Combined Static and Dynamic Automated Test Generation

Sai Zhang
University of Washington

Joint work with:

David Saff, Yingyi Bu, Michael D. Ernst

Unit Testing for Object-oriented Programs

- **Unit test = sequence of method calls + testing oracle**

- Automated test generation is **challenging**:
  - **Legal** sequences for constrained interfaces
  - **Behaviorally-diverse** sequences for good coverage
  - **Testing oracles** (**assertions**) to detect errors

Unit Testing a Database Program

```
public void testConnection() {
    Driver driver = new Driver();

    Connection connection =
        driver.connect("jdbc:tinysql");
    Statement s = connection.createStmt();
    s.execute("create table test (name char(25))");

    ....

    s.close();
    connection.close();
}
```

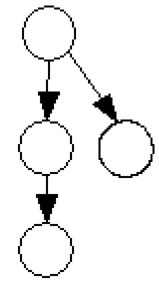
Constraint 1:
Method-call orders

Constraint 2:
Argument values

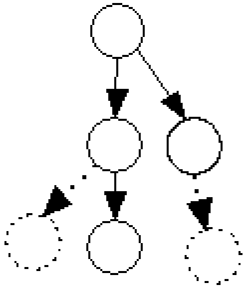
It is hard to create tests automatically!

Palus: Combining Dynamic and Static Analyses

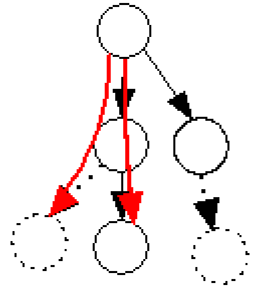
- **Dynamically infer an object behavior model from a sample (correct) execution trace**
  - Capture method-call order and argument constraints



- **Statically identify related methods**
  - **Expand** the (incomplete) dynamic model



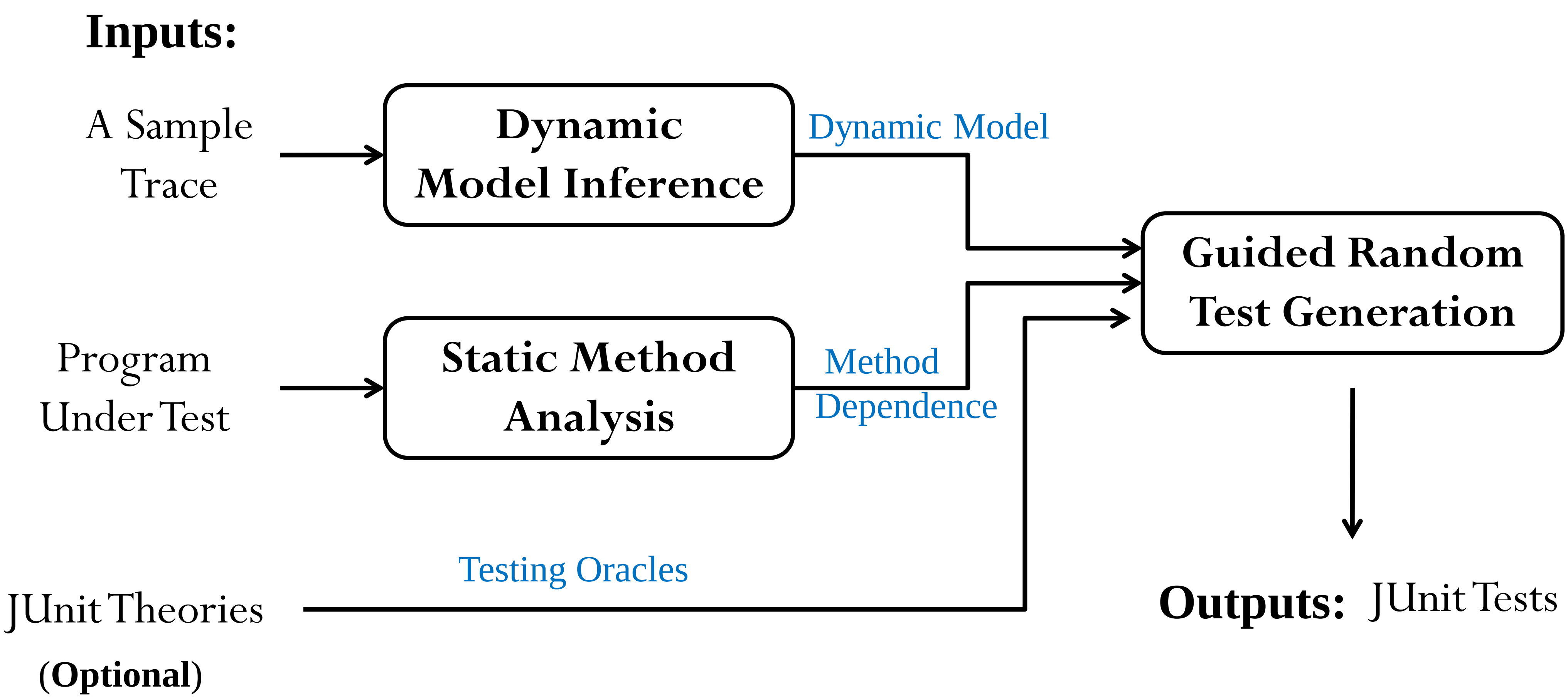
- **Model-Guided random test generation**
  - Fuzz along a specific legal path



Outline

- Motivation
- Approach
  - Dynamic model inference
  - Static model expansion
  - Model-guided test generation
- Evaluation
- Related Work
- Conclusion and Future Work

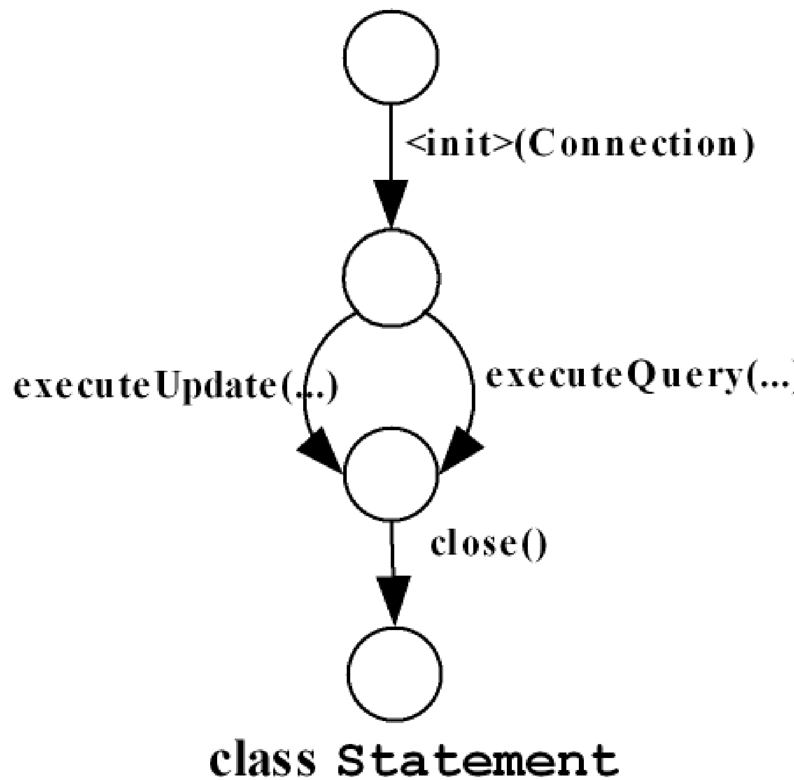
Overview of the Palus approach



(1) Dynamic Model Inference

- Infer a ***call sequence model*** for each tested class
  - Capture **possible ways** to create legal sequences

- A ***call sequence model***
  - A **rooted**, **acyclic** graph
  - **Node**: object state
  - **Edge**: method-call
  - One model **per class**



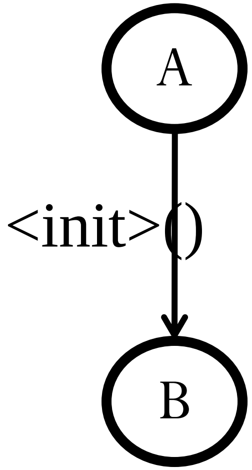
An Example Trace for Model Inference

```
Driver d = new Driver()
Connection con = driver.connection(“jdbc:dbname”);

Statement stmt1 = new Statement(con);
stmt1.executeQuery(“select * from table_name”);
stmt1.close();

Statement stmt2 = new Statement(con);
stmt2.executeUpdate(“drop table table_name”);
stmt2.close();

con.close();
```

Model Inference for class `Driver`

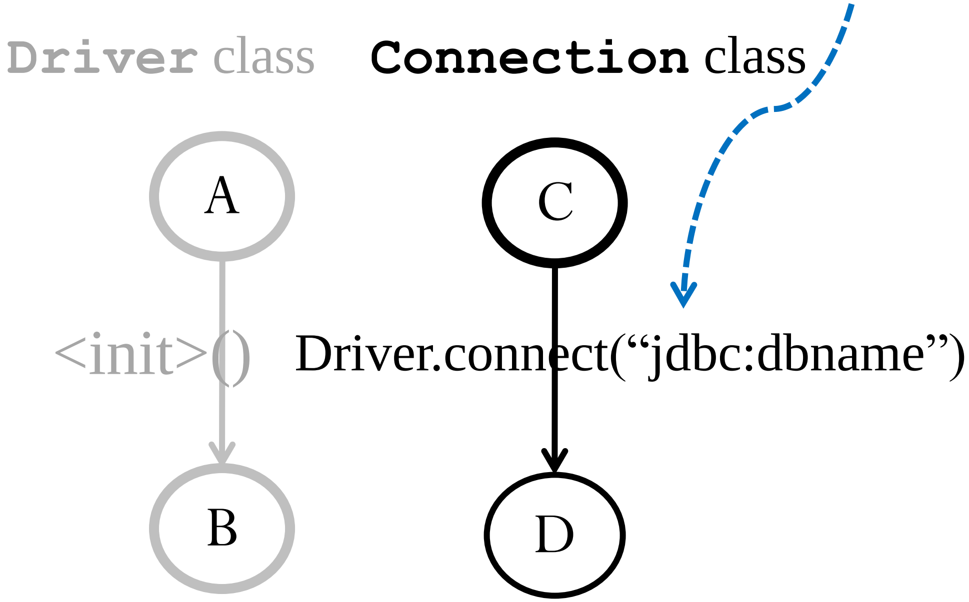
```
Driver d = new Driver();
```

`Driver` class



Model Inference for class `Connection`

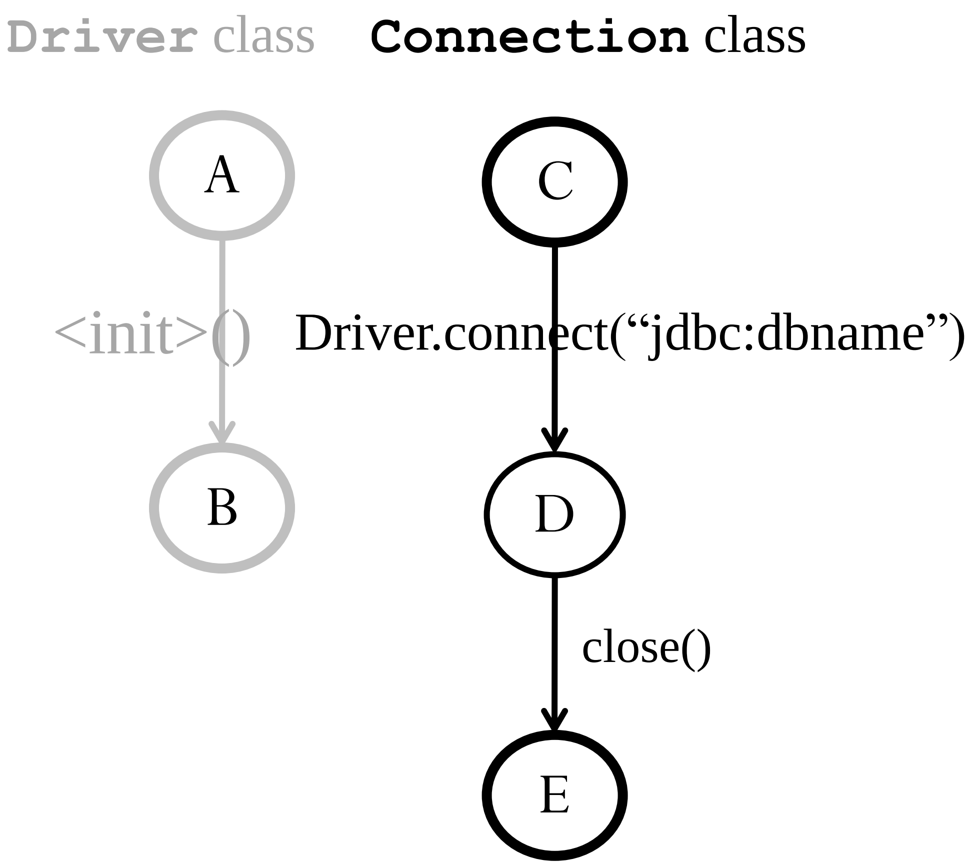
```
Connection con = driver.connect("jdbc:dbname");
```



 Nested calls are omitted for brevity

Model Inference for class `Connection`

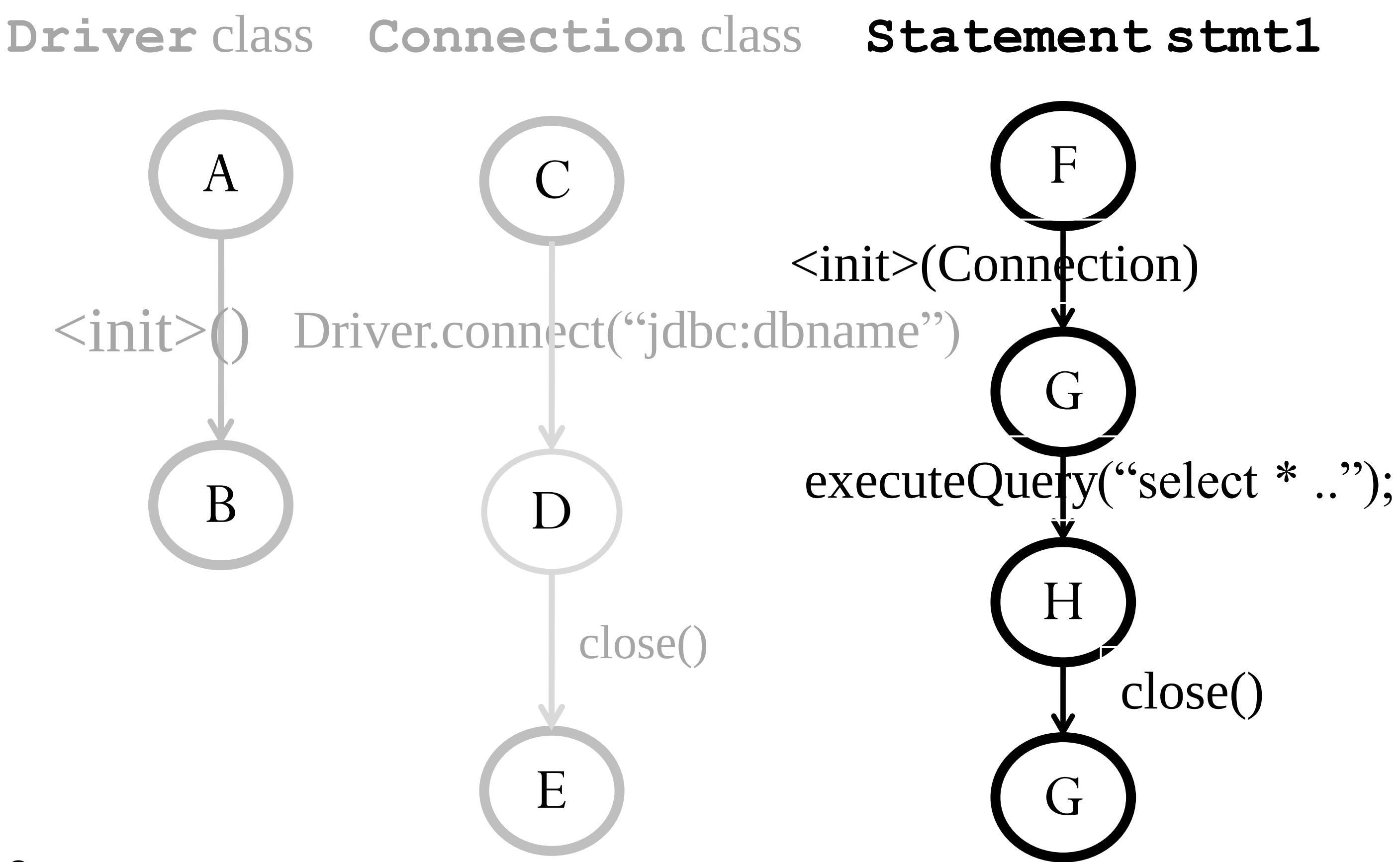
```
Connection con = driver.connect("jdbc:dbname");
con.close();
```



Nested calls are omitted for brevity

Model Inference for class `Statement`

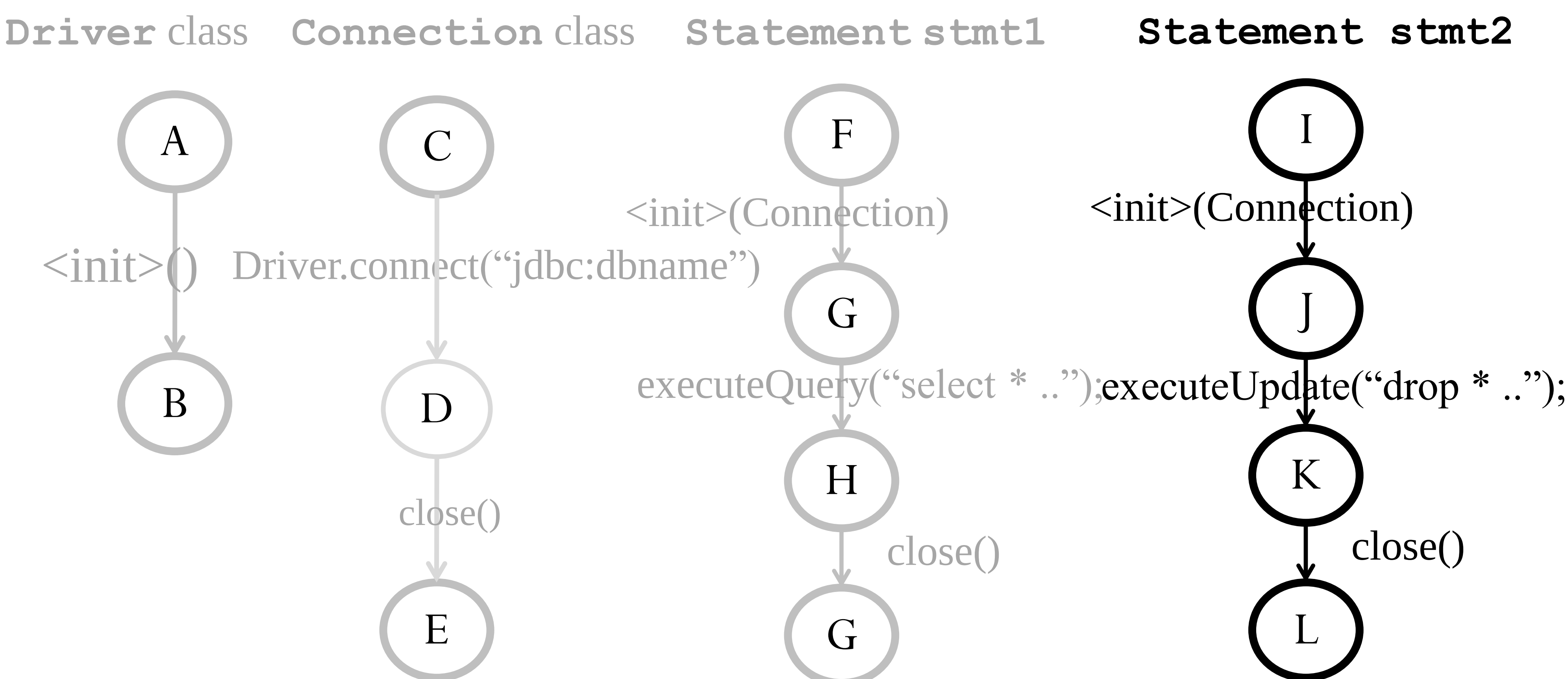
```
Statement stmt1 =  new Statement(con);
stmt1.executeQuery("select * from table_name");
stmt1.close();
```



Construct a call sequence model for **each observed object**

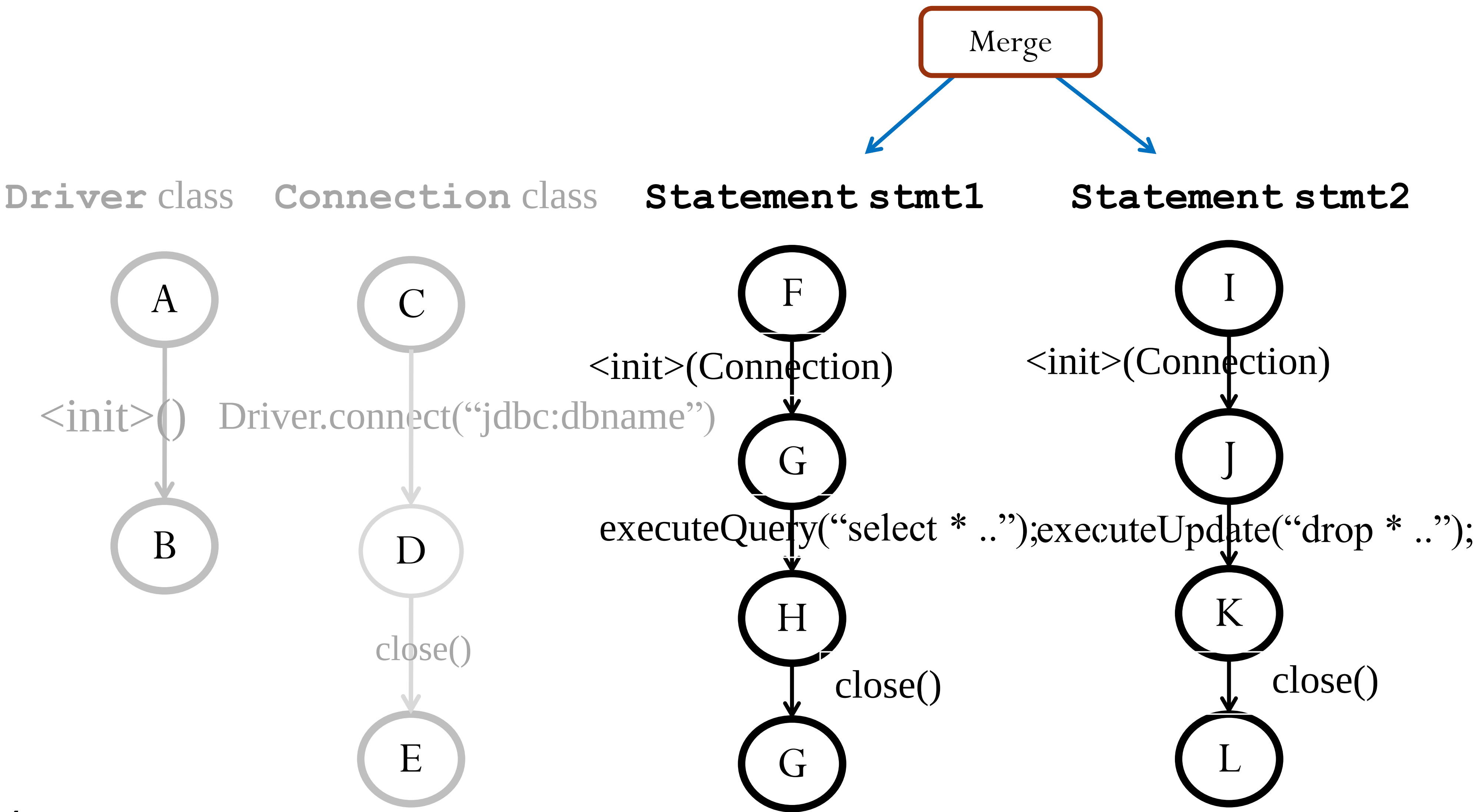
Model Inference for class `Statement`

```
Statement stmt2 =  new Statement(con);
stmt2.executeUpdate("drop table table_name");
stmt2.close();
```



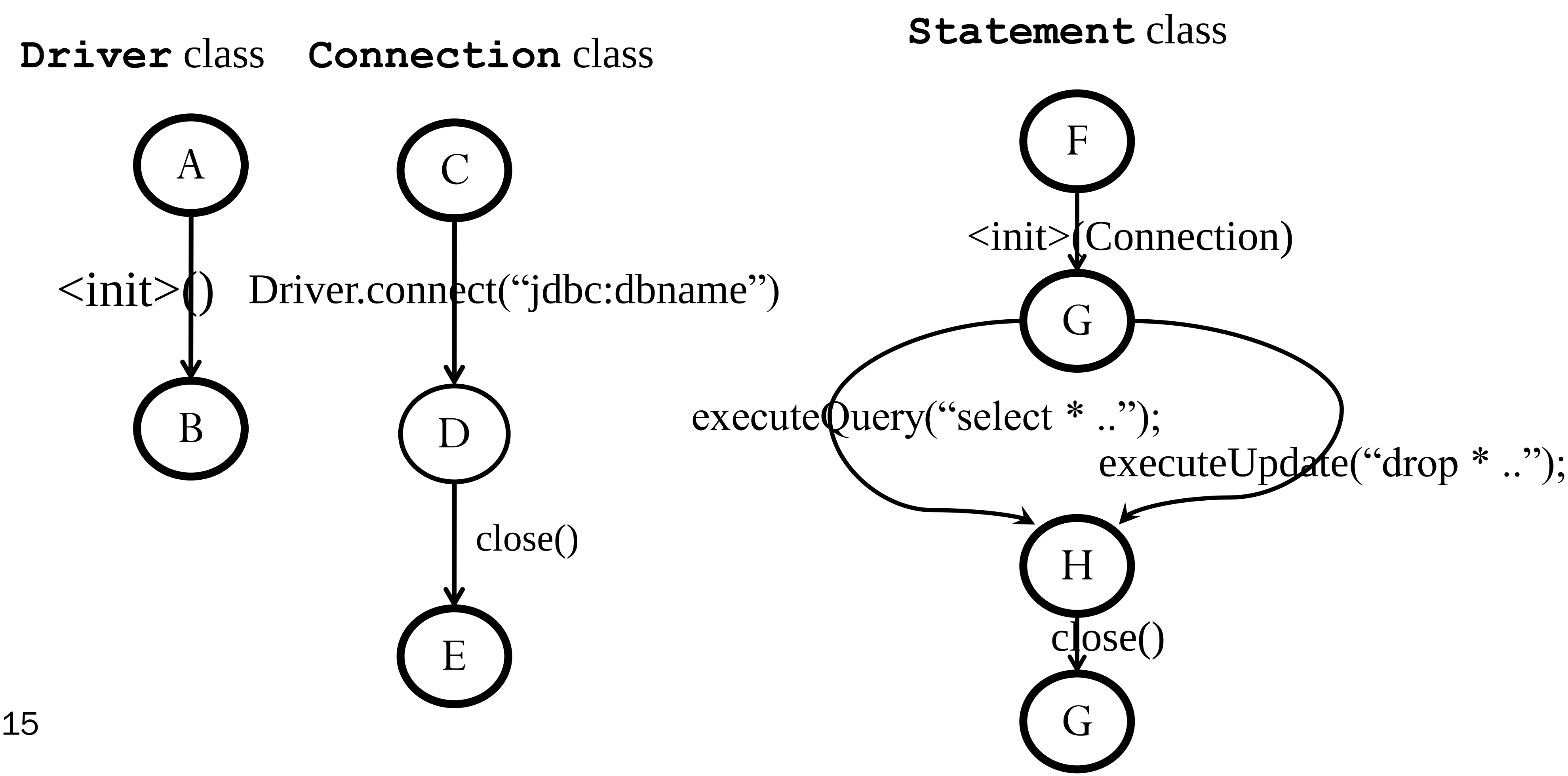
Construct a call sequence model for each **observed object**

Merge Models of the Same class

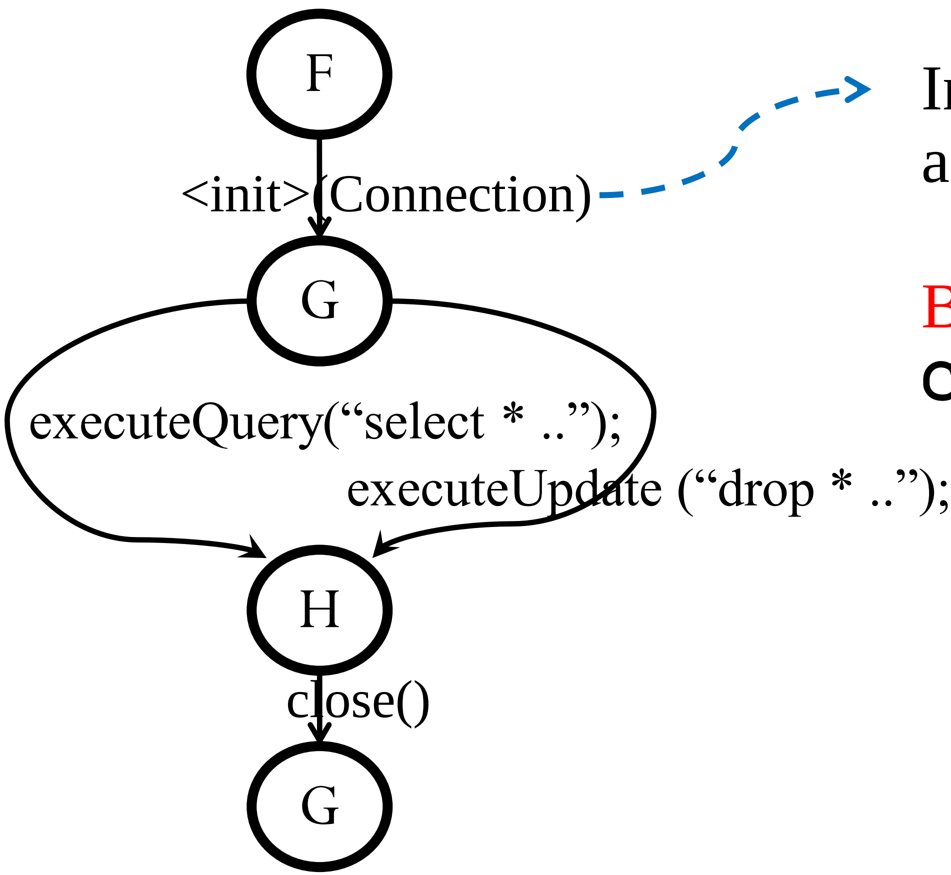


Merge models for all objects to form **one model per class**

Call Sequence Model after Merging



Enhance Call Sequence Models with Argument Constraints



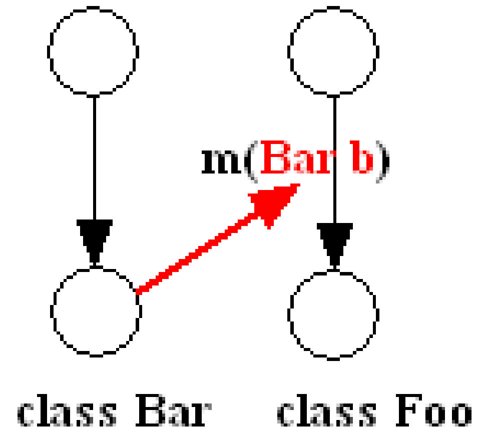
`Statement` class

Invoking the constructor requires a **`Connection`** object

But, how to choose a desirable **`Connection`** object ?

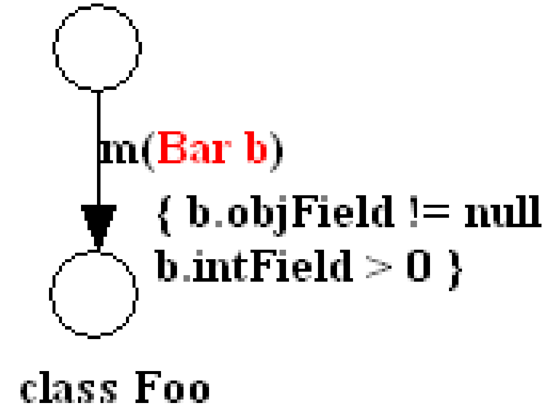
Argument Constraints

- **Argument dependence constraint**
  - Record where the argument object values **come from**
  - Add dependence edges in the call sequence models



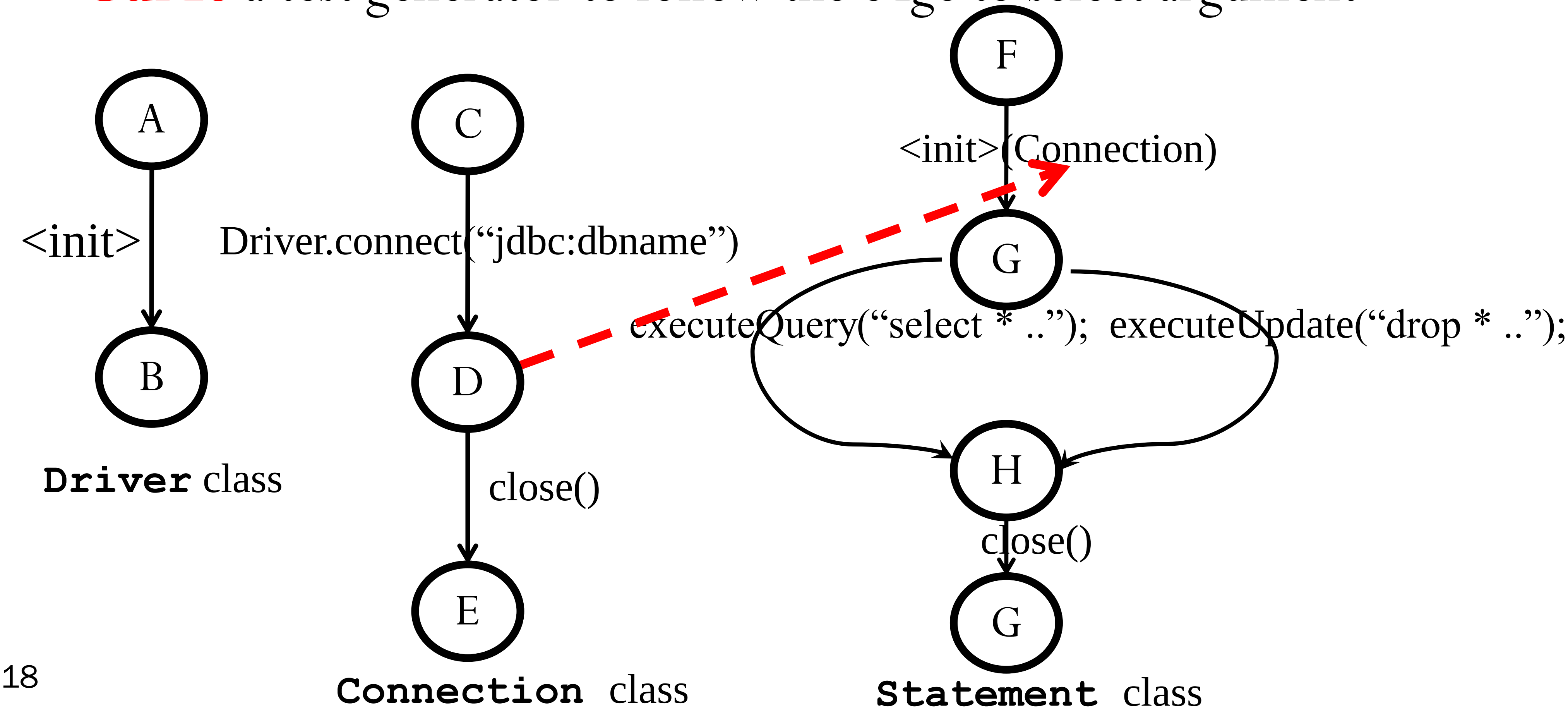
- **Abstract object profile constraint**
  - Record what the argument **value "is"**
  - Map each object field into an abstract domain

  as a coarse-grained measurement of "value similarity"



Argument Dependence Constraint

- Represent by a directed edge ($\dashrightarrow$ below)
- **Means**: transition **F** → **G** has data dependence on node **D**, it uses the result object at the node **D**
- **Guide** a test generator to follow the edge to select argument



Abstract Object Profile Constraint

- **For each field in an observed object**
  - Map the **concrete value** → an **abstract state**

Numeric value	→	`> 0,  = 0,  < 0`
Object	→	`= null,  != null`
Array	→	`empty, null, not_empty`
Bool / enum values	→	not abstracted

- **Annotate model edges with abstract object profiles of the observed argument values from dynamic analysis**

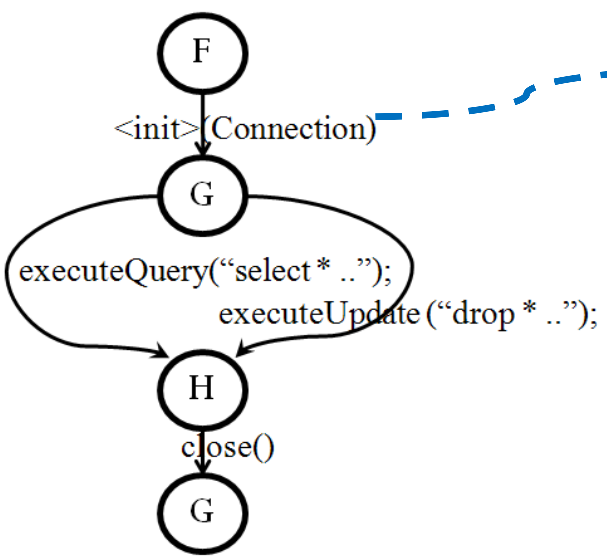
- **Guide** test generator to choose arguments similar to what was seen at runtime

Annotate Model Edges with Abstract Object Profiles

- Class **Connection** contains 3 fields

```
Driver driver;  String url;  String usr;
```

- All observed valid **Connection** objects have a profile like:
  {**driver != null, url != null, usr != null**}
  - Annotate the method-call edge: **<init>(Connection)**



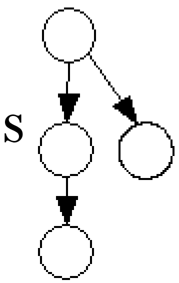
Argument **Connection**’s profile:
{driver != null, url != null, usr !=null}

Palus prefers to pick an argument with the **same profile**, when invoking : **<init>(Connection)**

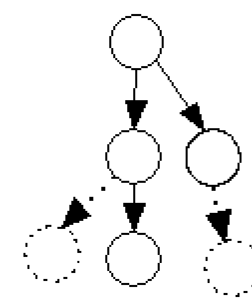
 Statement class

(2) Static Method Analysis

- Dynamic analysis is accurate, but **incomplete**
  - May fail to cover some methods or method invocation orders



- Palus uses static analysis to **expand** the dynamically-inferred model
  - Identify related methods, and test them together
  - Test methods not covered by the sample trace



Statically Identify Related Methods

- Two methods that access the same fields may be related (conservative)

- Two relations:
  - **Write-read**: method A reads a field that method B writes
  - **Read-read**: methods A and B reference the same field

Statically Recommends Related Methods for Testing

- Reach more program states
  - Call **`setX()`** before calling **`getX()`**

- Make the sequence more behaviorally-diverse
  - A correct execution observed by dynamic analysis will never contain:

```
Statement.close();
Statement.executeQuery("…")
```

  - But static analysis may suggest to call **`close()`** before **`executeQuery("…")`**

Weighting Pair-wise Method Dependence

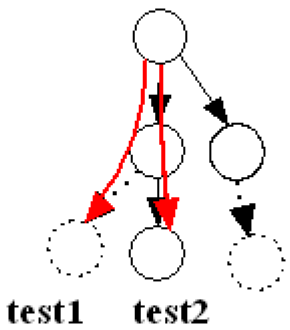
- **tf-idf weighting scheme** [Jones, 1972]
  - Palus uses it to measure the **importance** of a **field** to a **method**

$$tfidf\,(\boldsymbol{field}, \boldsymbol{method}) \propto \frac{\text{frequency of } \boldsymbol{field} \text{ in } \boldsymbol{method}}{\text{frequency of } \boldsymbol{field} \text{ in } \boldsymbol{all\ methods}}$$

- Dependence weight between two methods:

$$\boldsymbol{Weight}(\mathrm{m1}, \mathrm{m2}) = \sum_{\mathbf{f} \in \mathrm{OverlapFields}(\mathrm{m1}, \mathrm{m2})} \boldsymbol{tfidf}(\mathbf{f}, \mathbf{m1})$$

(3) Model-Guided Random Test Generation: A *2-Phase* algorithm



- **Phase1**:

  Loop:

  1. Follow the **dynamically-inferred model** to select **methods** to invoke
  2. For each selected **method**

     2.1 Choose arguments using:

     - Argument dependent edge
     - Captured abstract object profiles
     - Random selection

     2.2 Use **static method dependence** information to invoke related methods

- **Phase 2:**

  Randomly generate sequences for **model-uncovered** methods

  - Use **feedback-directed** random test generation [ICSE'07]

Specify Testing Oracles in JUnit Theory

- A project-specific testing oracle in JUnit theory

```
@Theory
public void checkIterNoException(Iterator it) {
  assumeNotNull(it);
  try {
    it.hasNext();
  } catch (Exception e) {
    fail("hasNext() should never throw exception!");
  }
}
```

Palus checks that, for **every** **`Iterator`** object, calling **`hasNext()`** should **never** throw exception!

Outline

- Motivation
- Approach
  - Dynamic model inference
  - Static model expansion
  - Model-guided test generation
- Evaluation
- Related Work
- Conclusion and Future Work

Research Questions

- Can tests generated by Palus achieve **higher structural coverage**
- Can Palus find (more) **real-world bugs**?
- Compare with three existing approaches:

Approaches	Dynamic	Static	Random
Randoop [ICSE'07]			●
Palulu [M-TOOS'06]	●		●
RecGen [ASE' 10]		●	●
Palus (**Our approach**)	●	●	●

Subjects in Evaluating Test Coverage

- 6 open-source projects

Program	Lines of Code
tinySQL	7,672
SAT4J	9,565
JSAP	4,890
Rhino	43,584
BCEL	24,465
Apache Commons	55,400

Many Constraints

Few Constraints

Experimental Procedure

- Obtain a sample execution trace by running a simple example from user manual, or its regression test suite
- Run each tool for until test coverage becomes saturated, using the same trace
- Compare the line/branch coverage of generated tests

Test Coverage Results

Approaches	Dynamic	Static	Random	Avg Coverage
Randoop [ICSE'07]			●	39%
Palulu [M-TOOS'06]	●		●	41%
RecGen [ASE' 10]		●	●	30%
Palus (**Our approach**)	●	●	●	**53%**

- Palus **increases** test coverage
  - Dynamic analysis helps to create **legal** tests
  - Static analysis / random testing helps to create **behaviorally-diverse** tests
- Palus **falls back** to **pure random** approach for programs with few constraints (Apache Commons)

Evaluating Bug-finding Ability

- Subjects:
  - The same 6 open-source projects
  - 4 large-scale Google products
- Procedure:
  - Check 5 **default Java contracts** for all subjects
  - Write 5 **simple theories** as additional testing oracles for Apache Commons, which has partial spec

Finding Bugs in 6 open-source Projects

- Checking default Java language contracts:
  - E.g., for a non-null object o: `o.equals(o)` returns true

	Dynamic	Static	Random	Bugs
Randoop [ICSE'07]			●	80
Palulu [M-TOOS'06]	●		●	76
RecGen [ASE' 10]		●	●	42
Palus (**Our approach**)	●	●	●	**80**

  - Finds **the same number** of bugs as Randoop

- Writing additional theories as testing oracle
  - Palus finds **one new bug** in Apache Commons
    - `FilterListIterator.hasNext()` throws exception
    - **Confirmed** by Apache Commons developers

Finding Bugs in 4 Google Products

- 4 large-scale Google products

Google Product	Number of tested classes
Product A	238
Product B	600
Product C	1,269
Product D	1,455

- Each has a regression test suite with 60%+ coverage
- Go through a rigorous peer-review process

Palus Finds More Bugs

- **Palus** finds **22** real, previously-unknown **bugs**

	Dynamic	Static	Random	Bugs
Randoop [ICSE'07]			●	19
Palulu [M-TOOS'06]	●		●	18
RecGen [ASE' 10]		●	●	--
Palus (**Our approach**)	●	●	●	22

  - **3 more** than existing approaches

- Primary reasons:
  - *Fuzz* a long specific *legal* path
  - Create a ***legal*** test, ***diversify*** it, and reach program states that have not been reached before

Outline

- Motivation
- Approach
  - Dynamic model inference
  - Static model expansion
  - Model-guided test generation
- Evaluation
- Related Work
- Conclusion and Future Work

Related Work

- Automated Test Generation
  - **Random approaches**: **Randoop** [ICSE'07], **Palulu** [M-Toos'06], **RecGen**[ASE'10]

    Challenge in creating legal / behaviorally-diverse tests
  - **Systematic approaches**: **Korat** [ISSTA'02], Symbolic-execution-based approaches (e.g., **JPF**, **CUTE**, **DART**, **KLEE**…)

    Scalability issues; create test inputs, not object-oriented method sequences
  - **Capture-replay -based approaches**: **OCAT** [ISSTA'10], **Test Factoring** [ASE'05] and **Carving** [FSE'05]

    Save object states in memory, not create method sequences

- Software Behavior Model Inference
  - **Daikon** [ICSE'99], **ADABU** [WODA'06], **GK-Tail** [ICSE'08] …

    For program understanding, not for test generation

Outline

- Motivation
- Approach
  - Dynamic model inference
  - Static model expansion
  - Model-guided test generation
- Evaluation
- Related Work
- Conclusion and Future Work

Future Work

- Investigate **alternative ways** to use program analysis techniques for test generation
  - How to *better* combine static/dynamic analysis?
- What is **a good abstraction** for automated test generation tools?
  - We use an enhanced call sequence model in Palus, what about other models?
- Explain **why a test fails**
  - Automated Documentation Inference [ASE'11 to appear]
  - Semantic test simplification

Contributions

- A hybrid automated test generation **technique**
  - Dynamic analysis: infer model to create legal tests
  - Static analysis: expand dynamically-inferred model
  - Random testing: create behaviorally-diverse tests
- A publicly-available **tool**

  http://code.google.com/p/tpalus/
- An empirical **evaluation** to show its effectiveness
  - Increases test coverage
  - Finds more bugs

Backup slides

Sensitivity to the Inputs

- Investigate on two subjects: **tinySQL** and **SAT4J**

Subject	Input Size	Coverage
tinySQL	10 SQL Statements	59%
	ALL Statements from Manual	61%
SAT4J	A 5-clause formula	65%
	A 188-clause formula	66%
	A 800-clause formula	66%

- This approach is **not very sensitive** to the inputs
  - Not too many constraints in subjects?

Breakdown of Contributions in Coverage Increase