Testing Database-Driven Applications: Challenges and Solutions

Gregory M. Kapfhammer
Department of Computer Science
University of Pittsburgh

Mary Lou Soffa
Department of Computer Science
University of Pittsburgh

Department of Computer Science
Allegheny College
Outline

- Introduction and Motivation
- Testing Challenges
- Database-Driven Applications
- A Unified Representation
- Test Adequacy Criteria
- Test Suite Execution
- Test Coverage Monitoring
- Conclusions and Resources
Motivation

The Risks Digest, Volume 22, Issue 64, 2003

**Jeppesen reports airspace boundary problems**

About 350 airspace boundaries contained in Jeppesen NavData are incorrect, the FAA has warned. The error occurred at Jeppesen after a software upgrade when information was pulled from a database containing 20,000 airspace boundaries worldwide for the March NavData update, which takes effect March 20.
Testing Challenges

- Should consider the environment in which software applications execute
- Must test a program and its interaction with a database
- Database-driven application’s state space is well-structured, but infinite (Chays et al.)
- Need to show program does not violate database integrity, where \( \textit{integrity} = \textit{consistency} + \textit{validity} \) (Motro)
- Must locate program and database coupling points that vary in granularity
Testing Challenges

- The structured query language’s (SQL) data manipulation language (DML) and data definition language (DDL) have different interaction characteristics
- Database state changes cause modifications to the program representation
- Different kinds of test suites require different techniques for managing database state during testing
Testing Challenges

The many testing challenges include, but are not limited to, the following:

- Unified program representation
- Family of test adequacy criteria
- Efficient test coverage monitoring techniques
- Intelligent approaches to test suite execution
Program $P$ interacts with two relational databases
A program can interact with a database at different levels of granularity.
Database Interaction Levels

A program can interact with a database at different levels of granularity.

Relation Level

<table>
<thead>
<tr>
<th>card_number</th>
<th>pin_number</th>
<th>user_name</th>
<th>acct_lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32142</td>
<td>Brian Zorman</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>41601</td>
<td>Robert Roos</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>45322</td>
<td>Marcus Bittman</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>56471</td>
<td>Geoffrey Arnold</td>
<td>0</td>
</tr>
</tbody>
</table>
A program can interact with a database at different levels of granularity.

Database Interaction Levels

```
<table>
<thead>
<tr>
<th>card_number</th>
<th>pin_number</th>
<th>user_name</th>
<th>acct_lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32142</td>
<td>Brian Zorman</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>41601</td>
<td>Robert Roos</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>45322</td>
<td>Marcus Bittman</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>56471</td>
<td>Geoffrey Arnold</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Database Interaction Levels

A program can interact with a database at different levels of granularity.

Attribute Level

D1 → Dn → P

<table>
<thead>
<tr>
<th>card_number</th>
<th>pin_number</th>
<th>user_name</th>
<th>acct_lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32142</td>
<td>Brian Zorman</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>41601</td>
<td>Robert Roos</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>45322</td>
<td>Marcus Bittman</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>56471</td>
<td>Geoffrey Arnold</td>
<td>0</td>
</tr>
</tbody>
</table>
A program can interact with a database at different levels of granularity.

<table>
<thead>
<tr>
<th>card_number</th>
<th>pin_number</th>
<th>user_name</th>
<th>acct_lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32142</td>
<td>Brian Zorman</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>41601</td>
<td>Robert Roos</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>45322</td>
<td>Marcus Bittman</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>56471</td>
<td>Geoffrey Arnold</td>
<td>0</td>
</tr>
</tbody>
</table>
Database Interaction Points

- Database interaction point $I_r \in I$ corresponds to the execution of a SQL DML statement.
- Consider a simplified version of SQL and ignore SQL DDL statements (for the moment).
- Interaction points are normally encoded within Java programs as dynamically constructed Strings.
- `select` uses $D_k$, `delete` defines $D_k$, `insert` defines $D_k$, `update` defines and/or uses $D_k$. 
Database Interaction Points (DML)

\[
\begin{align*}
\text{select } & A_1, A_2, \ldots, A_q \\
\text{from } & r_1, r_2, \ldots, r_m \\
\text{where } & Q \\
\text{delete from } & r \\
\text{where } & Q \\
\text{insert into } & r(A_1, A_2, \ldots, A_q) \\
\text{values} & (v_1, v_2, \ldots, v_q) \\
\text{update } & r \\
\text{set } & A_l = F(A_l) \\
\text{where } & Q
\end{align*}
\]
select * from $R_1$
where $A < (\text{select } \text{avg}(G) \text{ from } R_2)$

update $R_3$
set $J = 500$
where $L < 1000$
Test Adequacy Criteria

- $P$ violates a database $D_k$’s validity when it:
  - (1-v) inserts entities into $D_k$ that do not reflect real world

- $P$ violates a database $D_k$’s completeness when it:
  - (1-c) deletes entities from $D_k$ that still reflect real world

- In order to verify (1-v) and (1-c), $T$ must cause $P$ to define and then use entities within $D_1, \ldots, D_n$.!
Data Flow Information

- **Interaction point:** 
  ```sql`
  UPDATE UserInfo SET acct_lock=1 WHERE card_number='' + card_number + '';''
  ```

- **Database Level:** `define(BankDB)`
- **Attribute Level:** `define(acct_lock)` and `use(card_number)`

- Data flow information varies with respect to the granularity of the database interaction
### Database Entities

#### UserInfo

<table>
<thead>
<tr>
<th>card_number</th>
<th>pin_number</th>
<th>user_name</th>
<th>acct_lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32142</td>
<td>Brian Zorman</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>41601</td>
<td>Robert Roos</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>45322</td>
<td>Marcus Bittman</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>56471</td>
<td>Geoffrey Arnold</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ A_v(I_r) = \{ 1, 32142, \ldots, Geoffrey Arnold, 0 \} \]
entry lockAccount

temp1 = parameter0:c_n

temp2 = LocalDatabaseEntity0:Bank

temp3 = LocalDatabaseEntity1:acct_lock

temp4 = LocalDatabaseEntity2:card_number

“Database-enhanced” CFG for lockAccount

Define temporaries to represent the program’s interaction at the levels of database and attribute
The DICFG: A Unified Representation

- Database interaction graphs (DIGs) are placed before interaction point $I_r$
- Multiple DIGs can be integrated into a single CFG
- String at $I_r$ is determined in a control-fbw sensitive fashion
Test Adequacy Criteria

- Database interaction association (DIA) involves the *def* and *use* of a database entity
- DIAs can be located in the DICFG with data flow analysis
- *all-database-DUs* requires tests to exercise all DIAs for all of the accessed databases
Generating Test Requirements

- Measured time and space overhead when computing family of test adequacy criteria
- Modified ATM and mp3cd to contain appropriate database interaction points
- Soot 1.2.5 to calculate intraprocedural associations
- GNU/Linux workstation with kernel 2.4.18-smp and dual 1 GHz Pentium III Xeon processors
DIAs at attribute value level represent 16.8% of mp3cd’s and 9.6% of ATM’s total number of intraprocedural associations.
Measuring Time Overhead

Computing DIAs at the attribute value level incurs no more than a 5 second time overhead
Measuring Average Space Overhead

mp3cd shows a more marked increase in the average number of nodes and edges than ATM.
mp3cd shows a significantly greater maximum space overhead than ATM
Automatic Representation Construction

- Manual construction of DICFGs is not practical
- Use extension of BRICS Java String Analyzer (JSA) to determine content of String at \( I_r \)
- Per-class analysis is inter-procedural and control flow sensitive
- Conservative analysis might determine that all database entities are accessed
- Include coverage monitoring instrumentation to track DIGs that are covered during test suite execution
DIA coverage can be tracked by recording which DIGs within a DICFG were executed during testing.
Types of Test Suites

Independent

Partially Independent

Non-restricted
Test Suite Execution

- Independent test suites can be executed by using provided setup code to ensure that all $\Delta_\gamma = \Delta_0$
- Non-restricted test suites simply allow state to accrue
- Partially independent test suites must return to $\Delta_\varepsilon$ after $T_\varepsilon$ is executed by:
  1. Re-executing all SQL statements that resulted in the creation of $\Delta_\varepsilon$
  2. Creating a compensating transaction to undo the SQL statements executed by each test after $T_\varepsilon$
The execution of a SQL `insert` during testing requires the re-creation of DICFG(s).

The SQL `delete` does not require re-creation because we must still determine if deleted entity is ever used.

DICFG re-creation only needed when database interactions are viewed at the record or attribute-value level.

Representation extension ripples to other methods.

DICFGs can be re-constructed after test suite has executed, thus incurring smaller time overhead.
Test Coverage Monitoring

- For each tested method $m_i$ that interacts with a database and each interaction point $I_r$ that involves an insert we must:
  1. Update the DICFG
  2. Re-compute the test requirements
- We can compute the set of covered DIAs by consulting the DIG coverage table
- Test adequacy is: $\frac{\# \text{ covered DIAs}}{\# \text{ total DIAs}}$
Calculating Adequacy

\[ T_f \rightarrow m_i \rightarrow m_j \]

**Test Requirements \( M_i \)**

<table>
<thead>
<tr>
<th>DIA</th>
<th>COV?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{def}(e1), \text{use}(e1))&gt;</td>
<td>✅</td>
</tr>
<tr>
<td>(&lt;\text{def}(e2), \text{use}(e2))&gt;</td>
<td></td>
</tr>
<tr>
<td>(&lt;\text{def}(e3), \text{use}(e3))&gt;</td>
<td></td>
</tr>
<tr>
<td>(&lt;\text{def}(e4), \text{use}(e4))&gt;</td>
<td>✅</td>
</tr>
</tbody>
</table>

**Test Requirements \( M_j \)**

<table>
<thead>
<tr>
<th>DIA</th>
<th>COV?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{def}(e5), \text{use}(e5))&gt;</td>
<td>✅</td>
</tr>
<tr>
<td>(&lt;\text{def}(e6), \text{use}(e6))&gt;</td>
<td></td>
</tr>
<tr>
<td>(&lt;\text{def}(e7), \text{use}(e7))&gt;</td>
<td></td>
</tr>
<tr>
<td>(&lt;\text{def}(e8), \text{use}(e8))&gt;</td>
<td>✅</td>
</tr>
<tr>
<td>(&lt;\text{def}(e9), \text{use}(e9))&gt;</td>
<td>✅</td>
</tr>
<tr>
<td>(&lt;\text{def}(e10), \text{use}(e10))&gt;</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{cov}(m_i) = \frac{2}{4} \quad \text{cov}(m_j) = \frac{4}{6} \quad \text{cov}(T_f) = \frac{6}{10}
\]
Related Work

- Jin and Offutt and Whittaker and Voas have suggested that the environment of a software system is important.
- Chan and Cheung transform SQL statements into C code segments.
- Chays et al. and Chays and Deng have created the category-partition inspired AGENDA tool suite.
- Neufeld et al. and Zhang et al. have proposed techniques for database state generation.
- Dauo et al. focused on the regression testing of database-driven applications.
Conclusions

- Must test the program’s interaction with the database
- Many challenges associated with (1) unified program representation, (2) test adequacy criteria, (3) test coverage monitoring, (4) test suite execution
- The DICFG shows database interactions at varying levels of granularity
- Unique family of test adequacy criteria to detect type (1) violations of database validity and completeness
- Intraprocedural database interactions can be computed from a DICFG with minimal time and space overhead
Conclusions

- Test coverage monitoring instrumentation supports the tracking of DIAs executed during testing
- Three types of test suites require different techniques to manage the state of the database
- SQL `insert` statement causes the re-creation of the representation and re-computation of test requirements
- Data fbw-based test adequacy criteria can serve as the foundation for automatically generating test cases and supporting regression testing
Resources


http://cs.allegheny.edu/~gkapfham/research/diatoms/