Dynamic Symbolic Database
Application Testing

Chengkai Li, Christoph Csallner

Presenter: Aditya Mone
Overview

• Database Application vs Non-database Application
• Compatibility with existing data
• Systematic testing with mock database
• Could we do something different (and possibly better) ?
  ▪ Use existing data
  ▪ Learn from every execution
Existing Techniques

• Brute Force
  – Too costly

• Sampling the existing database
  – May not satisfy maximizing code coverage objective

• Generate custom mock database
  – Some times using real data is important

• Static Analysis
  – May give us false alarms
Agenda

• Introduction
• Motivation
• Assumption(s)
• Problem Statement
• Proposed Solution
• Implementation Details
• Future Work
Introduction

• Why Test?

• How important is code coverage?

• What is different with database application(s)?
  ▪ Different Tuples different Paths

• Idea:
  ▪ Analyze the path(s) covered by tuples corresponding to a query
  ▪ Invert a constraint to generate more queries
  ▪ More queries more paths covered

• Main-Advantages:
  ▪ Test against real data
  ▪ Side step mock database
Motivation

- Compatibility with existing data
  - Upgrade
  - New application development

- Static nature of data (append only)

- Sample Function:

```java
public void var (int x) {
    int z = -x;
    if ( z > 0 ) {
        if ( z < 100 ) //c1
            if ( z < 100 ) //c2
                // ..
    }
}
```
Motivation(2)

- Non-database function

```java
public void food (int [] arr) {
    // "app"
    for (int x: arr)
        bar (x);
}
```

- Function within a database application

```java
public void dbfoo (String q) {
    // "db app"
    query = "select * From r Where " +q;
    tuples = db.execute (query);
    for (Tuple t: tuples ) {
        int x = t.getValue (1);
        bar (x);
    }
}
```
Motivation (3)

• Program input as a query

• Challenge: Infinite set of possible program inputs

  • Exhaustive searching not possible
    ▪ Resource crunch
    ▪ Top-k search interface
    ▪ Maximum query limit
    ▪ Constrained query interface

• Existing Database Sampling Inadequate

• Key Idea: Leverage dynamic symbolic execution
Assumption(s)

Queries:
• Single relation conjunctive selection query
• Each conjunct is $a \Theta v$, where $a$ is an attribute, $v$ is a constant value, and $\Theta$ can be $<$, $\leq$, $>$, $\geq$, $=$ or $\neq$
• No complex queries (e.g., Aggregation, Join)

Program:
• Follow tuple-wise semantics
• If branching condition depends on database tuple, the condition can be re-written to the same form of the query conjuncts: $a \Theta v$
Problem Statement

Terminology:
- Program = set of paths
- Path (P) = path from program root to the leaf
- Path (P, R) = set of paths reachable given database R
- Path (P, R, M) = paths covered by the test method
  Path(P,R,M) ⊆ Path(P,R) ⊆ Path(P)

Aim:
To design a test method M that chooses test queries Q = {q_i} such that |Path(P, R, M)| is maximized while minimizing \( \sum Cost (q_i) \)
Solution Overview

After the initial query:
\[ q_1 = c_1 \cap c_2 \]

```
if (z > 0) {       // c1
    if (z < 100)   // c2
        // ..
}
```
After the query q1 the dashed queries represent the inversed branching condition and hence a candidate query.
Solution Overview (3)

The second test query:

\[ q_2 = \neg c_1 \]
Solution Overview (4)

After the second query the candidate queries are dashed again.
Solution Overview (5)

The third test query:
\[ q_3 = !c_1 \cap c_6 \cap c_7 \]
Solution Overview (6)

After the third test query:

\[ q_3 = \neg c_1 \cap c_6 \cap c_7 \]
Algorithm

1: \( q \leftarrow \text{define an initial test query}; \ Q \leftarrow \{ q \} \)
2: \textbf{repeat}
3: \( T \leftarrow \text{run } q \text{ and get the first } n_q \text{ result tuples} \)
4: \textbf{for} each tuple \( t \) in \( T \) \textbf{do}
5: \hspace{1em} \text{run the program over } t \text{ and update the execution tree } \text{tree}_Q \text{ with encountered new execution paths}
6: \hspace{1em} \text{tree}_Q \leftarrow \text{the complement tree of } \text{tree}_Q \hspace{1em} (\text{incorrect without } tree_Q)
7: \hspace{1em} Q_c \leftarrow \text{get the candidate queries based on } \text{tree}_Q \hspace{1em} (\text{incorrect without } \text{tree}_Q)
8: \hspace{1em} q \leftarrow \text{select a query from } Q_c \hspace{1em} (\text{incorrect without } Q_c)
9: \hspace{1em} Q \leftarrow Q \cup \{ q \} \hspace{1em} (\text{incorrect without } Q)
10: \textbf{until} stopping criteria satisfied
Details

- Execution Tree
- Complement Tree

Candidate Queries:
“All the queries that can potentially be executed as the next query \( q_{k+1} \) are termed as candidate queries.”

Possible Candidate queries:
- A leaf complement node
- A Queried node
- A non-queried internal node

~ Candidate queries
- An existing leaf node
- An internal node with no complement descendent
Details (2) - Select Next Query

- Rank candidates to minimize score

- Score function:

\[ \text{score}(q) = \frac{\text{cost}(q)}{|\text{Path}'(P,R,M,Q \cup \{q\})| - |\text{Path}(P,R,M,Q)|} \]

Here, Path’(P,R,M,Q U {q}) = Path estimate and cost (q) is the cost of using a particular query q

- Path and Cost are monotonic

- Running example for estimation:

```
          c1 50
         /   \
        c2 10 c2
```

\( c_1 \) 50
\( c_2 \) 10
\( !c_2 \) 40
Details (3) - Optimization function

- Given a program $P$ and a set of test queries $Q = \{q_i\}$
- Maximize coverage:
  - $\text{Path}(P, R, Q) = \{ \text{Path}_t \mid t \in U_{T_i} \}$, where $T_i$ is the first $n_i$ tuples for the query $q_i$

- Minimize cost:
  - $\text{cost}(Q) = \sum \text{cost}(q_i)$
  - $\text{cost}(q_i) = q\_\text{cost}(q_i) + t\_\text{cost}(q_i) = w + c \times n_i + t \times n_i$
  
Where $q\_\text{cost}$: $w$ is the query cost to get the first tuple, $c$ is the cost to get each additional tuple.
  
$t\_\text{cost}$: $t$ is the test cost per tuple
Details (4) – Stopping Criteria

- Testing resource limit reached
- No more candidate queries
- No candidate query can return non-empty result
- Total number of encountered tuples equals tuple size
Implementation

• Use Java 5 instrumentation facilities
• Use third-party open source bytecode instrumentation framework ASM
• Implement on top of new dynamic symbolic engine Dsc
• Dsc allows handling regular (non-query) program inputs
• Solve constraints on regular program inputs with powerful third-party satisfiability modulo theories (SMT) constraint solver Z3.
Future Work

- Implementation still in progress (as when the paper was written)
- Evaluation on existing database application and their respective databases
- Compare with mock-database generation technique
- Compare with database sampling technique
- Maximize code coverage beyond the limit that can be reached with existing database contents
- Handling complex queries
Reference(s)


- Content of some of the slides has been borrowed (verbatim) from the presentation provided along with the paper.
Questions are guaranteed in life;
Answers Aren’t 😊