Formal Methods for Database Application Evolution

İşıl Dillig
University of Texas at Austin

VSTTE 2020
Database Applications

DB application: program that interacts with underlying database

Database applications are ubiquitous

• Enterprise software, web applications, CRM applications, ….
Schema Refactoring

Common theme during DB app evolution: *schema refactoring*

**Examples**

- Splitting/merging tables
- Denormalization
- Even more extreme: Switch from relational schema to non-relational DB
Implications of Schema Changes

Structural schema changes have significant implications!

Data migration: Move data from source to target schema

Code migration: Must re-implement parts of program

Attracted lots of attention

Currently done manually!
Our Recent Research

Program verification and synthesis techniques to help programmers with DB schema refactoring

My goal: Convince you that there are lots of open & interesting problems in this space!
Idealized Model:
Parametrized SQL Programs

Program is a set of methods (transactions), where each method can take user input, but the body is straight-line SQL code.

Each method is either an update transaction (i.e., modifies DB), or query transaction.

```java
string getName(int id) {  
    SELECT name  
    FROM users  
    WHERE uid = id
}

void insertUser  
(int id,  string name) {  
    INSERT INTO users  
    VALUES (id,  name)
}
```
Outline

Part I: Equivalence verification for parametrized SQL programs (POPL’18)

Part II: Synthesizing new version of parametrized SQL program for a given target schema (PLDI’19)

Part III: Open problems & challenges
Verification Problem

Are the two programs equivalent before and after schema change?
void createSub(int id, String name, String fltr)
    INSERT INTO Subscriber VALUES (id, name, fltr);

void updateSub(int id, String fltr)
    UPDATE Subscriber SET filter=fltr WHERE sid=id;

List<Tuple> getSubFilter(int id)
    SELECT filter FROM Subscriber WHERE sid=id;

void createSub(int id, String name, String fltr)
    INSERT INTO Subscriber’ VALUES (id, name, UID_x);
    INSERT INTO Filter VALUES (UID_x, fltr);

void updateSub(int id, String fltr)
    UPDATE Filter SET params=fltr WHERE fid IN
        (SELECT fid_fk FROM Subscriber’ WHERE sid=id);

List<Tuple> getSubFilter(int id)
    SELECT params FROM Filter JOIN Subscriber’
        ON fid=fid_fk WHERE sid=id;

Are these implementations equivalent?
Defining Equivalence

Consider two SQL programs \( P \) and \( P' \) that provide same interface but different implementations:

\[ P \equiv P' \text{: Every query transaction yields the same result after invoking same sequence of update transactions} \]
Example Revisited

After an arbitrary sequence of `createSub` and `updateSub` transactions, `getSubFilter` should yield same answer.
Proving Equivalence: Methodology

Find \textit{bisimulation invariant} $\Phi$ that relates the two DB states

$\Phi \circlearrowright$  

$S$  

$D_0$  \hspace{1cm} $D_1$  \hspace{1cm} $D_2$  \hspace{1cm} $\ldots$  

$\Phi$  

$S'$  

$D_0'$  \hspace{1cm} $D_1'$  \hspace{1cm} $D_2'$  \hspace{1cm} $\ldots$  

$A = B \Join C$

$(\Phi \land \vec{x} = \vec{y}) \models [Q_i] = [Q'_i]$
Inferring Bisimulation Invariants

Automatically infer inductive bisimulation invariants using a guess-and-check approach (Houdini)

• Generate a universe of candidate predicates from a set of templates

\[ \Pi_\theta(?) = \Pi_\theta(?) \quad \Pi_\theta(?) = \Pi_\theta(\land ?) \]

• Perform fixed-point computation to find strongest (conjunctive) bisimulation invariant over this universe
Verification Workflow

1. Generate predicate universe $P$

2. Use SMT solver

3. Check if it implies query results are same

4. Generate VCs

5. $\Phi \equiv \bigwedge_{\varphi \in P} \varphi$

6. Check if $\varphi$ implies query results are same

7. If No, $P \setminus \{\varphi\}$

8. If Yes, $\varphi$ is inductive
Outline

Part I: Equivalence verification for parametrized SQL programs (POPL’18)

Part II: Synthesizing new version of parametrized SQL program for a given target schema (PLDI’19)

Part III: Open problems & challenges
Synthesis Problem

Program P

Schema S'

Enumerate-and-verify approach

Synthesizer

New program P'

Check equivalence using technique from Part I

P' is over new schema S' and equivalent to P
Challenge

Search space is very large!

Sketch generation

Sketch completion
Synthesis Methodology

Source schema → Value Correspondence Generator → Target schema

Mapping from source attributes to target attributes

Sketch Generator

Value Corr.

Sketch

Sketch Solver

Source prog → Target prog

Program with unknown tables and columns
Motivating Example for Synthesis

update addTA(int id, String name, Binary pic)
    INSERT INTO TA VALUES (id, name, pic);

update deleteTA(int id)
    DELETE FROM TA WHERE Tald=id;

query getTAInfo(int id)
    SELECT TName, TPic FROM TA WHERE Tald=id;
Motivating Example for Synthesis

update addTA(int id, String name, Binary pic)
    INSERT INTO TA VALUES (id, name, pic);

update deleteTA(int id)
    DELETE FROM TA WHERE Taid=id;

query getTAInfo(int id)
    SELECT TName, TPic FROM TA WHERE Taid=id;
Motivating Example for Synthesis

update addTA(int id, String name, Binary pic)
    INSERT INTO TA VALUES (id, name, pic);

update deleteTA(int id)
    DELETE FROM TA WHERE Tald=id;

query getTAInfo(int id)
    SELECT TName, TPic FROM TA WHERE Tald=id;

update addTA(int id, String name, Binary pic)
    INSERT INTO TA VALUES (id, name, pic);

update deleteTA(int id)
    DELETE FROM TA WHERE Tald=id;

query getTAInfo(int id)
    SELECT TName, TPic FROM TA WHERE Tald=id;

update addTA(int id, String name, Binary pic)
    INSERT INTO TA VALUES (id, name, pic);

update deleteTA(int id)
    DELETE FROM TA WHERE Tald=id;

query getTAInfo(int id)
    SELECT TName, TPic FROM TA WHERE Tald=id;
Motivating Example for Synthesis

update addTA(int id, String name, Binary pic)
    INSERT INTO TA VALUES (id, name, pic);

update deleteTA(int id)
    DELETE FROM TA WHERE Tald=id;

query getTAInfo(int id)
    SELECT TName, TPic FROM TA WHERE Tald=id;

query getTAInfo(int id)
    SELECT ??1, ??2 FROM ??3 WHERE ??4=id;

??1=TName   ??2=Pic   ??4=Tald
??3 ∈ { Picture ⋈ TA, Picture ⋈ TA ⋈ Inst }
Motivating Example for Synthesis

update addTA(int id, String name, Binary pic)
    INSERT INTO TA VALUES (id, name, pic);

update deleteTA(int id)
    DELETE FROM TA WHERE Taid = id;

query getTAInfo(int id)
    SELECT TName, TPic FROM TA WHERE Taid = id;

??1, ??4 ∈ \{ Picture ⋈ TA, Picture ⋈ TA ⋈ Inst \}
??2 ∈ \{ [Picture], [TA], ..., [Picture, TA, Inst] \}
??3 ∈ \{ Picture ⋈ TA, Picture ⋈ TA ⋈ Inst \}
Solving the Sketch

**Basic idea:** Enumerate all programs in search space

For each completion, check if it is equivalent to original program

Scalability? 15 holes, 3 instantiations: >14 million programs!

Use **conflict-driven learning** to prune search space

* Program Synthesis using Conflict-Driven Learning. Feng, Martins, Bastani, Dillig. PLDI'18
Learning from Conflicts

Whenever you enumerate an incorrect program, infer a set of other provably-incorrect programs!

Prior work* shows how to do this for programming-by-example, but not applicable here

Leverage notion of “minimum distinguishing inputs” to learn from failed synthesis attempts!

* Program Synthesis using Conflict-Driven Learning. Feng, Martins, Bastani, Dillig. PLDI’18
Distinguishing Inputs

Recall: Input to DB program is a set of function invocations along with their arguments.

Input:

\[
\text{addTA}(1, \text{"A"}, \text{null});
\]
\[
\text{getTALInfo}(1);
\]

Input I is distinguishing for a pair of programs P, P' iff P(I) \( \neq P'(I) \)
Conflict-Driven Learning from Distinguishing Inputs

Consider distinguishing input $I$ for original program $P$ and synthesized (incorrect) program $P'$

Suppose $P, P'$ contain $N$ functions but $I$ invokes only $K$ functions where $K \ll N$

Any program $P''$ that agrees with $P'$ on those $K$ functions will also be incorrect!!

Rather than just rejecting a single program, we can reject a whole SET!

The smaller the input, the bigger the set, so want minimum distinguishing inputs!
Distinguishing Inputs in Action

Synthesized incorrect program:

```
update addTA(int id, String name, Binary pic)
    INSERT INTO TA VALUES (id, name, UID1);
    INSERT INTO Picture VALUES (UID1, pic);

update deleteTA(int id)
    DELETE Picture FROM Picture
        JOIN TA ON Picture.PicId = TA.PicId
        JOIN Inst ON TA.PicId = Inst.PicId WHERE TaId = id;

query getTaInfo(int id)
    SELECT TName, Pic FROM Picture
        JOIN TA ON Picture.PicId = TA.PicId
        JOIN Inst ON TA.PicId = Inst.PicId WHERE Taid = id;
```

Distinguishing Input:
```
addTA(1, “A”, null);
getTAInfo(1);
```

Assignments to holes in deleteTA are irrelevant!

Can use this information to rule out 14 programs instead of a single one!
Evaluation

Implemented tool called Migrator: https://github.com/utopia-group/migrator

Used Migrator to synthesize new versions of 20 different parametrized SQL programs
Statistics about Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>Avg functions</th>
<th>Avg # tables (source)</th>
<th>Avg # tables (target)</th>
<th>Avg # cols (source)</th>
<th>Avg # cols (target)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text book</td>
<td>10</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Github</td>
<td>105</td>
<td>12</td>
<td>13</td>
<td>107</td>
<td>110</td>
</tr>
</tbody>
</table>
Key Results

Successfully synthesized equivalent versions of all 20 SQL programs!

<table>
<thead>
<tr>
<th></th>
<th># programs enumerated</th>
<th>Average synthesis time</th>
<th>Average total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Textbook Benchmarks</td>
<td>3</td>
<td>0.4</td>
<td>5.6</td>
</tr>
<tr>
<td>Github benchmarks</td>
<td>19</td>
<td>138</td>
<td>155</td>
</tr>
</tbody>
</table>

Migrator can synthesize new version of SQL programs with over 100 transactions in 2.5 minutes!
Outline

Part I: Equivalence verification for parametrized SQL programs (POPL’18)

Part II: Synthesizing new version of parametrized SQL program for a given target schema (PLDI’19)

Part III: Open problems & challenges
Future Work

Good starting point, but lots of research remains to be done…
Challenge #1

From parametrized SQL programs to real-world DB applications

- Interaction between SQL and other languages
- SQL code dynamically generated
- Computation on query inputs & result
Challenge #2

From relational DB applications to no-SQL applications

Verify equivalence between programs written in different languages

Larger search space: need to synthesize entire program
Challenge #3

Unified verification & falsification

POPL’18 work cannot disprove equivalence

CEGAR-like approaches that search for both proofs & cexs
Call for New Research

Lots of exciting research opportunities for FM research in automating DB application evolution!
Acknowledgements

Yuepeng Wang
PhD student @ UT Austin
Acknowledgements, cont.

James Dong
UT Austin undergrad

Rushi Shah
UT Austin undergrad
Acknowledgements, cont.

Shuvendu Lahiri
Principal Researcher @ MSR

William Cook
Faculty @ UT Austin
Thank you!
Comparison Results

Textbook Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>Migrator</th>
<th>Sketch</th>
<th>Enum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthesis Time (s)</td>
<td>0.4</td>
<td>17770.8</td>
<td>5437.1</td>
</tr>
</tbody>
</table>

Github Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>Migrator</th>
<th>Sketch</th>
<th>Enum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthesis Time (s)</td>
<td>138.4</td>
<td>86400</td>
<td>26643.5</td>
</tr>
</tbody>
</table>

**Take-away:** >190x average speedup compared to enumeration and >750x speed up compared to Sketch!
## Key Results

<table>
<thead>
<tr>
<th></th>
<th># benchmarks verified</th>
<th>Avg verification time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Textbook Benchmarks</td>
<td>10/10</td>
<td>12 s</td>
</tr>
<tr>
<td>Github benchmarks</td>
<td>10/11</td>
<td>47 s</td>
</tr>
</tbody>
</table>

Mediator can verify all but one benchmark!

Avg verification time is under 1 min!

Cause of FP: bisimulation invariant not strong enough
Recall: Bisimulation invariants relate DB states

No existing first-order theory for expressing such invariants

Axiomatized new theory $\mathcal{T}_{RA}$ (Relational Algebra with Updates)

Subscriber

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Subscriber’

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>fid_fk</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Filter

<table>
<thead>
<tr>
<th>fid</th>
<th>params</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

$\Pi_{sid, name}(\text{Subscriber}) = \Pi_{sid', name'}(\text{Subscriber'}) \land$
$\Pi_{sid, name, filter}(\text{Subscriber}) = \Pi_{sid', name', params'}(\text{Subscriber'} \bowtie \text{Filter'})$

See POPL'18 paper!
Refinement

Refactor/Change +New features

Program 1

Database 1

Program 2

Database 2
## Refinement

### Subscriber

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Filter

<table>
<thead>
<tr>
<th>fid</th>
<th>params</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Subscriber’

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>fid_fk</th>
<th>email</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- New transactions may use more tables and columns
- New results “include” old results
Refinement Checking

- Simulation invariant $\Phi$ holds with empty databases
  - Updates $\lambda \vec{x}. U_i$ and $\lambda \vec{y}. U'_i$
  - Queries $\lambda \vec{x}. Q_i$ and $\lambda \vec{y}. Q'_i$

\[
\begin{align*}
\text{Inductive} & \quad \{ \Phi \land \bigwedge_{x_j \in \vec{x}} x_j = y_j \} \quad U_i ; U'_i \quad \{ \Phi \} \\
\text{Sufficient} & \quad (\Phi \land \bigwedge_{x_j \in \vec{x}} x_j = y_j) \models \exists L. \; Q_i = \Pi_L(Q'_i)
\end{align*}
\]

New results include the old results
Generating Value Correspondence

To ensure completeness of synthesis algorithm, need to enumerate **all** possible value correspondences.

Can’t do eagerly!

Need to lazily enumerate in decreasing order of likelihood.

Use MaxSAT-based approach.

- Weighted soft constraints for similarity between attr names.
- Hard constraints for type compatibility requirements.

Allows lazy enumeration of most likely value correspondences.
Sketch Completion Algorithm

Source program → Sketch Encoder → SAT Solver → Target Program

- Sketch Encoder
- SAT Solver
- Equivalence Verifier
- Blocking Clause Learner
- Minimum Differentiating Input
- UNSAT