SEDGE: Symbolic Example Data Generation for Dataflow Programs
Motivation
**Problem**: Generate **fewest** test cases to exercise **all** key behaviors of operators in a dataflow program

- e.g. both passing and failing a filter

**Our approach**: First dynamic-symbolic (aka “concolic”) testing engine for a dataflow language

**Results**:
- Improved coverage and running time over industrial state-of-the-art
  - e.g. Pig Latin “illustrate”, SIGMOD ’09 best paper award
Overview of Talk

- Background: Dataflow languages
- Metrics
- Our algorithm in action
- Comparison with state-of-the-art
Example Dataflow Program

- Input: Galaxy profiles, Star profiles
- Find stars with a surface brightness less than 100 from galaxies with a squared ellipticity > 0.25
  - \(\text{ellipticity}^2 = g_r^2 + u_r^2\)

![Dataflow Diagram]

1. **Load** `Star(brightness, gid)`
2. **Filter** `brightness < 100`
3. **Join** `id = gid`
4. **Load** `Galaxy(id, g_r, u_r)`
5. **Filter** `ellipticity^2 > 0.25`
A=LOAD 'Star' using PigStorage AS (brightness:int, gid:int);
B=LOAD 'Galaxy' using PigStorage AS (id: int, g_r, u_r:double);
C=FILTER A BY brightness < 100;
D=FILTER B BY power(g_r, 2) + power(u_r, 2) > 0.25;
E=JOIN C ON gid, D ON id;
Goal: Coverage with Fewest Tests

- Completeness
  - similar to “branch coverage” in imperative languages
- Conciseness
  - with as few tests as possible
**But What Is Coverage?**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>Input is not empty.</td>
</tr>
<tr>
<td>FILTER</td>
<td>(a) one record that passes the filter; (b) one that does not pass.</td>
</tr>
<tr>
<td>JOIN</td>
<td>Output is not empty.</td>
</tr>
</tbody>
</table>

**Diagram:**
- **Load Star(brightness, gid)**
- **Filter brightness < 100**
- **Load Galaxy(id, g_r, u_r)**
- **Filter ellipticity² > 0.25**
- **Join id = gid**
Fewest tests

- Conciseness = \frac{\# \text{ coverage requirements}}{\# \text{ test cases}}
Our Algorithm

- Dynamic execution
- Pruning
- Symbolic synthesis
Our Algorithm

Dynamic execution

Pruning

Symbolic synthesis

Run sampled existing data

Cache concrete results of user-defined functions (UDF) across runs (if any)
Dynamic Execution

<table>
<thead>
<tr>
<th>UDF</th>
<th>Parameters</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>power</td>
<td>(0.6, 2)</td>
<td>0.36</td>
</tr>
<tr>
<td>power</td>
<td>(0.1, 2)</td>
<td>0.01</td>
</tr>
<tr>
<td>power</td>
<td>(0.2, 2)</td>
<td>0.04</td>
</tr>
<tr>
<td>power</td>
<td>(0.5, 2)</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Load Star(brightness, gid)
Load Galaxy(id, g_r, u_r)
Filter brightness <100
Filter ellipticity^2 > 0.25
Join id = gid
Our Algorithm

- Dynamic execution
- Pruning
- Symbolic synthesis
- Eliminate redundancy
Our Algorithm

1. Dynamic execution
2. Pruning
3. Symbolic synthesis

- Backward symbolic execution to remedy incompleteness
- Concretize UDF in symbolic constraints
- Solve symbolic constraints via constraint solvers
Load Star(brightness, gid)
Load Galaxy(id, g_r, u_r)
Filter brightness < 100
Filter ellipticity^2 > 0.25
Join id = gid

 gid==id && id==4

P
Backward Symbolic Execution

Load `Star(brightness, gid)`

Filter `brightness < 100`

Join `id = gid`

Load `Galaxy(id, g_r, u_r)`

Filter `ellipticity^2 > 0.25`

Filter `ellipticity^2 <= 0.25`

P

F

gid == id && id == 4

ellipticity^2 <= 0.25

(155, 2)

(4, 0.6, 0.1)
Load Star(brightness, gid)

Load Galaxy(id, g_r, u_r)

Filter brightness < 100

Filter ellipticity^2 > 0.25

Join id = gid

Filter ellipticity^2 <= 0.25

P gid==id && id==4

ellipticity^2 <= 0.25

(155, 2)

(4, 0.6, 0.1)
Constraint Generation

Load Star(brightness, gid)

Filter brightness < 100

Join id = gid

Load Galaxy(id, g_r, u_r)

Filter ellipticity² > 0.25

(4, 0.6, 0.1)

(155, 2)

P: power(g_r, 2) + power(u_r, 2) <= 0.25

F: ellipticity² <= 0.25

gid==id && id==4
Backward Symbolic Execution

Load Star(brightness, gid)

Filter brightness < 100

Filter ellipticity^2 > 0.25

Join id = gid

Load Galaxy(id, g_r, u_r)

Filter ellipticity^2 <= 0.25

P

Fresh
Backward Symbolic Execution

Load Star(brightness, gid)

Load Galaxy(id, g_r, u_r)

Filter brightness < 100

Filter ellipticity^2 > 0.25

Join id = gid

Filter brightness < 100

gid == id && id == 4

ellipticity^2 <= 0.25
Constraint Generation

P2: gid==id && id==4 && brightness<100

Load Star(brightness, gid)

Filter brightness <100

Join id = gid

Load Galaxy(id, g_r, u_r)

Filter ellipticity^2 > 0.25

(4, 0.6, 0.1)

(155, 2)

brightens<100

P
gid==id && id==4

ellipticity^2<=0.25

F
Concretization

P1: \( \text{power}(g_r, 2) + \text{power}(u_r, 2) \leq 0.25 \)

P2: \( \text{gid} == \text{id} \) && \( \text{id} == 4 \) && \( \text{brightness} < 100 \)

- Load Star\((\text{brightness}, \text{gid})\)
- Filter brightness < 100
- Join \( \text{id} = \text{gid} \)
- Load Galaxy\((\text{id}, g_r, u_r)\)
- Filter \( \text{ellipticity}^2 > 0.25 \)
- (4, 0.6, 0.1)
**Concretization**

Model the behavior of a user-defined function as its input-output behavior across all observed test cases.

<table>
<thead>
<tr>
<th>UDF</th>
<th>Parameters</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>power</td>
<td>(0.6, 2)</td>
<td>0.36</td>
</tr>
<tr>
<td>power</td>
<td>(0.1, 2)</td>
<td>0.01</td>
</tr>
<tr>
<td>power</td>
<td>(0.2, 2)</td>
<td>0.04</td>
</tr>
<tr>
<td>power</td>
<td>(0.5, 2)</td>
<td>0.25</td>
</tr>
</tbody>
</table>

**Uninterpreted function**

```
(define-fun power ((u Double) (v Int)) Double
    (ite (and (= u 0.6) (= v 2)) 0.36
         (ite (and (= u 0.1) (= v 2)) 0.01
              (ite (and (= u 0.2) (= v 2)) 0.04
                   (ite (and (= u 0.5) (= v 2)) 0.25
                        0))))
(declare-const x Double)
(declare-const y Int)
(assert (not (= (power x y) 0)))
```

**Assumption**

UDFs return values only depend on argument value(s) in dataflow programs.
Concretization

P1: gid==id && id==4 && brightness<100

P2: power(g_r, 2) + power(u_r, 2) <=0.25

(155, 2)

Load
Star(brightness, gid)

Filter
brightness <100

Load
Galaxy(id, g_r, u_r)

Filter
ellipticity^2 > 0.25

(4, 0.6, 0.1)

Join
id = gid

(4, 0.6, 0.1)

(define-fun power ((u Double) (v Int)) Double
(ite (and (= u 0.6) (= v 2)) 0.36
(ite (and (= u 0.1) (= v 2)) 0.01
(ite (and (= u 0.2) (= v 2)) 0.04
(ite (and (= u 0.5) (= v 2)) 0.25
0))))

(declare-const x Double)
(declare-const y Int)
(assert (not (= (power x y) 0)))
Constraint Solving

\[ P_1: \text{gid}==\text{id} \land \text{id}==4 \land \text{brightness}<100 \]

\[ P_2: \text{power}(\text{g}_r, 2) + \text{power}(\text{u}_r, 2) \leq 0.25 \]

\[
\text{(define-fun} \quad \text{power} ((\text{u} \text{ Double}) (\text{v} \text{ Int})) \text{ Double} \\
\quad \text{(ite}} \quad \text{(and}} \quad (= \text{u 0.6}) (= \text{v 2}) 0.36 \\
\quad \text{(ite}} \quad \text{(and}} \quad (= \text{u 0.1}) (= \text{v 2}) 0.01 \\
\quad \text{(ite}} \quad \text{(and}} \quad (= \text{u 0.2}) (= \text{v 2}) 0.04 \\
\quad \text{(ite}} \quad \text{(and}} \quad (= \text{u 0.5}) (= \text{v 2}) 0.25 \\
\quad \text{0)))} \\
\quad \text{(declare-const} \ x \text{ Double}) \\
\quad \text{(declare-const} \ y \text{ Int}) \\
\quad \text{(assert} \quad \text{(not}} \quad (= \text{(power} \ x \ y) 0)))
\]
Experiments

- Two benchmark suites
  - PigMix: 20 representative Pig programs from Pig community
  - SDSS: 11 Pig programs hand-translated from sample SQL queries
    - from the Sloan Digital Sky Survey (astronomy DB)
Pig Latin Illustrate

- Cannot handle UDFs
- Poor constraint solving ability
- Generate local constraint without looking ahead

SEDGE

- Support UDFs
- Stronger constraint solving
- Generate inter-related constraints

Pig Latin Illustrate is the current state-of-the-art in example data generation for dataflow programs.

- Industrial tool (“illustrate” functionality in Pig, by Yahoo)
- SIGMOD ’09 best paper

SEDGE can generate example data for dataflow programs better (completeness) and cheaper (running time)
Completeness Comparison Achieved for SDEG Benchmark

- **Supporting UDF (3, 11)**
- **Stronger constraint solving (4, 5, 6, 7)**
- **Generating inter-related constraints (9, 10)**

![Bar chart showing completeness comparison between SEDGE and illustrate](chart.png)
Completeness Comparison Achieved for BigMix Benchmark

generating inter-related constraints (S1, L5, L12-1, L12-2, L12-3)
Timing Comparison for SDSS Benchmark

SEDGE is faster in 9 out of 11 programs.
Timing Comparison for PigMix Benchmark

SEDGE is faster in 18 out of 20 programs.
Summary

- Created the first dynamic-symbolic (aka “concolic”) testing engine for dataflow languages
- Suggested the use of concrete results across runs of a UDF to represent the UDF
- Proposed an approach that balances completeness, conciseness, and running time