ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

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 million 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
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
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
```

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

```
django.db.utils.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)

```
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'])
```

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

Expected SQL query

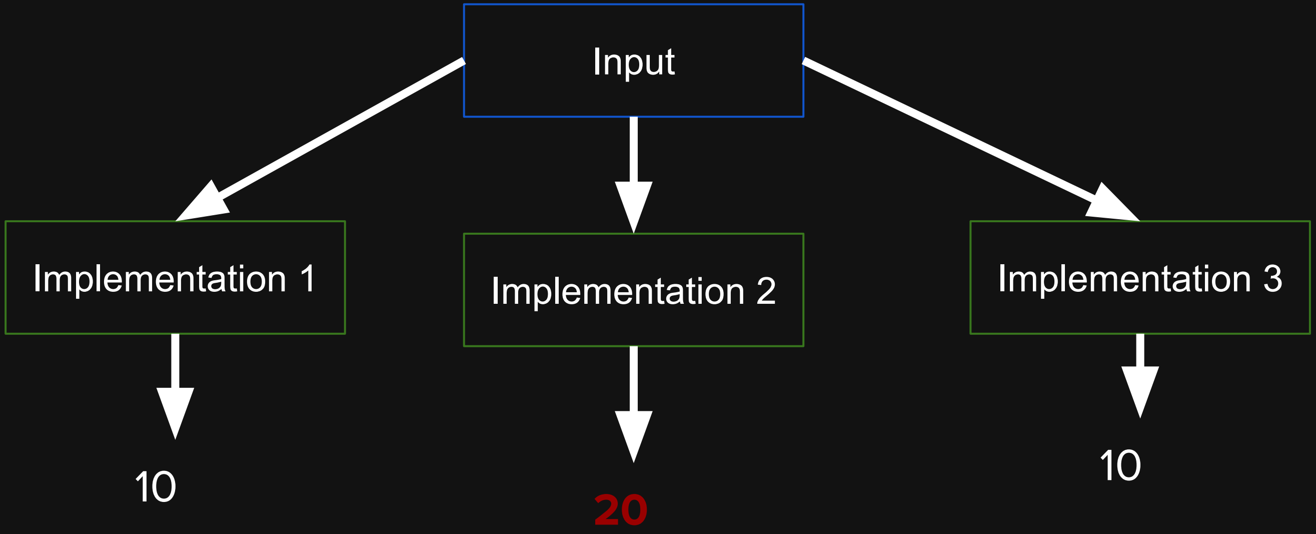
```
SELECT SUM(1 + "t"."col"),
       AVG((1 + "t"."col") * (1 + "t"."col"))
FROM "t" AS "t"
```

```
< avgExpr 19.00
< avgExpr 25.00
< avgExpr 29.00
< avgExpr 47.00
< avgExpr 7.00
< avgExpr 87.00
---
> avgExpr 100.00
> avgExpr 16.00
> avgExpr 169.00
> avgExpr 1936.00
> avgExpr 225.00
> avgExpr 576.00
```

Peewee generates a both syntactically and semantically valid SQL.
However, the query produces the wrong results.

Test Oracle

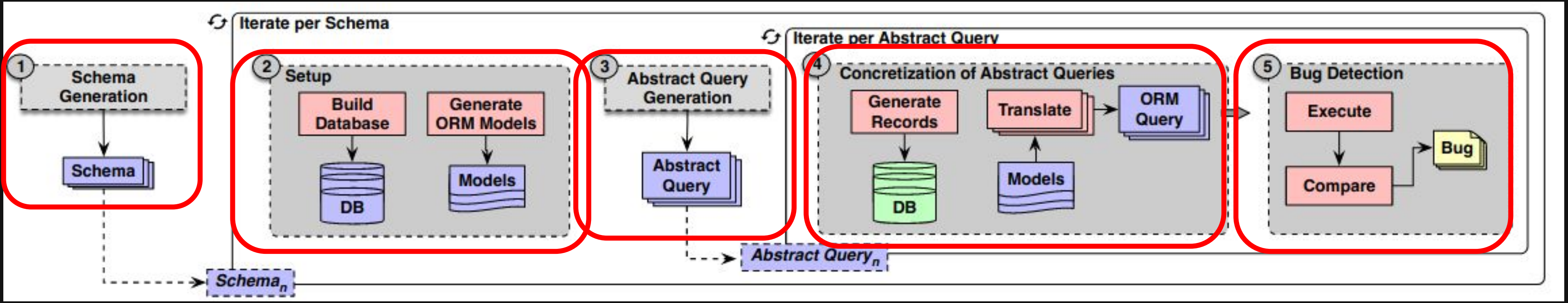
We use differential testing for establishing a test oracle



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 rage 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

$$
\begin{aligned}
q \in \mathit{Query} &::= \textbf{eval}\ qs \mid qs[i] \mid qs[i:i] \mid \textbf{fold}\ \{\ (l : \alpha\ e)^{+}\ \}\ qs \\
qs \in \mathit{QuerySet} &::= \textbf{new}\ t \mid \textbf{apply}\ \lambda\ qs \mid qs \cup qs \\
&\quad \mid qs \cap qs \\
\lambda \in \mathit{Func} &::= \textbf{filter}\ p \mid \textbf{map}\ d \mid \textbf{unique}\ \phi \\
&\quad \mid \textbf{sort}\ (\phi\ \textbf{asc}) \mid \textbf{sort}\ (\phi\ \textbf{desc}) \\
d \in \mathit{FieldDecl} &::= l : e \mid \textbf{hidden}\ l : e \mid d; d \\
p \in \mathit{Pred} &::= \phi \oplus e \mid p \wedge p \mid p \vee p \mid \neg p \\
e \in \mathit{Expr} &::= c \mid \phi \mid \alpha\ e \mid e + e \mid e - e \mid e * e \mid e/e \\
\phi \in \mathit{Field} &::= t.c \mid l \mid \phi.c \\
\alpha \in \mathit{AggrFunc} &::= \textbf{count} \mid \textbf{sum} \mid \textbf{avg} \mid \textbf{max} \mid \textbf{min} \\
\oplus \in \mathit{BinaryOp} &::= \ = \mid > \mid \geq \mid < \mid \leq \\
&\quad \mid \textbf{contains} \mid \textbf{startswith} \mid \textbf{endswith}
\end{aligned}
$$

Data Generation

An AQL query is encoded as an SMT formula

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

```
(declare-fun Table.id_1 () Int)
(declare-fun Table.id_0 () Int)
(declare-fun Table.id_2 () Int)
(declare-fun Table.id_3 () Int)
(declare-fun Table.id_4 () Int)
(declare-fun Table.str_1 () String)
(declare-fun Table.str_0 () String)
(declare-fun Table.str_2 () String)
(declare-fun Table.str_3 () String)
(declare-fun Table.str_4 () String)
(assert (and (not (= Table.id_3 Table.id_4))
      (not (= Table.id_2 Table.id_4))
      (not (= Table.id_2 Table.id_3))
      (not (= Table.id_1 Table.id_4))
      (not (= Table.id_1 Table.id_3))
      (not (= Table.id_1 Table.id_2))
      (not (= Table.id_0 Table.id_4))
      (not (= Table.id_0 Table.id_3))
      (not (= Table.id_0 Table.id_2))
      (not (= Table.id_0 Table.id_1))))
(assert (and (not (= Table.str_3 Table.str_4))
      (not (= Table.str_2 Table.str_4))
      (not (= Table.str_2 Table.str_3))
      (not (= Table.str_1 Table.str_4))
      (not (= Table.str_1 Table.str_3))
      (not (= Table.str_1 Table.str_2))
      (not (= Table.str_0 Table.str_4))
      (not (= Table.str_0 Table.str_3))
      (not (= Table.str_0 Table.str_2))
      (not (= Table.str_0 Table.str_1))))
(assert (or (str.suffixof "Paul" Table.str_0)
     (str.suffixof "Paul" Table.str_1)
     (str.suffixof "Paul" Table.str_2)
     (str.suffixof "Paul" Table.str_3)
     (str.suffixof "Paul" Table.str_4)))
```

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

```
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

```
import os, django
from django.db.models import *
os.environ.setdefault("DJANGO_SETTINGS_MODULE",
                      "djangoproject.settings")
django.setup()
from project.models import *

addCol = F("colA") + F("t2__colB")
q = T1.objects.using("sqlite")\
  .annotate(addCol=addCol).filter(addCol__gt=5)\
  .values("addCol")

for r in q:
  print("addCol", r["addCol"])
```

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

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

ORM	Total	Fixed	Confirmed	Unconfirmed
Django	10	6	3	1
SQLAlchemy	8	8	0	0
Sequelize	5	2	1	2
peewee	4	4	0	0
Activerecord	1	0	1	0
Total	**28**	**20**	**5**	**3**

Effectiveness of Solver-Based Data Generation

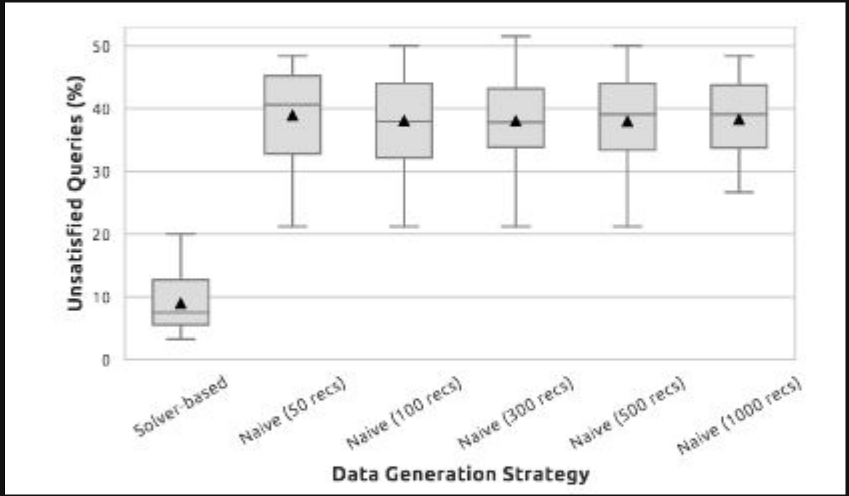
- We compared our solver-based approach with a "naive" approach
  - I.e., generating random records without considering the constraints of the generated queries

- 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

Type	**# Bugs**	**All**	**SQLite**	**MySQL**	**PostgreSQL**	**MSSQL**
Logic Error	12	11	0	0	0	1
Invalid SQL	11	3	1	3	2	3
Crash	5	3	0	0	2	0
Total	**28**	**17**	**1**	**3**	**4**	**4**

- 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