Plural and PLAID :

Protocols in Practice

Jonathan Aldrich

Workshop on Behavioral Types

April 2011

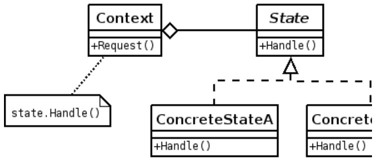
Carnegie Mellon University
School of Computer Science

Empirical Study: Protocols in Java

- Object Protocol [Beckman, Kim, & A – to appear in ECOOP 2011]
  - Finite set of abstract states, among which an object will transition
  - Clients must be aware of the current state to use an object correctly
- Question: how commonly are protocols defined & used?
  - Corpus study on 2 million LOC: Java standard library, open source
- Results
  - 7% of all types define object protocols
    - c.f. 2.5% of types define type parameters using Java Generics
  - 13% of all classes act as object protocol clients
  - 25% of these protocols are in classes designed for concurrent use

Empirical Study: Protocols in Java

- Empirically discovered “protocol design patterns”

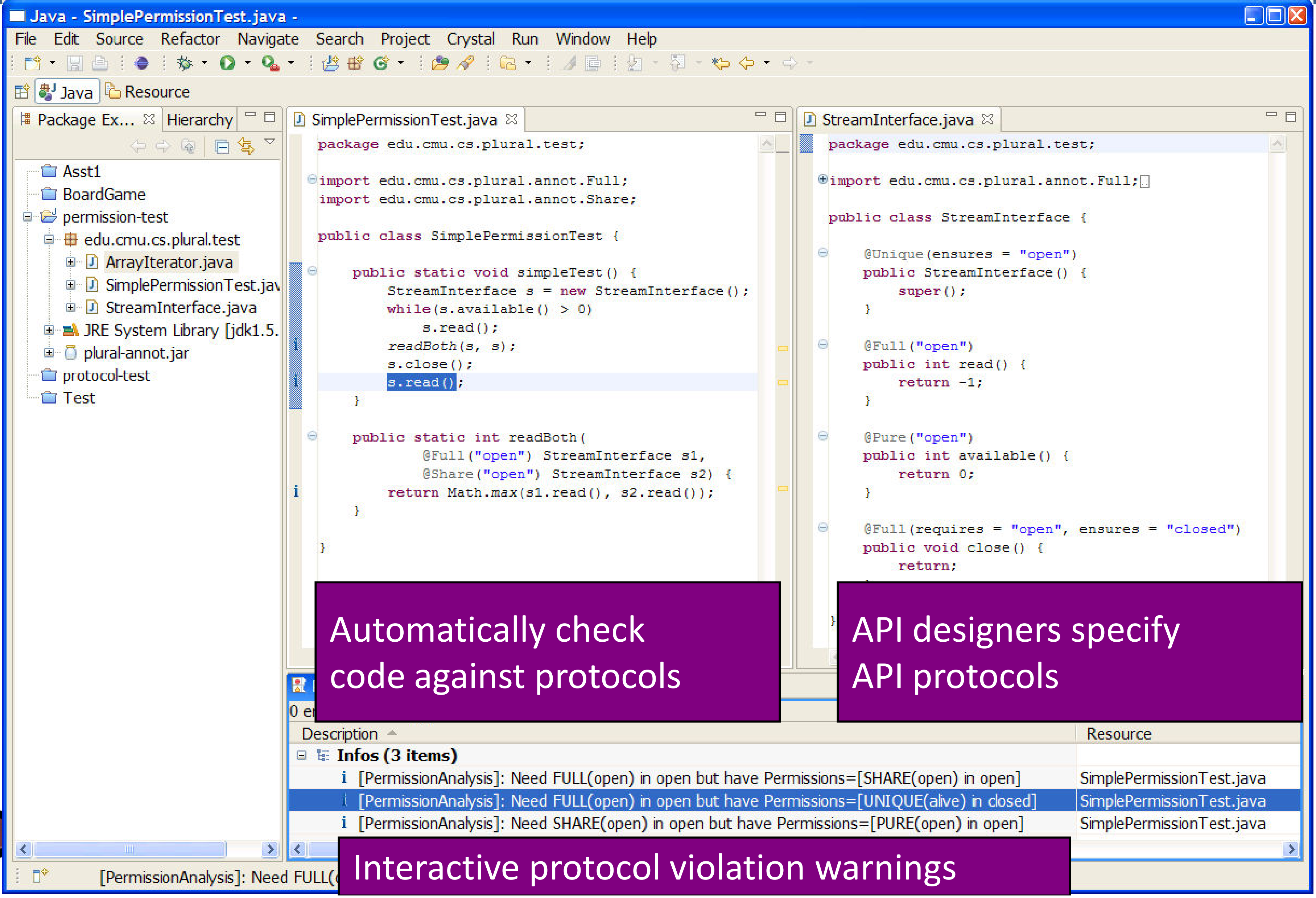


  - **28% Initialization** before use – e.g. init(), open(), connect()
  - **26% Deactivation** – e.g. close()
  - **16% Type Qualifier** – marks a subset of objects with an interface, e.g. immutable collections
  - **8% Preparation** – e.g. call mark() before reset() on a stream
  - **8% Boundary** check – e.g. hasNext()
  - **7% Non-redundancy** – can only call a method once, e.g. setCause()
  - **5% Domain Mode** – one or more domain-specific modes can be enabled and disabled, thereby enabling or disabling a group of methods, e.g. compression modes for javax.imageio.ImageWriteParam
  - **2% Others** (lifecycle protocols, strict lock/unlock alternation)

Plural: Typestate Checking for Java

- Plural: static checking of object protocol use in Java [Bierhoff & A 2007]
  - Checks which methods are available at each program point
  - Similar goals to [Gay, Vasconcelos, Ravara, Gesbert, Caldeira 2010]
    - but we focus only within a program, not on distributed systems
- Approach: type-like annotations
  - Typestate formalism
    - Vs. session types: named states, nominal subtyping
  - Supports external and internal choice
  - Verifies that implementation is safe with respect to interface
  - Affine, not linear (can forget an object)
  - Implementation in Eclipse: flow-sensitive static analysis based on type theory
- Distinguishing characteristics
  - Hierarchical and compositional specification of state space
  - Supports aliased objects through novel permission forms
  - Supports re-entrant code
  - Supports borrowing as well as internal uses of **this**
  - Checks typestate in the presence of concurrency

Plural: Typestate in Java



```
package edu.cmu.cs.plural.test;

import edu.cmu.cs.plural.annot.Full;
import edu.cmu.cs.plural.annot.Share;

public class SimplePermissionTest {

    public static void simpleTest() {
        StreamInterface s = new StreamInterface();
        while(s.available() > 0)
            s.read();
        readBoth(s, s);
        s.close();
        s.read();
    }

    public static int readBoth(
            @Full("open") StreamInterface s1,
            @Share("open") StreamInterface s2) {
        return Math.max(s1.read(), s2.read());
    }

}
```

```
package edu.cmu.cs.plural.test;

import edu.cmu.cs.plural.annot.Full;

public class StreamInterface {

    @Unique(ensures = "open")
    public StreamInterface() {
        super();
    }

    @Full("open")
    public int read() {
        return -1;
    }

    @Pure("open")
    public int available() {
        return 0;
    }

    @Full(requires = "open", ensures = "closed")
    public void close() {
        return;
```

0 er

Description	Resource
Infos (3 items)	
i [PermissionAnalysis]: Need FULL(open) in open but have Permissions=[SHARE(open) in open]	SimplePermissionTest.java
i [PermissionAnalysis]: Need FULL(open) in open but have Permissions=[UNIQUE(alive) in closed]	SimplePermissionTest.java
i [PermissionAnalysis]: Need SHARE(open) in open but have Permissions=[PURE(open) in open]	SimplePermissionTest.java

[PermissionAnalysis]: Need FULL(

Automatically check code against protocols

API designers specify API protocols

Interactive protocol violation warnings

Plural Case Studies

- JabRef, 74 kLOC multithreaded BibTeX tool
  - APIs verified: Timers, sockets, readers, XML nodes, Tree data structures, 9 others…
  - 4 bugs found
- JSpider, 9 kLOC multithreaded web robot
  - Verified Task protocol with ownership transfer
  - 2 bugs found
- PMD, 35 kLOC static analysis tool
  - Verified iterator usage
- JDBC, 10 kLOC database access interface
  - Specified complex protocol: 838 annotations on 440 methods
- Apache Beehive, 2 kLOC resource access library
  - Implements iterator interface in terms of JDBC
- Results
  - Low false positive rate: approx 1 per 400 LOC
  - Low annotation overhead: from 1/25 to 1/200 LOC (depends on protocol use)
  - Covers all protocols we see in informal documentation, but more succinctly

Plural Case Study Observations

- Aliasing was common in our case studies
    - Views or iterators over a collection in PMD
    - Shared resources (e.g. JDBC interfaces in Beehive)
- Many protocols are not documented or dynamically enforced
- State tests are common
    - hasNext(), isEmpty(), etc.
- Intersection types for methods: A -> B & C -> D
- Many uses of type qualifiers (“marker” states)
- Borrowing is common
    - Temporarily “capturing” a reference (e.g. iterators over a collection)
    - Temporary use of values from getters

Queue, Racy Client Usage

```
final Queue<String> q = new Queue<String>();
(new Thread() {
  public void run() {
    while( !q.is_closed() ) {
      String s = q.dequeue();
      System.out.println(“Got: ” + s);
    }
  }}).start();

for(int i=0;i<5;i++)
  q.enqueue(“Object ” + i);

Thread.sleep(4000);
q.close();
```

Consumer Thread

Race!

Producer Thread

Plaid: a Typestate-Oriented Language

- What does typestate-oriented mean?

  programs are made up of dynamically created **objects**,

  each object has a **typestate** that is **changeable**

  and each typestate has an **interface**, **representation**, and **behavior**.

- Why organize a language around typestate?
  - Typestate is common and important!
  - Cleaner typestate specification and verification
  - Expressive object model
  - Cleaner invariant checking

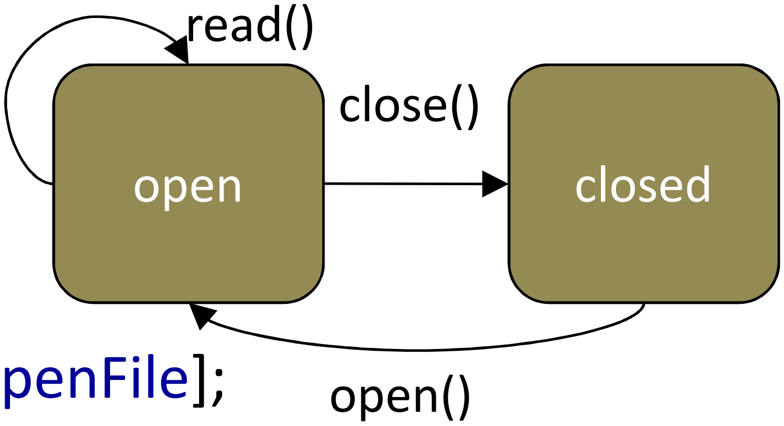
Plaid: a Typestate-Oriented Language

```
state File {
    val String filename;
}
state ClosedFile = File with {
    method void open() [ClosedFile>>OpenFile];
}
state OpenFile = File with {
    private val CFile fileResource;

    method int read();
    method void close() [OpenFile>>ClosedFile];
}
```

State transition

Different representation

New methods



Implementing Typestate Changes

```
method void open() [ClosedFile>>OpenFile] {
    this <- OpenFile {
        fileResource = fopen(filename);
    }
}
```

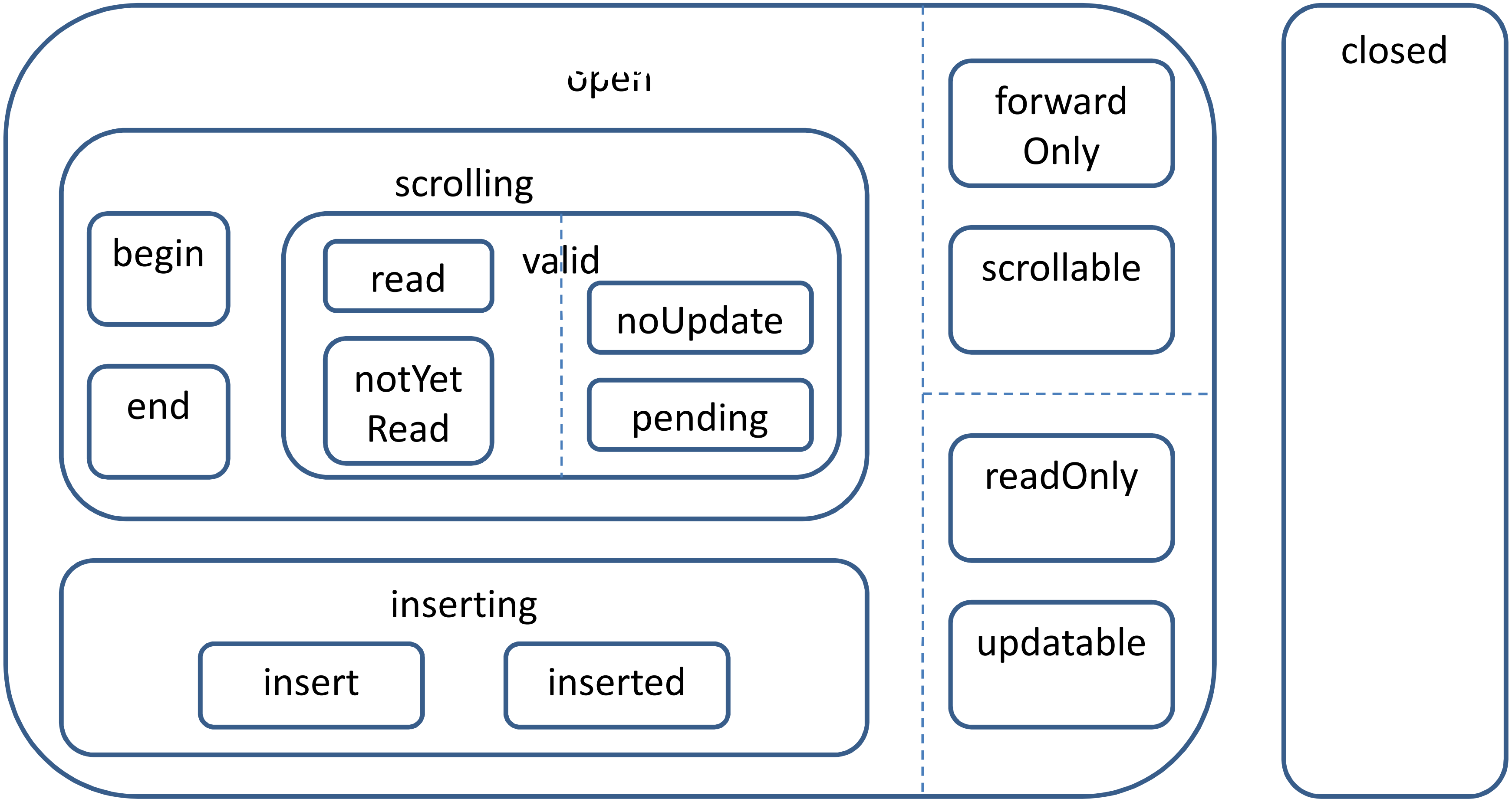
Typestate change primitive – like Smalltalk *become*

Values must be specified for each new field

:

State Protocols are Complex

Java Database Connectivity (JDBC) Library State Space



open

scrolling

begin

end

valid

read

notYet Read

noUpdate

pending

inserting

insert

inserted

forward Only

scrollable

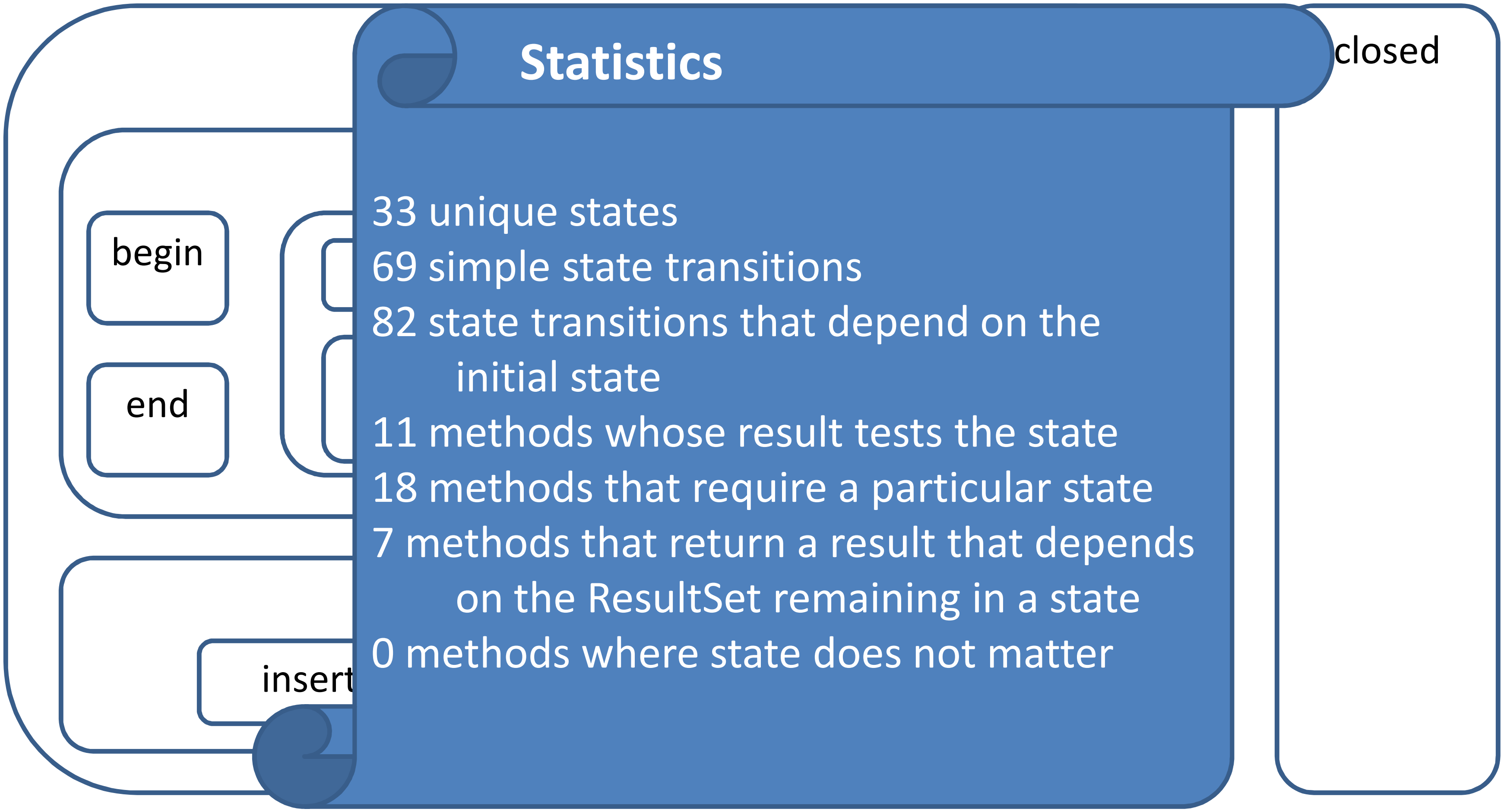
readOnly

updatable

closed

State Protocols are Complex

Java Database Connectivity (JDBC) Library State Space

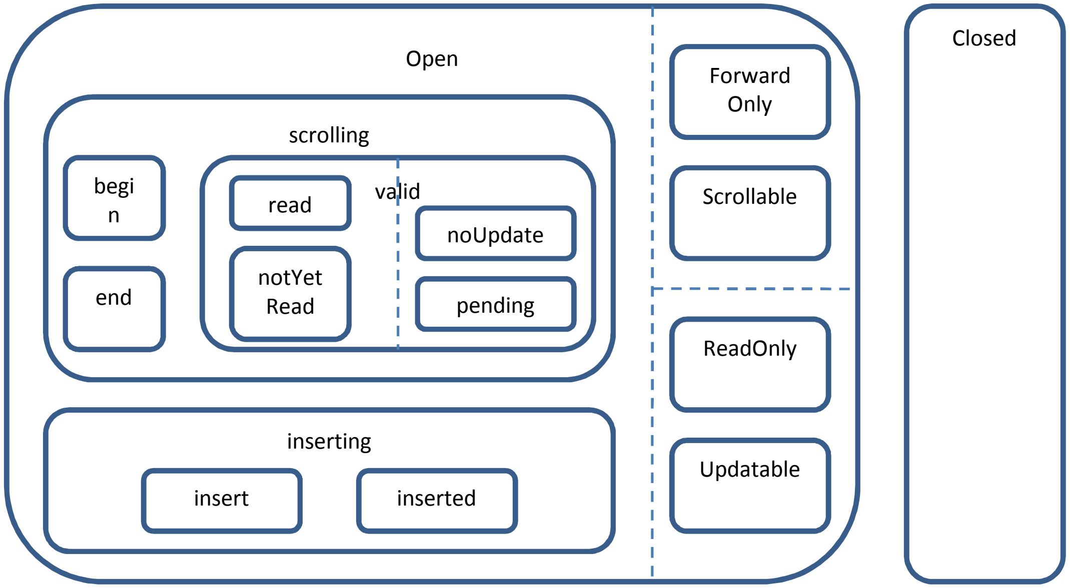


Statistics

33 unique states
69 simple state transitions
82 state transitions that depend on the initial state
11 methods whose result tests the state
18 methods that require a particular state
7 methods that return a result that depends on the ResultSet remaining in a state
0 methods where state does not matter

Modeling JDBC in Plaid

```
state ResultSet = …
state Open case of ResultSet =
    Direction with Status with Action
state Closed case of ResultSet;

state Direction;
state ForwardOnly case of Direction;
state Scrollable case of Direction

state Status …
```

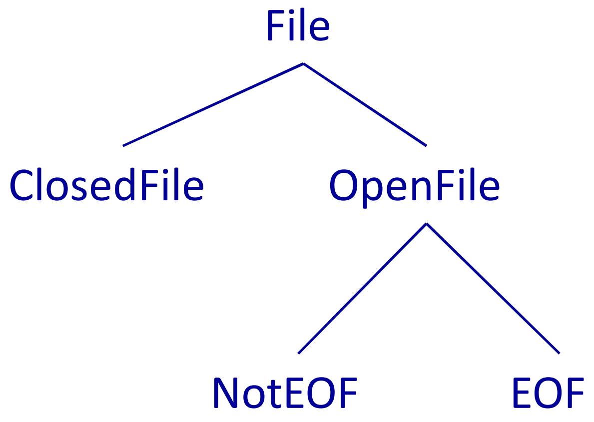


case of hierarchies model alternatives (OR-states)
state composition (“**with**”) models orthogonal state spaces (AND-states)

Typestate Permissions

- **unique** OpenFile
  - File is open; no aliases exist
  - Default for mutable objects
- **immutable** OpenFile
  - Cannot change the File
    - Cannot close it
    - Cannot write to it, or change the position
  - Aliases may exist but do not matter
  - Default for immutable objects

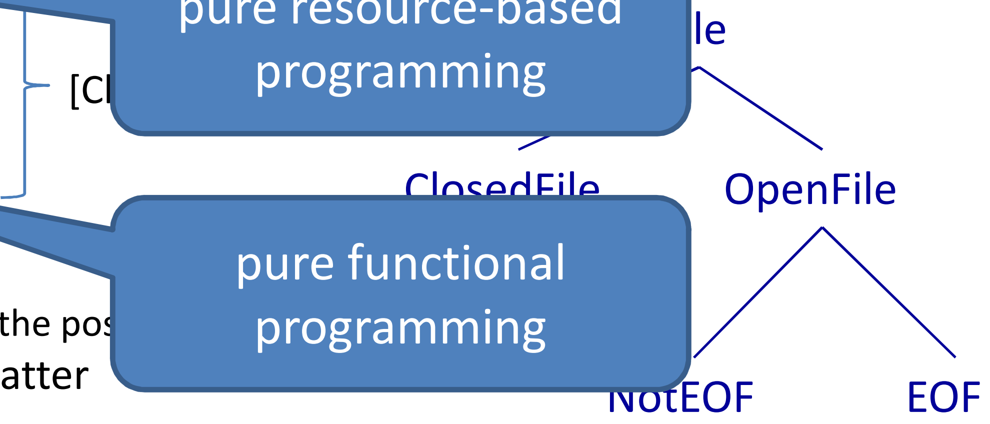
[Chan et al. ’98]



- **shared** OpenFile@NotEOF [OOPSLA ’07]
  - File is aliased
  - File is currently not at EOF
    - Any function call could change that, due to aliasing
  - It is forbidden to close the File
    - OpenFile is a *guaranteed* state that must be respected by all operations through all aliases
- **full** – like **shared** but is the exclusive writer
- **pure** – like **shared** but cannot write

Typestate Permissions

- **unique** OpenFile
  - File is open; no aliases exist
  - Default for mutable objects
- **immutable** OpenFile
  - Cannot change the File
    - Cannot close it
    - Cannot write to it, or change the pos
  - Aliases may exist but do not matter
  - Default for immutable objects
- **shared** OpenFile@NotEOF [OOPSLA '
  - File is aliased
  - File is currently not at EOF
    - Any function call could change that, due to alia
  - It is forbidden to close the File
    - OpenFile is a *guaranteed* state that must be respected by all operations through all aliases
- **full** – like **shared** but is the exclusive writer
- **pure** – like **shared** but cannot write



pure resource-based programming

pure functional programming

shared OpenFile@OpenFile is (almost) traditional object-oriented programming

Key innovations vs. prior work (c.f. Fugue, Boyland, Haskell monads, separation logic, etc.)

Permission Splitting

- Permissions may not be duplicated
  - No aliases to a unique object!

- Splitting that follows permission semantics is allowed, however
  - **unique → full**
  - **unique → shared**
  - **unique → immutable**
  - **shared → shared, shared**
  - **immutable → immutable, immutable**
  - **X → X**, **pure** *// for any non-**unique** permission X*

- How do we get unique back?
  - borrowing, fractions, or a dynamic test

Packing/Unpacking

- How to store a linear object in a non-linear object?

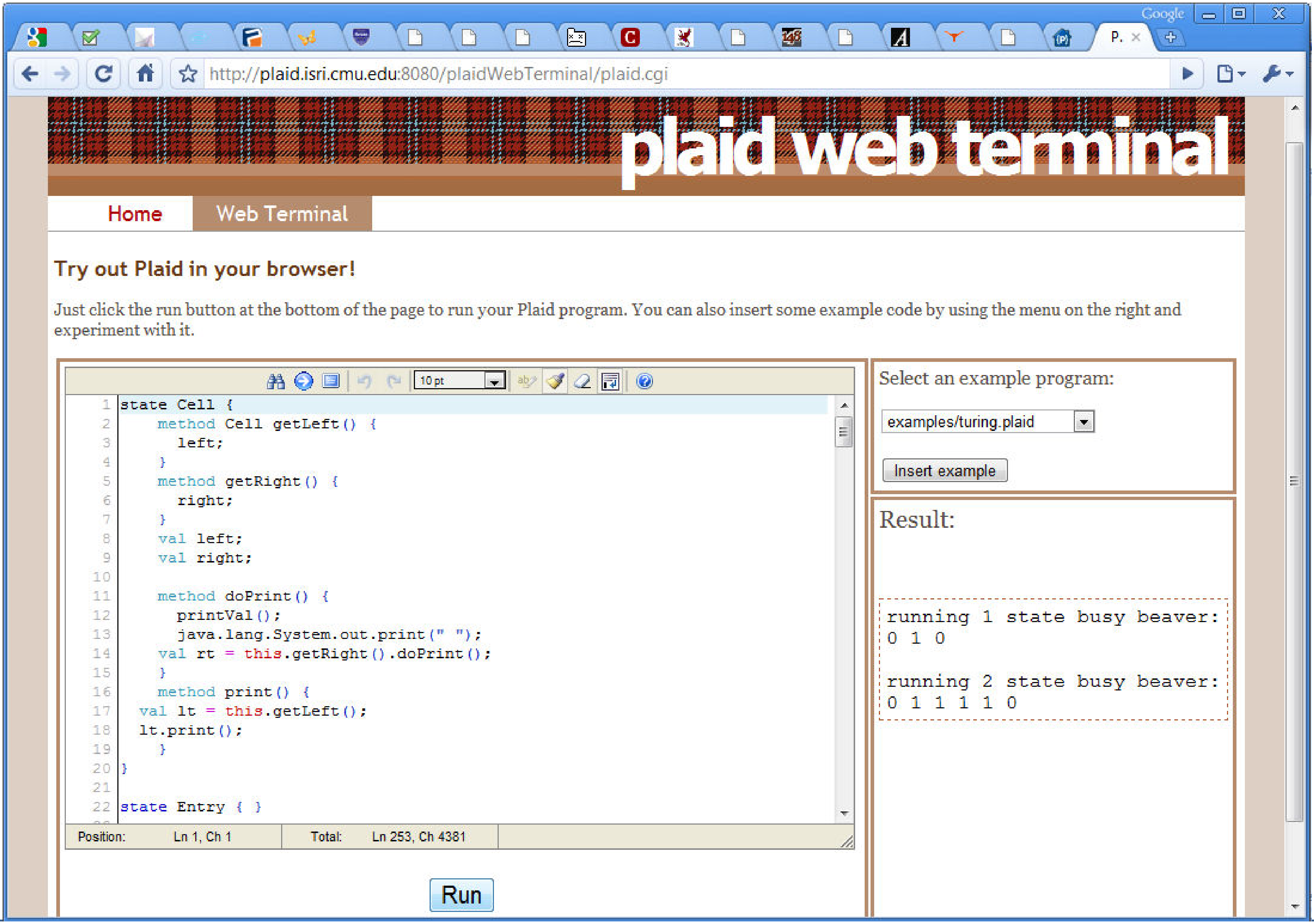
```
void operateOnMe() {
    // unpack object here, get field permissions
    uniqueField.doSomething();
    store(anotherUniqueField);
    anotherUniqueField = new UniqueObject();

    // pack object here, re-verify field permissions
    finishOperation(this);
}
```

- Re-entrancy
  - Permitted, but must ensure we do not unpack the same object twice
    - e.g. in a call back from doSomething()
  - Static check (e.g. with ownership) or dynamic check

Other (Eventual) Features of Plaid

- Concurrency by Default
  - Uses permissions to infer dataflow dependencies
  - Executes program in parallel subject to dependencies
- Dynamic state tests via pattern matching
- Recover **unique** via casts
  - supported via reference counting
- Gradual types
  - state-based modeling useful even if states are checked dynamically
- First-class state objects, trait-like composition operators
- Good support for functional programming
- Strong information hiding guarantees

Try (dynamic) Plaid!



Try out Plaid in your browser!

Just click the run button at the bottom of the page to run your Plaid program. You can also insert some example code by using the menu on the right and experiment with it.

```
state Cell {
  method Cell getLeft() {
   left;
  }
  method getRight() {
   right;
  }
  val left;
  val right;

  method doPrint() {
   printVal();
   java.lang.System.out.print(" ");
  val rt = this.getRight().doPrint();
  }
  method print() {
 val lt = this.getLeft();
 lt.print();
  }
}

state Entry { }
```

Position: Ln 1, Ch 1 Total: Ln 253, Ch 4381

Select an example program: examples/turing.plaid

Result:

```
running 1 state busy beaver:
0 1 0

running 2 state busy beaver:
0 1 1 1 1 0
```

Plural and Plaid: Protocols in Practice

- Empirical evidence regarding object protocols
  - Protocols are common – 7% define, 13% use
  - Fall into common patterns, useful for evaluating specifiers and checkers

- Challenging but real requirements for effective static checking
  - Object aliasing – temporary and permanent
  - Hierarchical state spaces
  - State tests
  - Concurrent sharing of protocol-defining objects (≥25% of cases)
  - Reentrant code
  - Linear objects stored in nonlinear objects

- Plaid: native integration of state into the object model
  - First-class abstractions for characterizing state change
  - Use permission flow to infer concurrent execution
  - Practical mix of static & dynamic checking

http://www.plaid-lang.org/