Testing Database-Driven Applications: Challenges and Solutions

Gregory M. Kapfhammer
Department of Computer Science
University of Pittsburgh

Department of Computer Science
Allegheny College

Mary Lou Soffa
Department of Computer Science
University of Pittsburgh

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 *integrity* = *consistency* + *validity* (Motro)
- Must locate program and database coupling points that vary in granularity

Testing Challenges

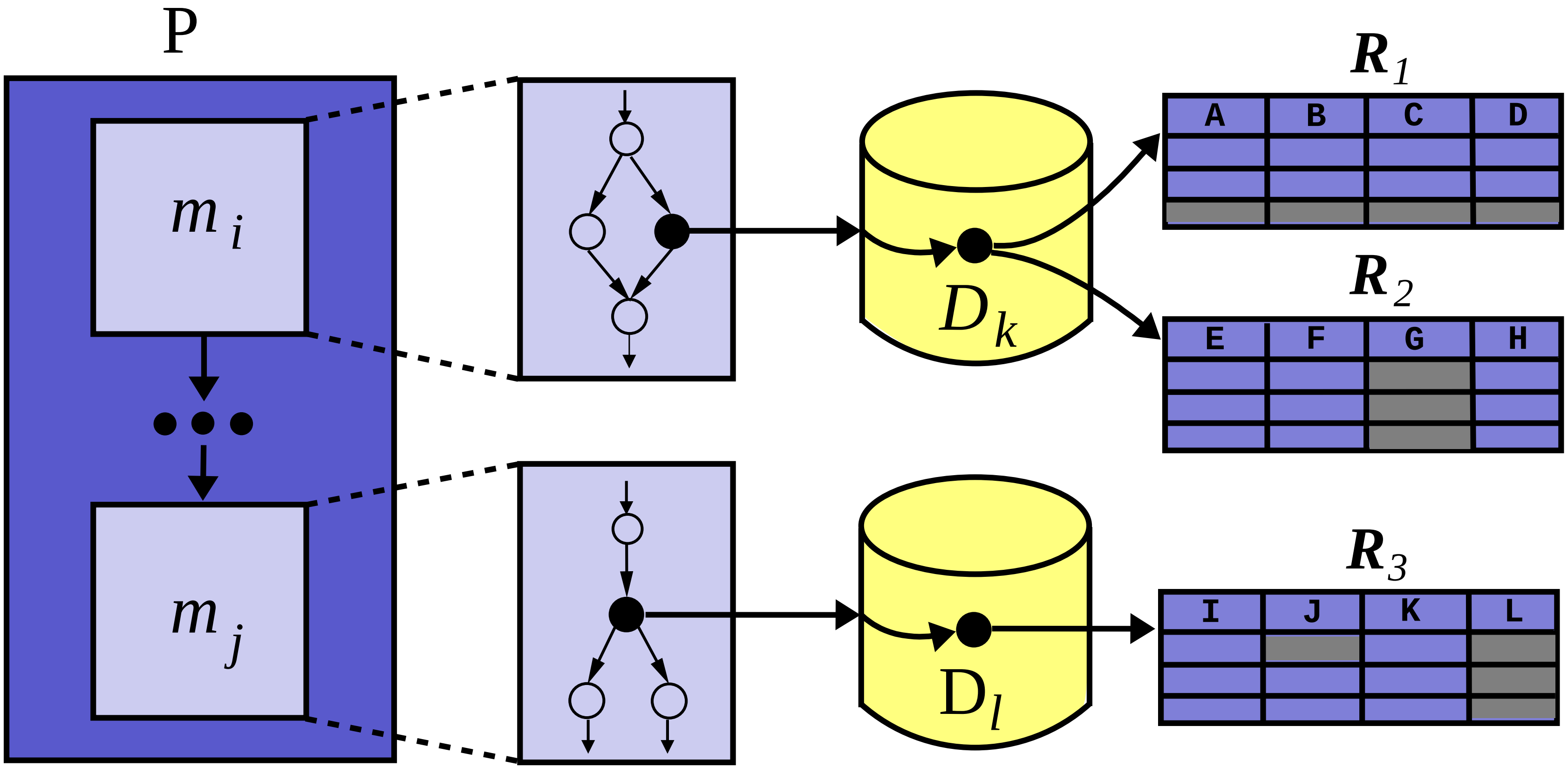
- The structured query language's (SQL) data manipulation language (DML) and data definition language (DDL) have different interaction characteristics
- Database state changes cause modifications to the program representation
- Different kinds of test suites require different techniques for managing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

Testing Challenges

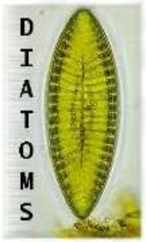
The many testing challenges include, but are not limited to, the following:

- Unified program representation
- Family of test adequacy criteria
- Efficient test coverage monitoring techinques
- Intelligent approaches to test suite execution

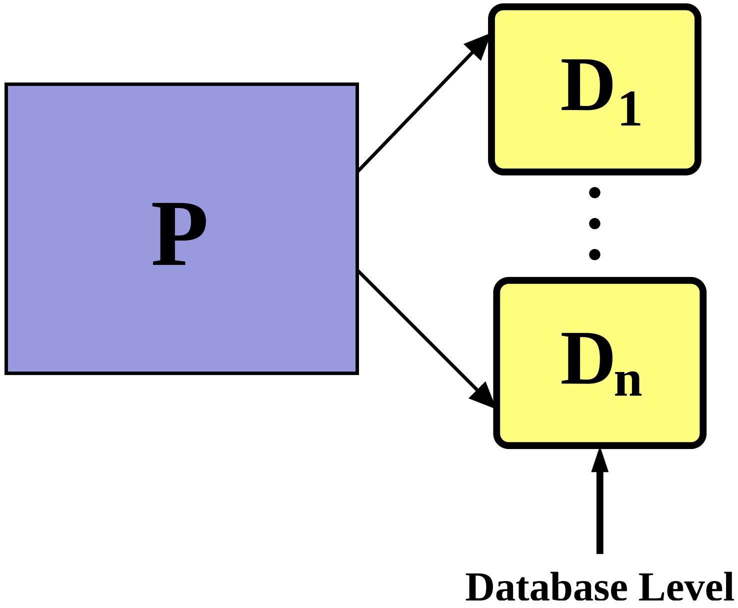
Database-Driven Applications



- Program P interacts with two relational databases

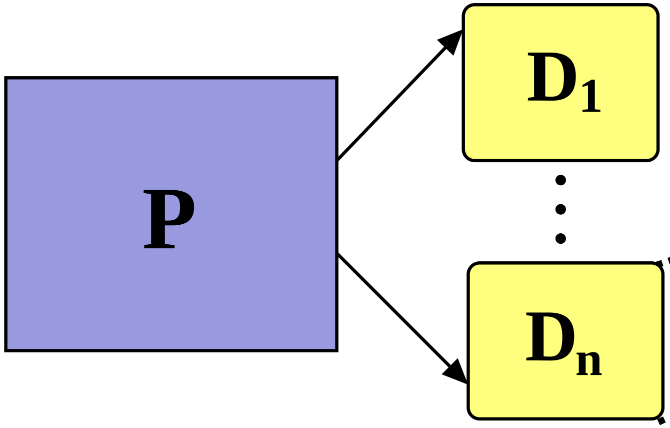


Database Interaction Levels



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

Database Interaction Levels

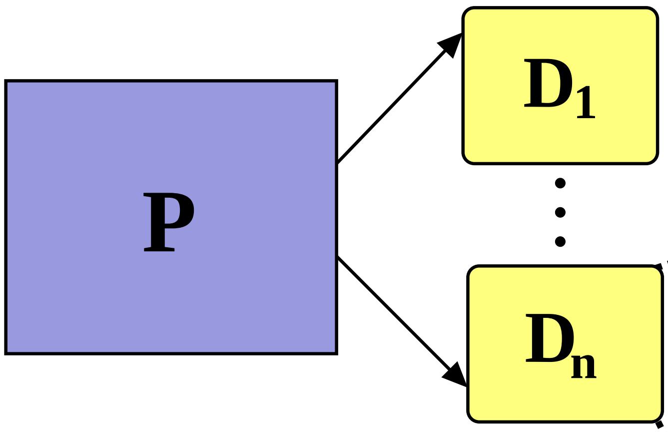


UserInfo

card_number	pin_number	user_name	acct_lock
1	32142	Brian Zorman	0
2	41601	Robert Roos	0
3	45322	Marcus Bittman	0
4	56471	Geoffrey Arnold	0

Relation Level

Database Interaction Levels

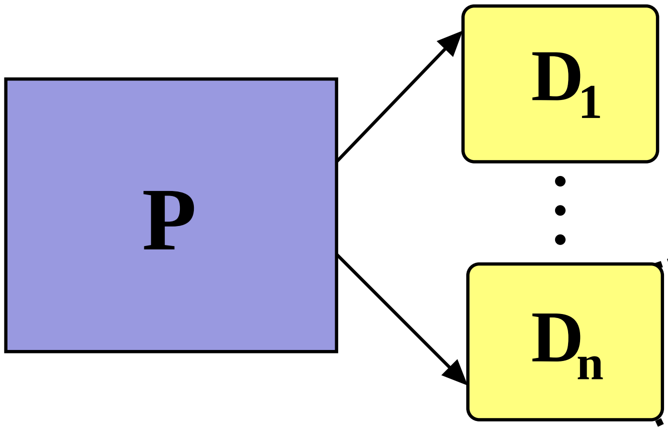


UserInfo

card_number	pin_number	user_name	acct_lock
1	32142	Brian Zorman	0
2	41601	Robert Roos	0
3	45322	Marcus Bittman	0
4	56471	Geoffrey Arnold	0

Record Level

Database Interaction Levels

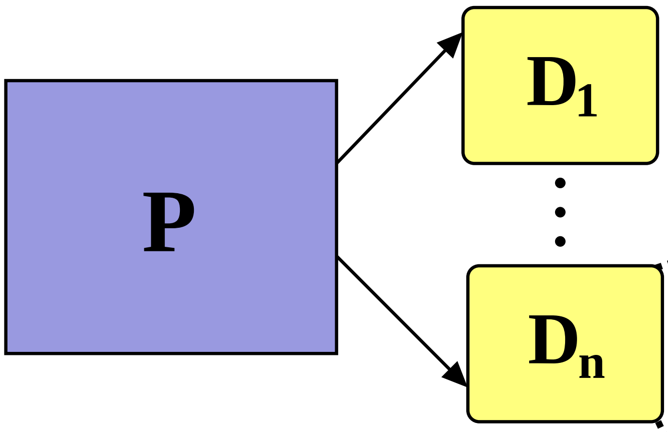


UserInfo

card_number	pin_number	user_name	acct_lock
1	32142	Brian Zorman	0
2	41601	Robert Roos	0
3	45322	Marcus Bittman	0
4	56471	Geoffrey Arnold	0

Attribute Level

Database Interaction Levels



UserInfo

card_number	pin_number	user_name	acct_lock
1	32142	Brian Zorman	0
2	41601	Robert Roos	0
3	45322	$\boxed{\text{Marcus Bittman}}$	0
4	56471	Geoffrey Arnold	0

Attribute Value Level

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)

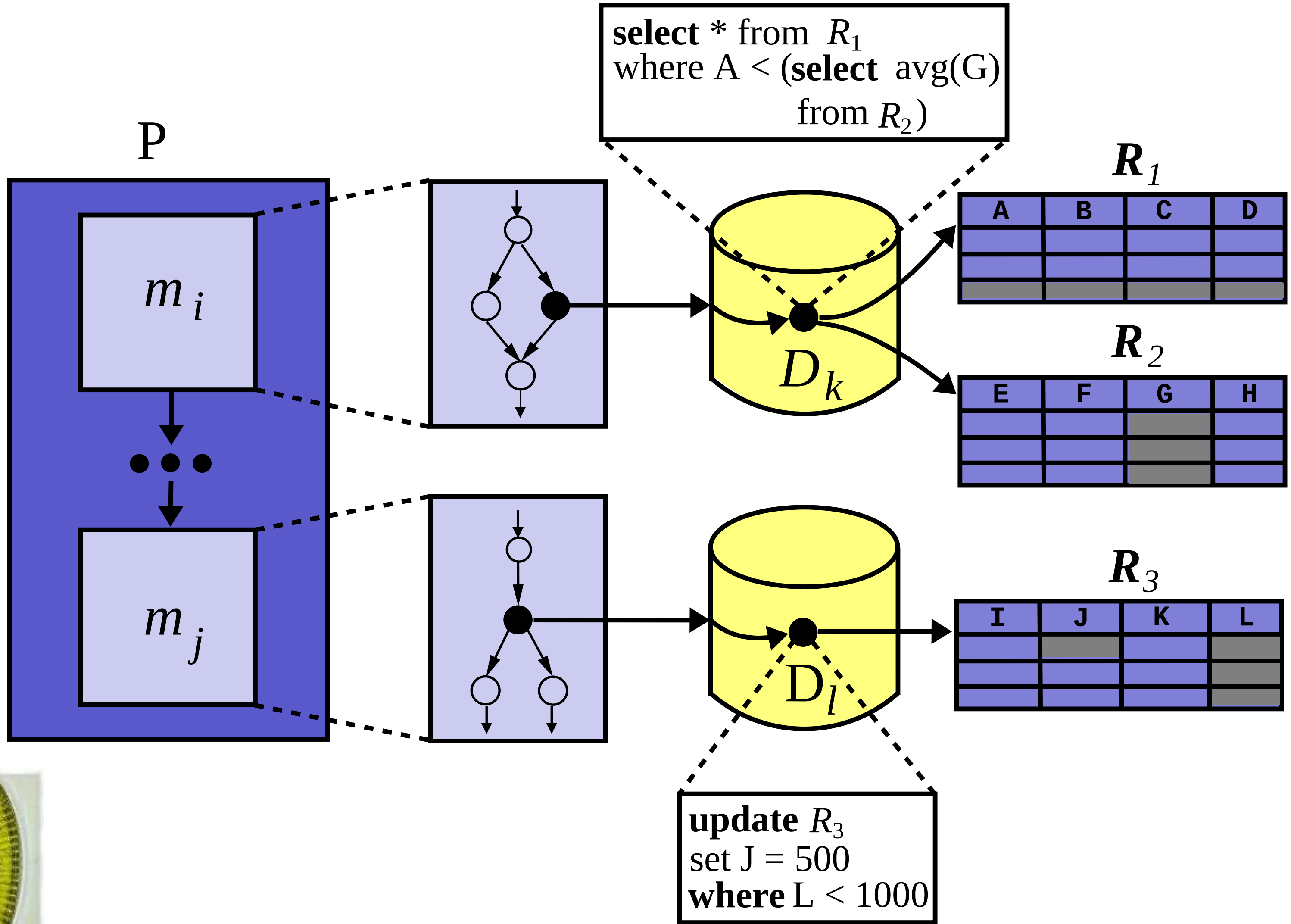
$$\textbf{select } A_1, A_2, \ldots, A_q$$
$$\textbf{from } r_1, r_2, \ldots, r_m$$
$$\textbf{where } Q$$

$$\textbf{delete from } r$$
$$\textbf{where } Q$$

$$\textbf{insert into } r(A_1, A_2, \ldots, A_q)$$
$$\textbf{values}(v_1, v_2, \ldots, v_q)$$

$$\textbf{update } r$$
$$\textbf{set } A_l = F(A_l)$$
$$\textbf{where } Q$$

Refined Database-Driven Application



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: `‘‘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 fbw information varies with respect to the granularity of the database interaction

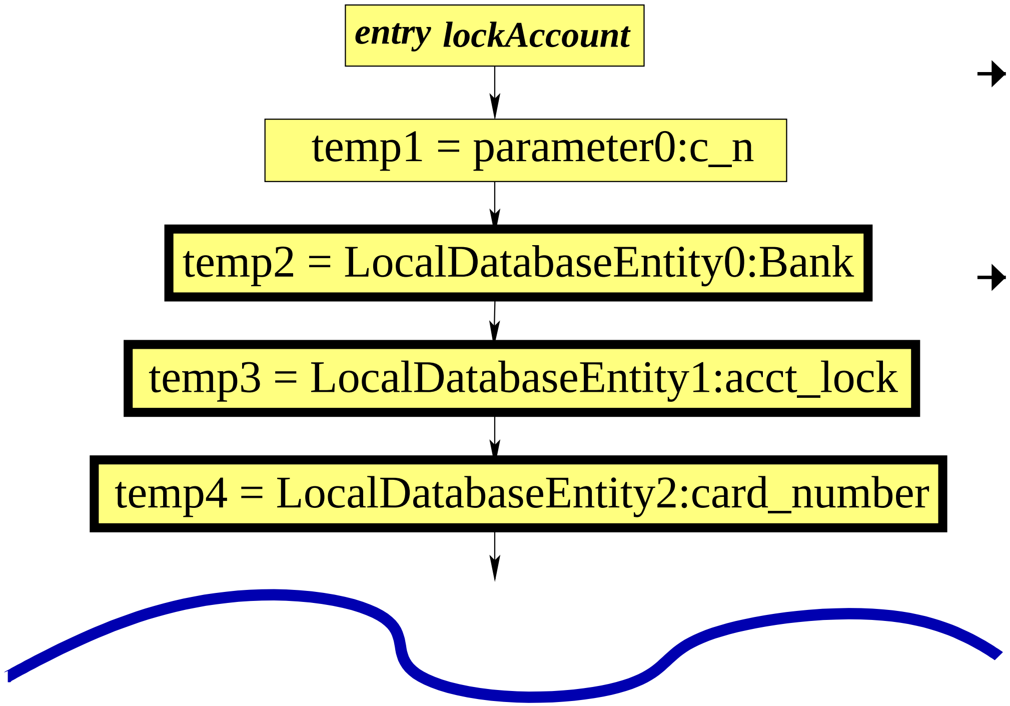
Database Entities

UserInfo

card_number	pin_number	user_name	acct_lock
1	32142	Brian Zorman	0
2	41601	Robert Roos	0
3	45322	Marcus Bittman	0
4	56471	Geoffrey Arnold	0

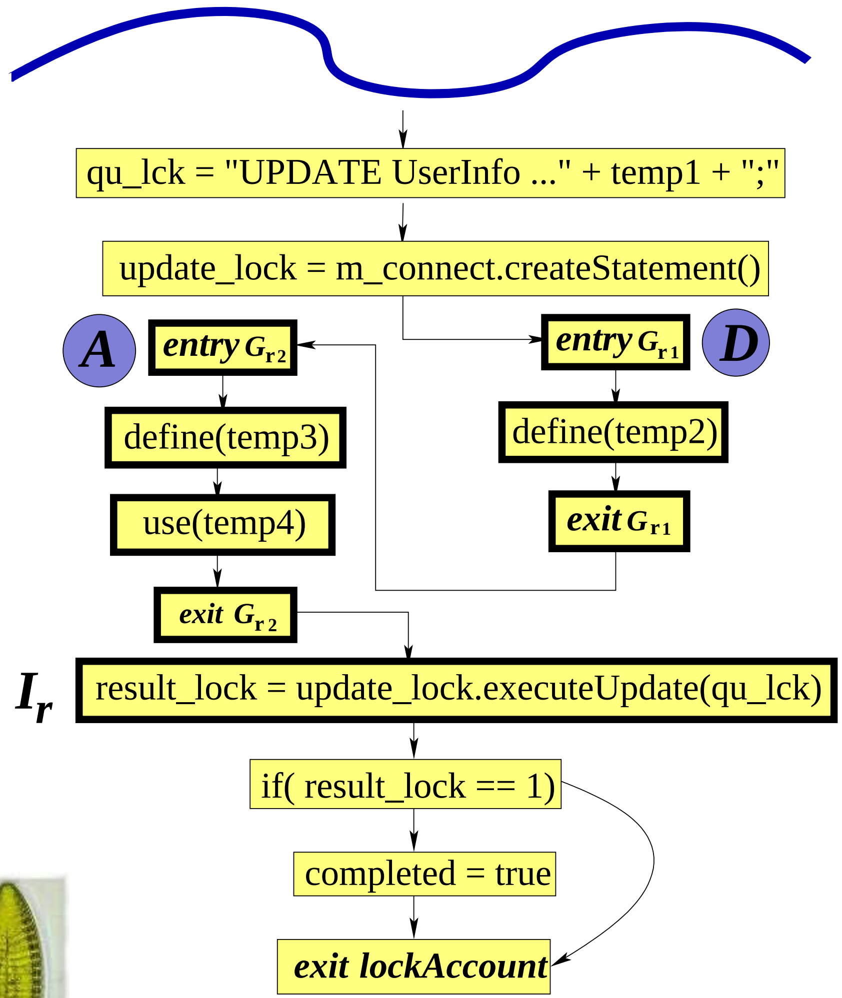
$$A_v(I_r) = \{ \boxed{1}, \boxed{32142}, \ldots, \boxed{\text{Geoffrey Arnold}}, \boxed{0} \}$$

The DICFG: A Unified Representation



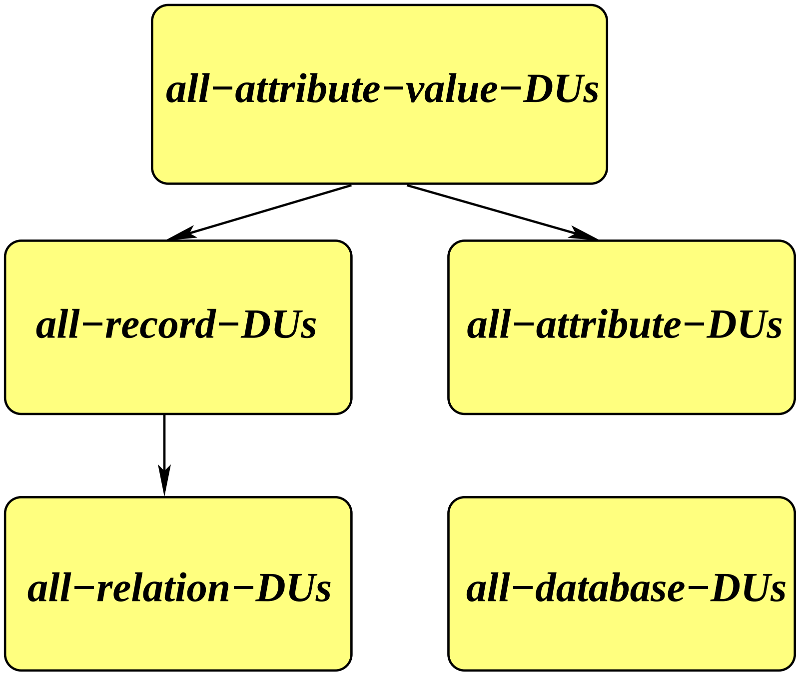
- “Database-enhanced” CFG for `lockAccount`
- Define temporaries to represent the program’s interaction at the levels of database and attribute

The DICFG: A Unified Representation



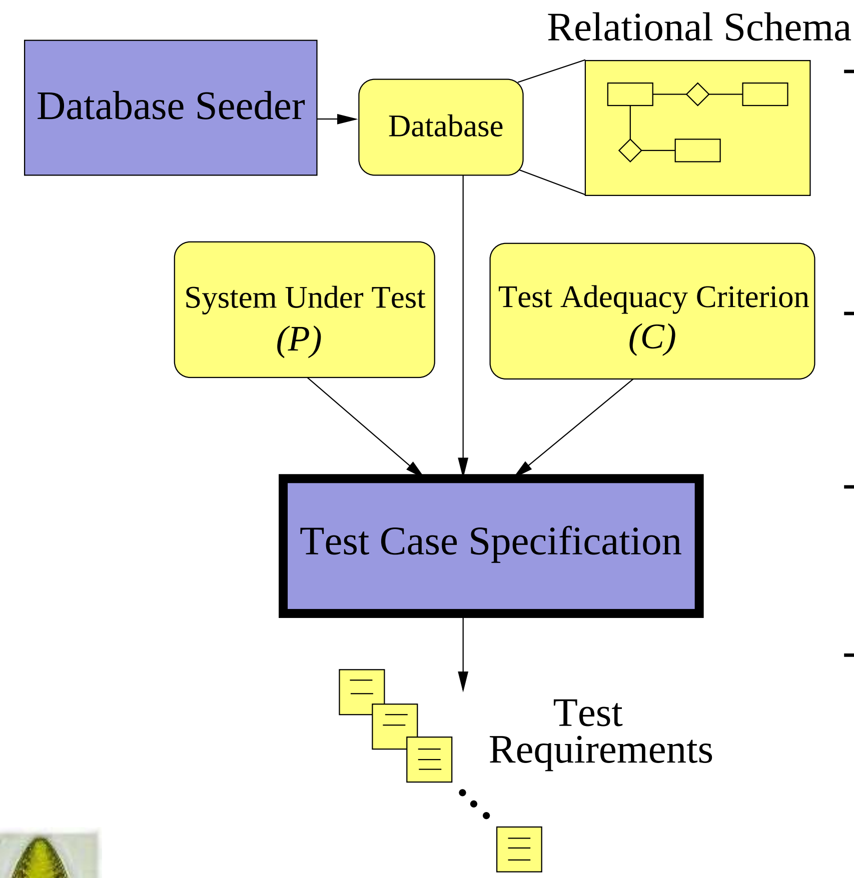
- Database interaction graphs (DIGs) are placed before interaction point I_r
- Multiple DIGs can be integrated into a single CFG
- `String` at I_r is determined in a control-flow sensitive fashion

Test Adequacy Criteria

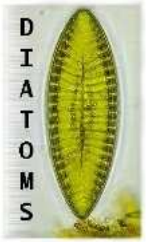


- Database interaction association (DIA) involves the *def* and *use* of a database entity
- DIAs can be located in the DICFG with data flow analysis
- *all-database-DUs* requires tests to exercise all DIAs for all of the accessed databases

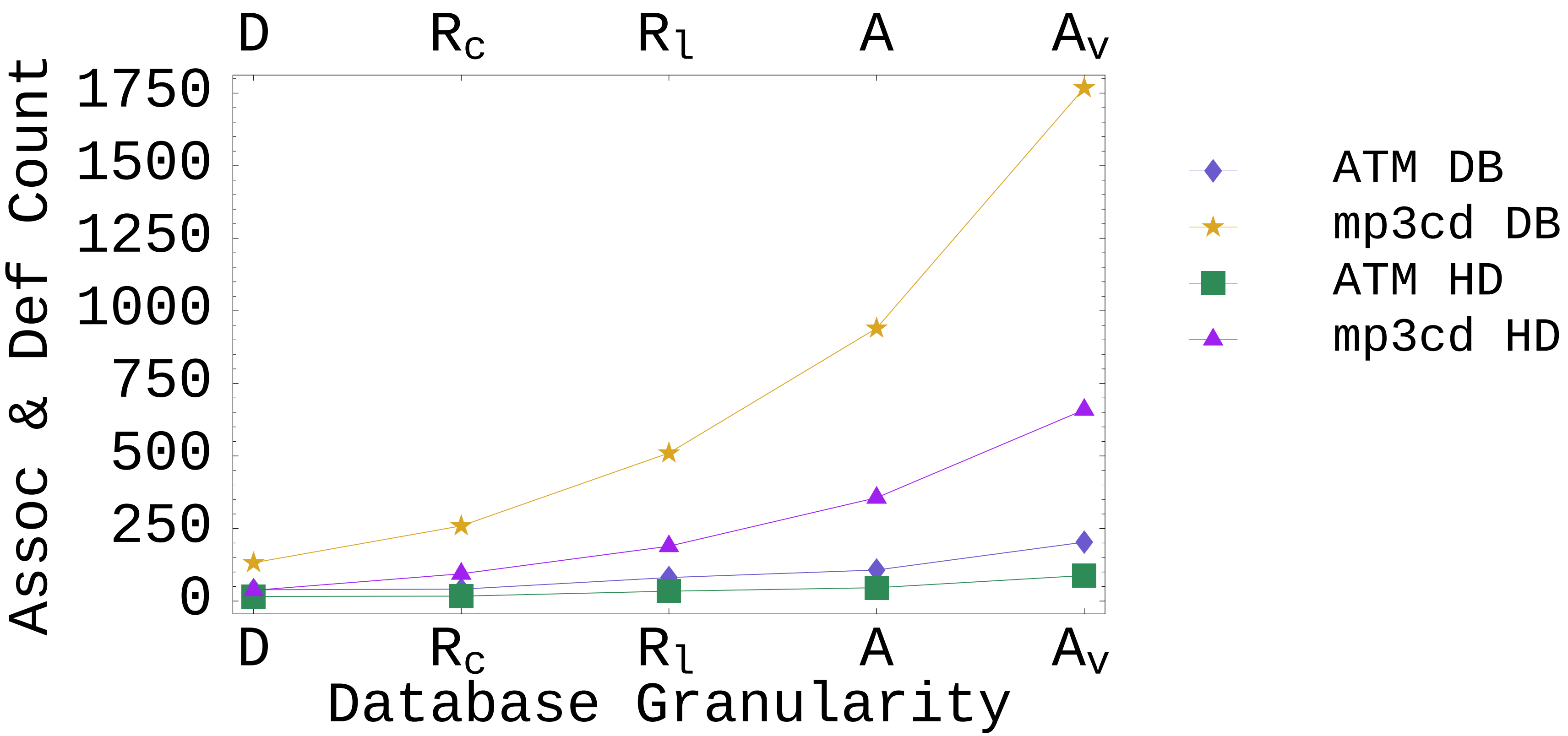
Generating Test Requirements



- Measured time and space overhead when computing family of test adequacy criteria
- Modifi ed `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

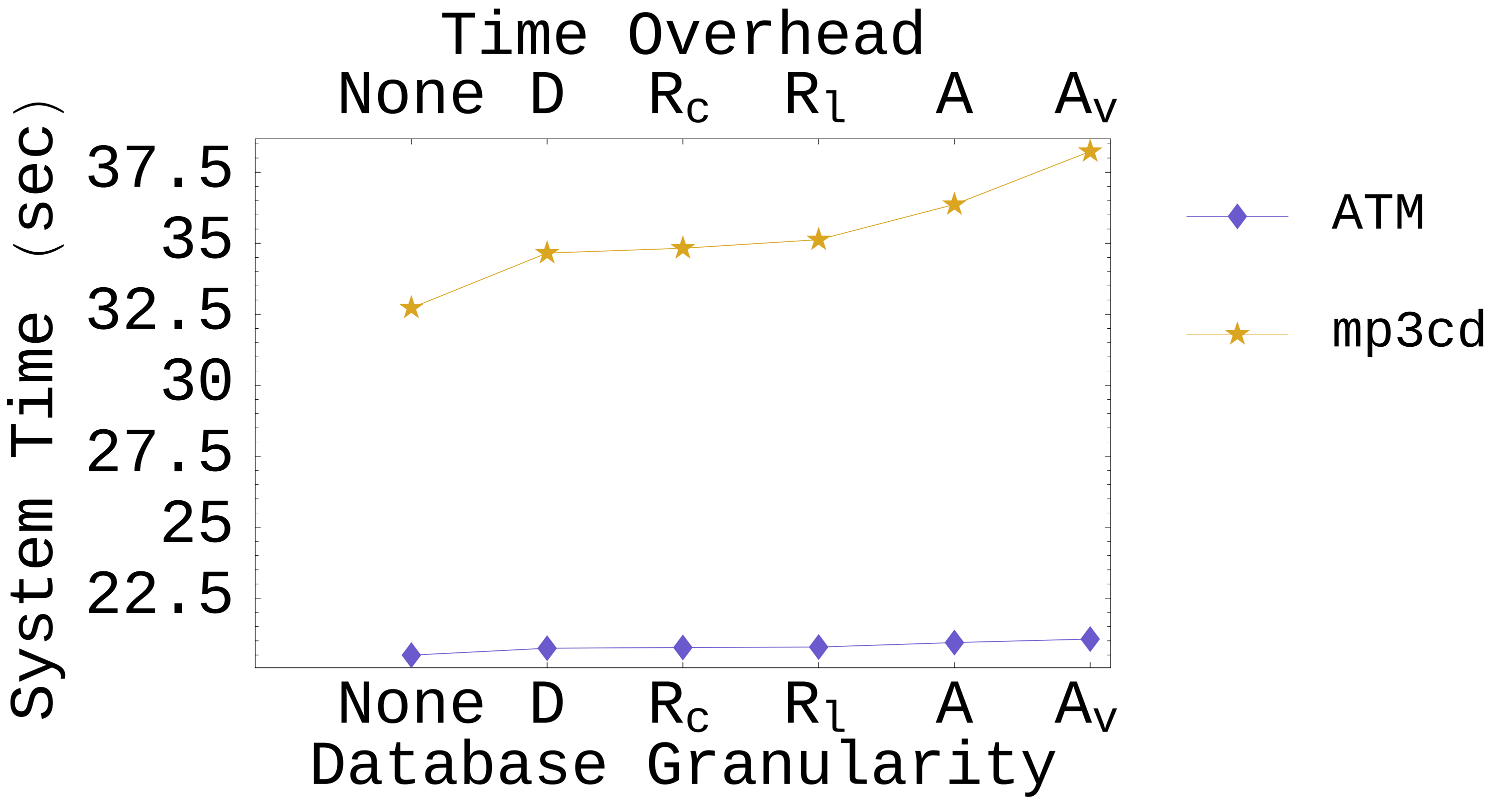


Counting Associations and Definitions



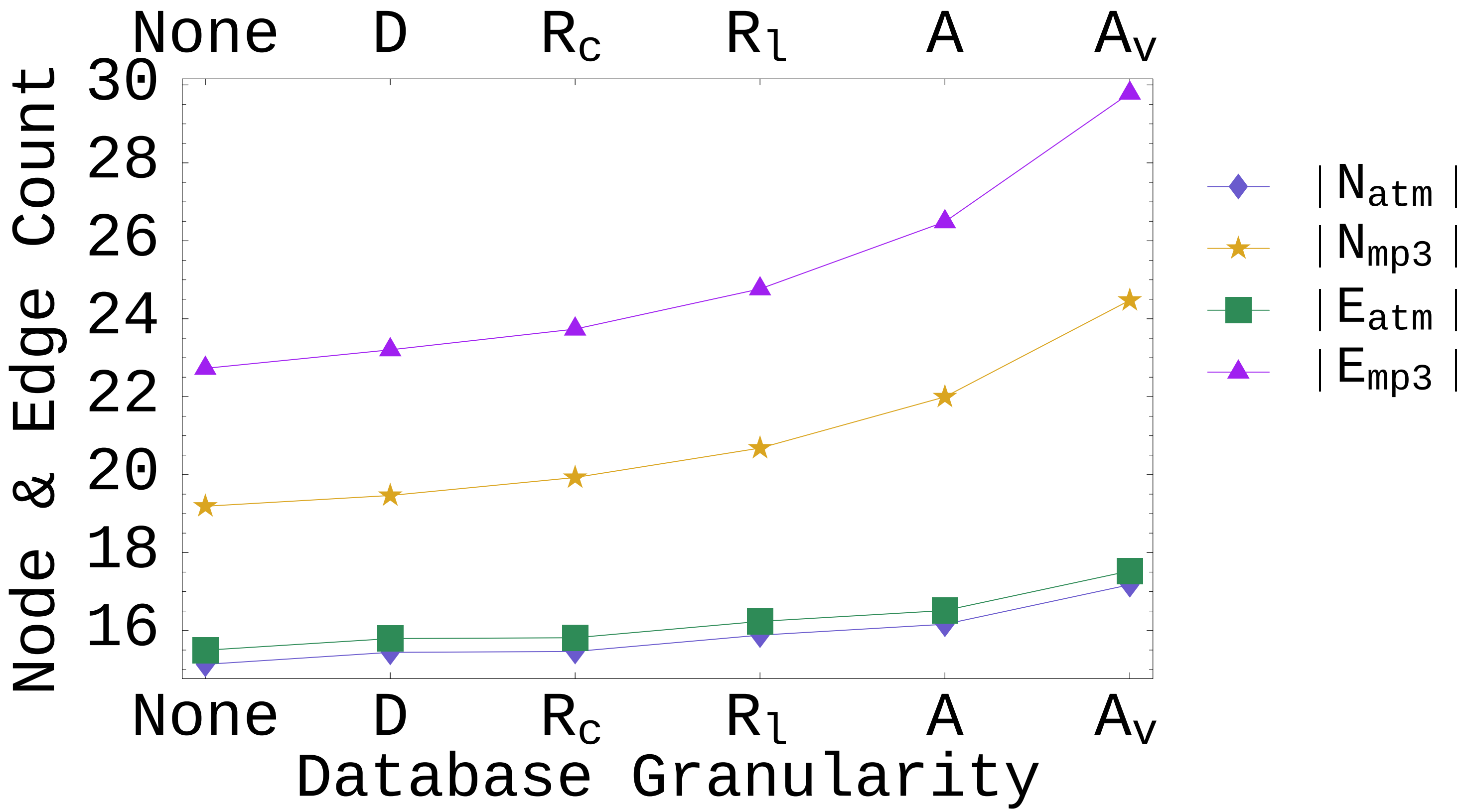
- DIAs at attribute value level represent 16.8% of mp3cd's and 9.6% of ATM's total number of intraprocedural associations

Measuring Time Overhead



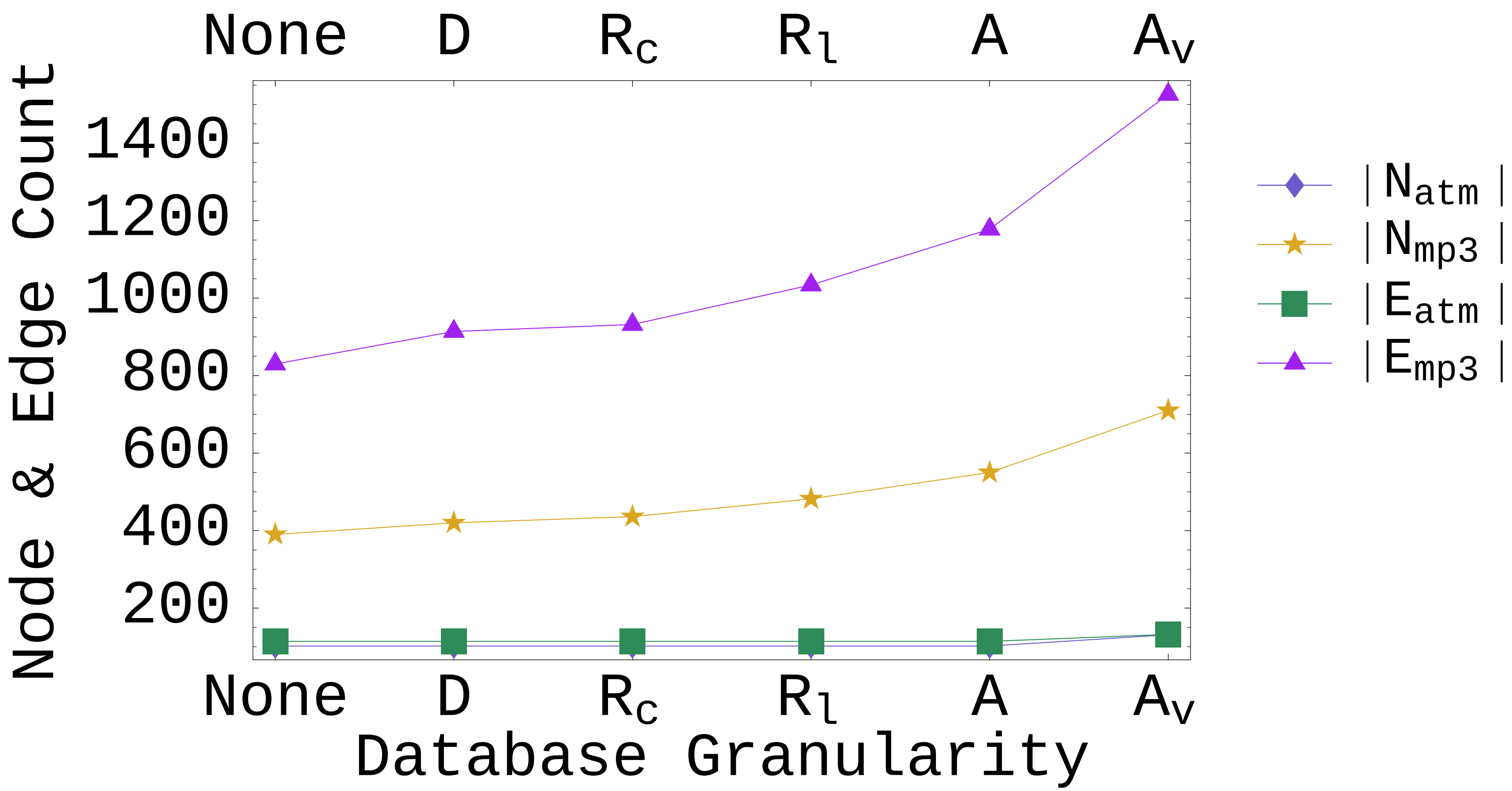
- Computing DIAs at the attribute value level incurs no more than a 5 second time overhead

Measuring Average Space Overhead



- mp3cd shows a more marked increase in the average number of nodes and edges than ATM

Measuring Maximum Space Overhead

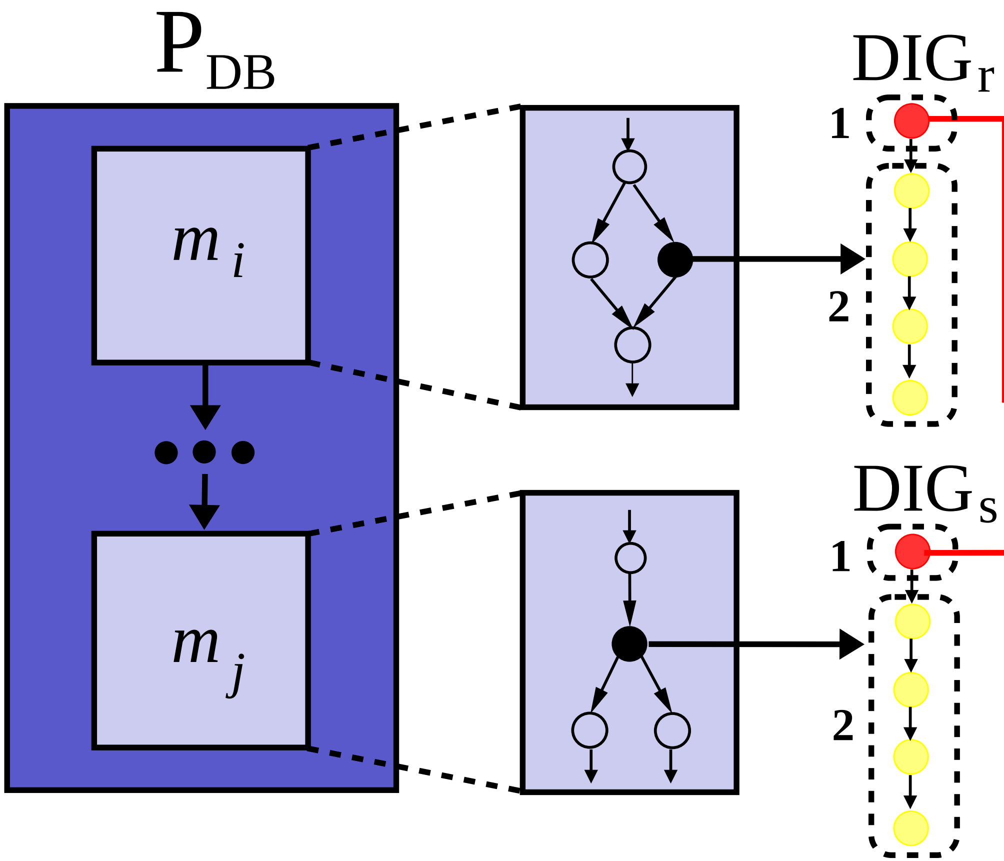


- mp3cd shows a signifi cantly greater maximum space overhead than ATM

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

Tracking Covered DIGs and DIAs

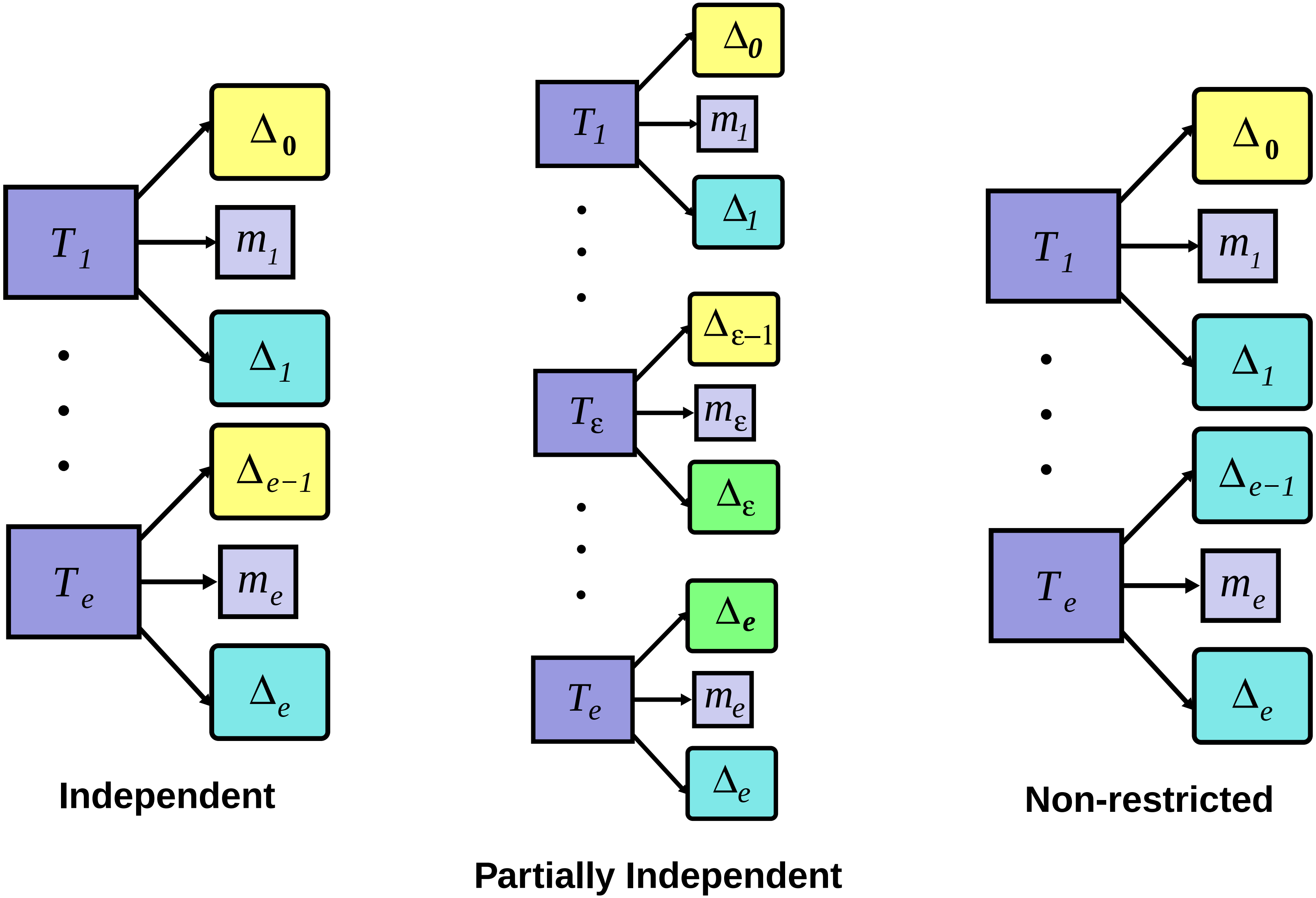


DIG Coverage Table

DIG #	DEF	USE	TEST	COV?
1	{ ... }	{ ... }	{ ... }	✓
• • •				• • •
q	{ ... }	{ ... }	{ ... }	

- DIA coverage can be tracked by recording which DIGs within a DICFG were executed during testing

Types of Test Suites



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 Δ_ε after T_ε is executed by :
  1. Re-executing all SQL statements that resulted in the creation of Δ_ε
  2. Creating a compensating transaction to undo the SQL statements executed by each test after T_ε

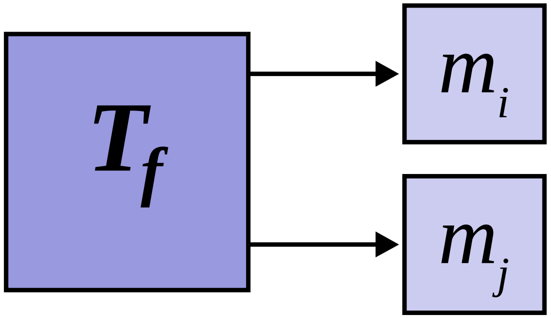
Representation Extension

- 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 : *# covered DIAs / # total DIAs*

Calculating Adequacy



Test Requirements m_i

DIA	COV?
<def(e1), use(e1)>	✓
<def(e2), use(e2)>	
<def(e3), use(e3)>	
<def(e4), use(e4)>	✓

Test Requirements m_j

DIA	COV?
<def(e5), use(e5)>	✓
<def(e6), use(e6)>	
<def(e7), use(e7)>	✓
<def(e8), use(e8)>	✓
<def(e9), use(e9)>	
<def(e10), use(e10)>	✓

$$cov(m_i) = \frac{2}{4} \quad cov(m_j) = \frac{4}{6} \quad cov(T_f) = \frac{6}{10}$$

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) unifi ed 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

Gregory M. Kapfhammer and Mary Lou Soffa. A Family of Test Adequacy Criteria for Database-Driven Applications. In FSE 2003.

Gregory M. Kapfhammer. Software Testing. CRC Press Computer Science Handbook. June, 2004.

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