature of explicit intermediate results. There is little work (discussed in earlier sections) that addresses test data generation for dataflow languages. Instead, the related work from various research communities has focused on the extreme ends of this spectrum, i.e., either on SQL or on Java-like programming languages.

Specifically, related work in the software engineering community has focused on traditional procedural and object-oriented database-centric programs, tested via combinations of static and dynamic reasoning [22]. The main approaches use static symbolic execution [17] or dynamic symbolic execution [11], [15], [20]. While our work is inspired by such earlier dynamic symbolic execution approaches, we adapted this work to dataflow programs and their execution semantics. At the other end, there is work that automatically generates database data that satisfy external constraints [23] but there is no coverage or conciseness goal and no application to dataflow languages. Other work [26] has introduced the idea of code coverage to SQL queries. For our purposes, we reused the concept of coverage for Pig Latin as defined by Olston et al. [18].

In the formal methods community, Qex is generating test inputs for SQL queries [27]. Similar to our work, Qex maps a SQL query to SMT and uses the Z3 constraint solver to infer data tables. However Qex differs from our work in that Qex does not have a dynamic program analysis component and therefore cannot observe how a query processes existing example data. Earlier work in the software engineering community on dynamic symbolic execution has shown that dynamic analysis can make such program analysis more efficient and enable it to reason about user-defined functions, which we leverage in our work.

In the database community, a common methodology for testing a database management system or a database application is to generate a set of test databases given target query workloads. Overall our problem differs in that, instead of a whole database, we aim to generate a small (or minimum if desired) set of tuples that have perfect path coverage of a given dataflow program. The recent work on reverse query processing [2] takes an application query and a result set as input, and generates a corresponding input database by exploiting *reverse relational algebra*. In comparison, our work focuses on dataflow programs for big data applications, where many operators are non-relational, e.g., map(), reduce(), and arbitrary user-defined functions, and hence a "reverse algebra" may not exist. The QAGen system [3] further takes into account a set of constraints, usually cardinality and data distribution in input and operator output tables, and aims to generate a database that satisfies these constraints. Analogously to earlier work in the formal methods community, this work performs a static symbolic analysis and does not obtain additional information from a dynamic analysis.

VIII. CONCLUSIONS AND FUTURE WORK

Generating example input data for dataflow programs has emerged as an important challenge. We presented SEDGE: an approach and tool for generating example data of dataflow programs using DSE, in order to achieve high coverage. SEDGE builds symbolic constraints over the equivalence classes induced by dataflow programming language constructs and can reason over constraints on user-defined functions by exploiting dynamic values as hints. We implemented our technique for the Pig dataflow system and compared it empirically with the most closely related prior work. While we currently focus on the Pig Latin programming language, the principles are quite general. The same high-level technique can be applied to other

[8] Although there are DSE engines for Java—e.g., Dsc [14]—they do not match the industrial-strength nature of Pex. Dsc, for instance, does not support programs with floating point numbers, which are common in Pig Latin.

dataflow programming languages, such as DryadLINQ [13] and Hyracks/Asterix [1], and to relational algebra primitives, such as relational division and anti-join. Our evaluation on third-party applications demonstrates that, with similar computing resources, our technique achieves better coverage of a given dataflow program. We described the first example input data generator for dataflow programs that leverages dynamic symbolic program analysis.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grants No. 1017305, 1117369, and IIS-1218524, and funds from the UMass Science & Technology Program. We also gratefully acknowledge funding by the European Union under a Marie Curie International Reintegration Grant and a European Research Council Starting/Consolidator grant; and by the Greek Secretariat for Research and Technology under an Excellence (Aristeia) award.

Our tool is available at: https://github.com/kaituo/sedge.

REFERENCES

[1] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[2] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *Proc. 23rd International Conference on Data Engineering (ICDE)*, pages 506–515. IEEE, Apr. 2007.

[3] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. QAGen: Generating query-aware test databases. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 341–352. ACM, June 2007.

[4] M. Borges, M. d'Amorim, S. Anand, D. Bushnell, and C. S. Pasareanu. Symbolic execution with interval solving and meta-heuristic search. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 111–120, Washington, DC, USA, 2012. IEEE Computer Society.

[5] R. E. Bryant, S. M. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proc. 11th International Conference on Computer Aided Verification (CAV)*, pages 470–482. Springer, 1999.

[6] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. 6th International Conference on Computer Aided Verification (CAV)*, pages 68–80. Springer, 1994.

[7] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *Proc. 12th International SPIN Workshop on Model Checking of Software*, pages 2–23. Springer, Aug. 2005.

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

[9] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 151–162. ACM, July 2007.

[10] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, June 2005.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, pages 59–72. ACM, 2007.

[12] M. Islam and C. Csallner. Dsc+mock: a test case + mock class generator in support of coding against interfaces. In *Proceedings of the Eighth International Workshop on Dynamic Analysis*, WODA '10, pages 26–31, New York, NY, USA, 2010. ACM.

[13] C. Li and C. Csallner. Dynamic symbolic database application testing. In *Proc. 3rd International Workshop on Testing Database Systems (DBTest)*. ACM, June 2010.

[14] S. Liang. *Java Native Interface: Programmer's Guide and Specification*. Prentice Hall, June 1999.

[15] M. Marcozzi, W. Vanhoof, and J.-L. Hainaut. Test input generation for database programs using relational constraints. In *Proc. 5th International Workshop on Testing Database Systems (DBTest)*. ACM, May 2012.

[16] C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *Proc. 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 245–256. ACM, 2009.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1099–1110. ACM, 2008.

[18] 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)*, pages 73–82. IEEE, Nov. 2011.

[19] K. Sen and G. Agha. Cute and jCute: Concolic unit testing and explicit path model-checking tools. In *Proc. 18th International Conference on Computer Aided Verification (CAV)*, pages 419–423. Springer, Aug. 2006.

[20] Y. Smaragdakis and C. Csallner. Combining static and dynamic reasoning for bug detection. In Y. Gurevich and B. Meyer, editors, *TAP*, volume 4454 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2007.

[21] Y. Smaragdakis, C. Csallner, and R. Subramanian. Scalable automatic test data generation from modeling diagrams. In *Automated Software Engineering conference (ASE)*, pages 4–13. ACM Press, Nov. 2007.

[22] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[23] N. Tillmann and J. De Halleux. Pex: White box test generation for .Net. In *Proc. 2nd International Conference on Tests and Proofs (TAP)*, pages 134–153. Springer, 2008.

[24] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva. Full predicate coverage for testing SQL database queries. *Software Testing, Verification & Reliability (STVR)*, 20(3):237–288, Sept. 2010.

[25] M. Veanes, N. Tillmann, and J. de Halleux. Qex: Symbolic SQL query explorer. In *Proc. 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 425–446. Springer, Apr. 2010.

[26] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 611–620, New York, NY, USA, 2011. ACM.