Data-Oriented Differential Testing of Object-Relational Mapping Systems

Thodoris Sotiropoulos, Stefanos Chaliasos, Vaggelis Atlidakis, Dimitris Mitropoulos and Diomidis Spinellis

ICSE 2021
Distinguished Artifact Award
Object-Relational Mapping (ORM)

- Object-oriented interface on top of relational databases
- Promotes
  - Portability
  - Developers’ productivity
- ORM frameworks are used by millions of applications (e.g., OpenStack, Gitlab, Dropbox)

```python
from django.db import models
class Person(models.Model):
age = models.IntegerField()
name = models.CharField(max_length=20)
p1 = Person(age=31, name="John")
p1.save()
p2 = Person.objects.get(age=32)
p2.delete()
```

SQL queries:

```
INSERT INTO PERSON (age, name) VALUES (31, 'John')
SELECT * FROM PERSON WHERE AGE = 32 LIMIT 1
DELETE FROM PERSON WHERE ID = 2
```
ORM bugs (Django example)

```python
def q1 = T1.objects.using("mysql")
q2 = T2.objects.using("mysql")
q3 = T3.objects.using("mysql")
combined = q1.union(q2).union(q3)
for row in combined:
    pass
```

```
(SELECT 't1'.'id' FROM 't1')
 UNION
 (SELECT 't2'.'id' FROM 't2')
 UNION
  (SELECT 't3'.'id' FROM 't3'))
```

```
django.db.util.ProgrammingError: (1064, "You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'UNION (SELECT 'listing'.'id', 'listing'.'foo' FROM 'listing'))' subquery' at line 1")
```

Django generates a syntactically invalid SQL query with regards to MySQL
ORM bugs (peewee example)

```python
expr = (1 + T.col)
squared = (expr * expr)
data = T.select(fn.sum(expr),
              fn.avg(squared)).all()
for row in data:
    print('avgExpr', row['avgExpr'])
```

Expected SQL query

```sql
SELECT SUM(1 + "t"."col"),
       AVG(1 + "t"."col" * 1 + "t"."col")
FROM "t" AS "t"
```

Peewee generates a both syntactically and semantically valid SQL.
However, the query produces the wrong results.
We use differential testing for establishing a test oracle
Challenges

- Lack of a common specification and input language
- Non-deterministic query results
- DBMS-dependent results (see Django bug)
- Data generation (see peewee bug)
Approach

1. Schema Generation
2. Schema Setup
3. Abstract Query Generation
4. Concretization of Abstract Queries
5. Bug Detection
Abstract Query Language (AQL)

- Supports wide range of operations (through functional notation)
  - Filtering
  - Sorting
  - Aliasing
  - Folding
  - Compound expressions
  - Aggregate functions
  - Unions / Intersections
- Closer to ORM APIs rather than SQL
- AQL queries are generated randomly up to a certain depth

```plaintext
q ∈ Query ::= eval qs | qs[i] | qs[i : i] | fold { (l : α e)^+ } qs
qs ∈ QuerySet ::= new t | apply λ qs | qs ∪ qs
| qs ∩ qs
λ ∈ Func ::= filter p | map d | unique φ
| sort (φ asc) | sort (φ desc)
d ∈ FieldDecl ::= l : e | hidden l : e | d ; d
p ∈ Pred ::= φ ⊕ e | p ∧ p | p ∨ p | ¬p
e ∈ Expr ::= c | φ | α e | e + e | e − e | e * e | e / e
φ ∈ Field ::= t.c | l | φ.c
α ∈ AggrFunc ::= count | sum | avg | max | min
⊕ ∈ BinaryOp ::= > | ≥ | < | ≤
| contains | startswith | endswith
```
Data Generation

An AQL query is encoded as an SMT formula

```plaintext
apply(
    filter ("Table.str" contains "Paul")
    new Table
)
```

A theorem prover generates assignments from which we derive executable INSERT statements

```plaintext
DELETE FROM "table";
INSERT INTO "table"("id","str") VALUES (7,'Paul');
INSERT INTO "table"("id","str") VALUES (-5,'!');
INSERT INTO "table"("id","str") VALUES (-6,'H');
INSERT INTO "table"("id","str") VALUES (13,'\xa0');
INSERT INTO "table"("id","str") VALUES (0,'B');
```
From AQL queries to ORM queries

- Use ORM-specific translators
- Each translator generates
  - The necessary boilerplate code (e.g., imports, db setup)
  - The actual ORM query
  - Code that prints results of the query

```
apply (filter "addCol" > 5 apply (map "addCol": t1.colA + t1.t2.colB new t1)
```

```
1 import os, django
2 from django.db.models import *
3 os.environ.setdefault("DJANGO_SETTINGS_MODULE", "djangoproject.settings")
4 django.setup()
5 from project.models import *
6
7 addCol = F("colA") + F("t2__colB")
8 q = T1.objects.using("sqlite")
9 .annotate(addCol=addCol).filter(addCol__gt=5)
10 .values("addCol")
11
12 for r in q:
13    print("addCol", r["addCol"])
```
Implementation Details

- We implement our approach as a tool called Cynthia
  - Implemented in Scala (~9k LoC)
  - Cynthia uses the Z3 theorem prover
- Cynthia currently provides support for five popular ORMs
  - Django
  - SQLAlchemy
  - Peewee
  - Sequelize
  - Activerecord
- ... and four DBMSs (Sqlite, MySQL, PostgreSQL, MS SQL Server)
Effectiveness

- Cynthia has found 28 bugs, of which 20 have been fixed.
- Most of the bugs have been discovered in Django and SQLAlchemy.
- DBMS-dependent bugs (11 / 28)
- Most of DBMS-dependent bugs are triggered when the code is run on top of PostgreSQL and MSSQL.

<table>
<thead>
<tr>
<th>ORM</th>
<th>Total</th>
<th>Fixed</th>
<th>Confirmed</th>
<th>Unconfirmed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Django</td>
<td>10</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>SQLAlchemy</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sequelize</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>peewee</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Activerecord</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>28</strong></td>
<td><strong>20</strong></td>
<td><strong>5</strong></td>
<td><strong>3</strong></td>
</tr>
</tbody>
</table>
Effectiveness of Solver-Based Data Generation

- We compared our solver-based approach with a “naive” approach
  - I.e., generating random records without considering the constraints of the generated queries

- We spawned 20 testing sessions consisting of 100 AQL queries, and measured in how many queries the ORMs returned empty results

- Unsatisfied Queries (Solver-based approach): **7.9%**

- Unsatisfied Queries ("Naive" approach): **38%**

- We get no improvement even if we generate more records
  - generating 50 random records is the same with generating 1000 random records
Conclusion

- Introduced the first data-oriented differential testing approach for systematically testing ORM implementations
- We showed that differential testing can be also applicable to (seemingly) dissimilar interfaces, such as ORMs
- We showed that compared with other simplistic approaches, our solver-based approach enhances the bug detection capability, and is suitable for differential testing
- Our tool, Cynthia, discovered 28 bugs, most of which have been fixed by the developers.
- The effectiveness of Cynthia can be improved by considering
  - other forms of queries (e.g., write queries)
  - transaction management
Thank you

Tool: https://github.com/theosotr/cynthia

Artifact: https://doi.org/10.5281/zenodo.4455486
### Characteristics of Discovered Bugs

<table>
<thead>
<tr>
<th>Type</th>
<th># Bugs</th>
<th>All</th>
<th>SQLite</th>
<th>MySQL</th>
<th>PostgreSQL</th>
<th>MSSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Error</td>
<td>12</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Invalid SQL</td>
<td>11</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Crash</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>28</strong></td>
<td><strong>17</strong></td>
<td><strong>1</strong></td>
<td><strong>3</strong></td>
<td><strong>4</strong></td>
<td><strong>4</strong></td>
</tr>
</tbody>
</table>

- Most of the discovered bugs are logic errors (12 / 28)
- Followed by “Invalid SQL” bugs (11 / 28) and crashes (5 / 28)
- Almost all “logic errors” are DBMS-independent
- Yet, there is a large number of DBMS-dependent bugs (11 / 28)
- Most of DBMS-dependent bugs are triggered when the code is run on top of PostgreSQL and MSSQL
Bug Detection

- We make DBMS-specific comparisons
- A bug is found when one of the following holds
  - Two ORMs produce different results on the same DBMS.
  - An ORM query is successfully run on a specific DBMS, but the same query written in another ORM fails on the same DBMS.
Concretization of Abstract Queries

- Data Generation
- Translation of AQL query into a concrete ORM query