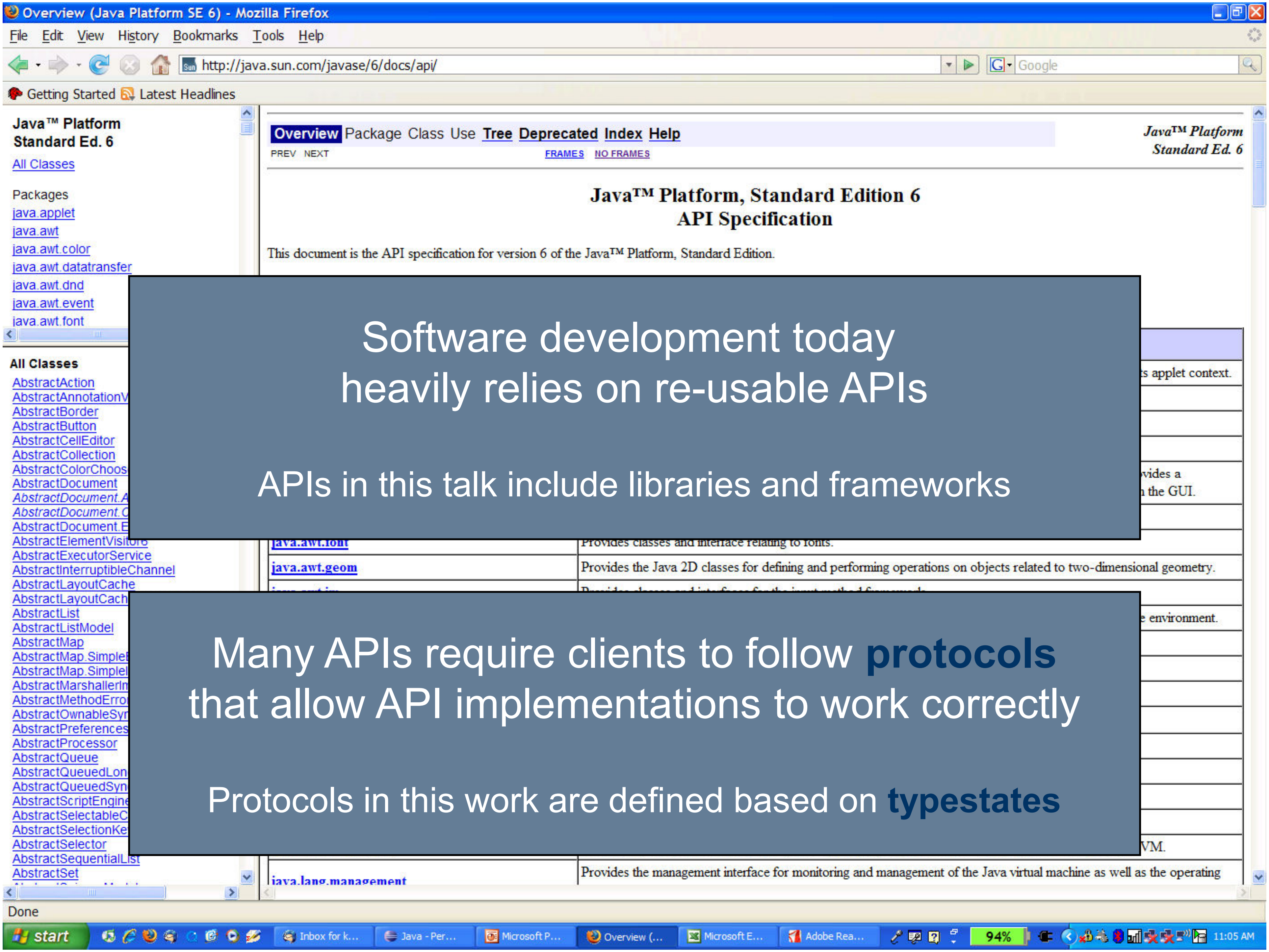
Pragmatic Typestate Verification with Permissions

Jonathan Aldrich
Carnegie Mellon University

with Kevin Bierhoff,
Nels Beckman,
Sven Stork,
and Yoon Phil Kim

Spring 2010

(Yoon Phil Kim not pictured)



Software development today
heavily relies on re-usable APIs

APIs in this talk include libraries and frameworks

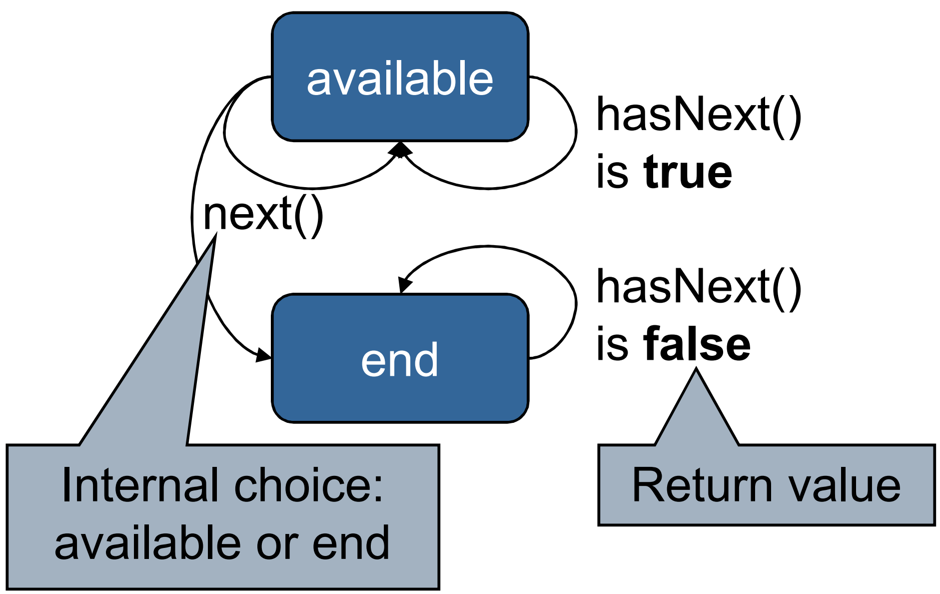
Many APIs require clients to follow **protocols**
that allow API implementations to work correctly

Protocols in this work are defined based on **typestates**

Protocol Examples:
Iterators and InputStreams

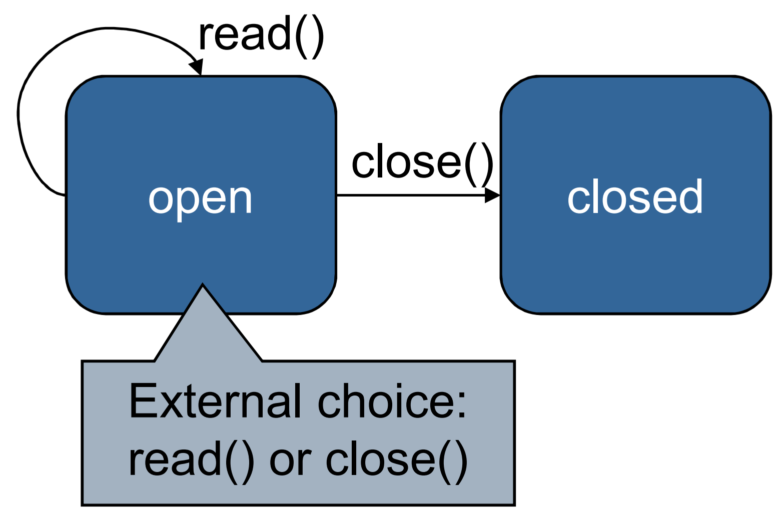
Iterators

- Return all elements of a sequence



InputStreams

- Read from a character stream



Typestates: modular static analysis tracks current "state" of objects

APIs are hard to use and implement

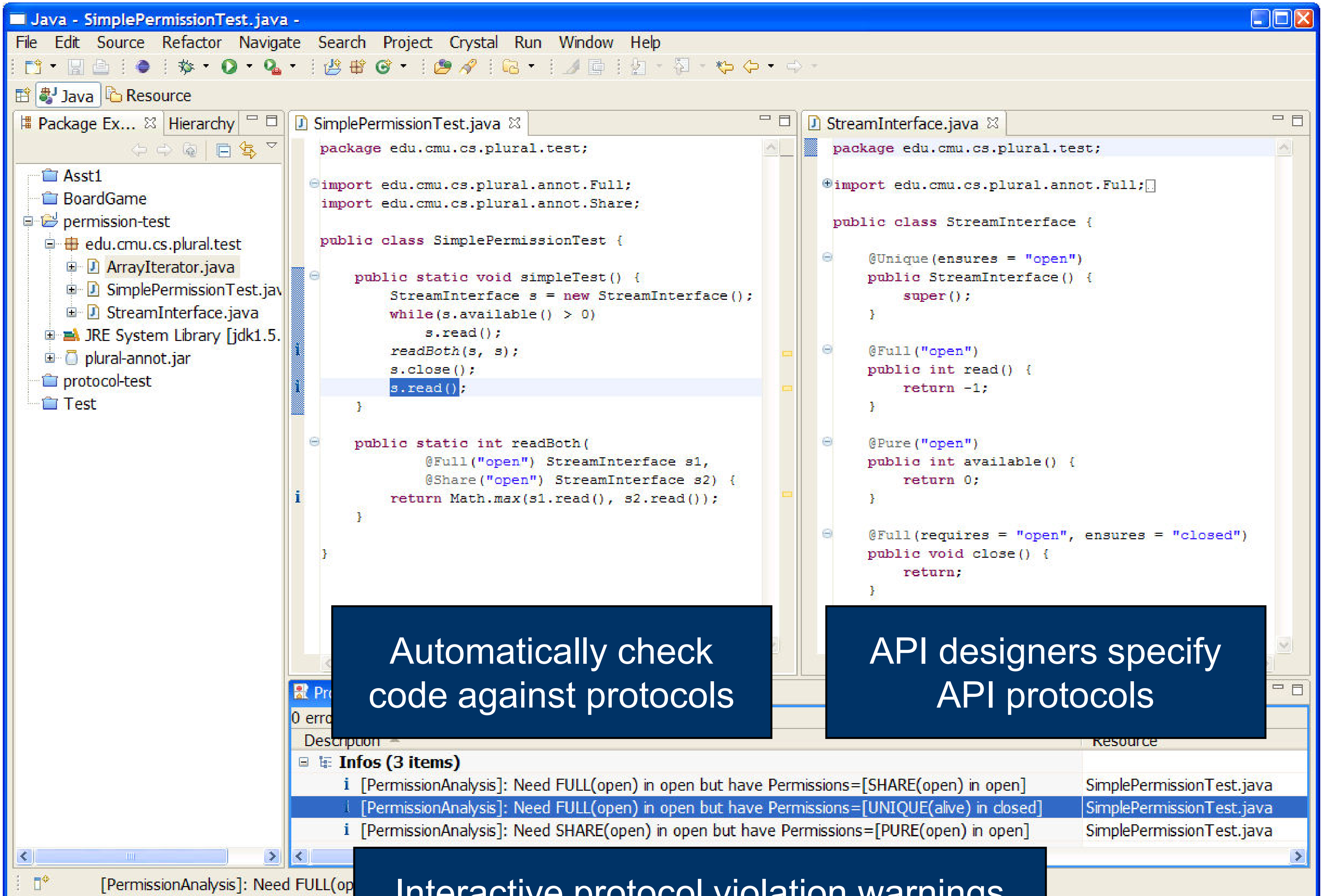
API users (clients)

- Difficult to understand correct usage
- Incorrect use does not always lead to clear errors
- Hard to guarantee protocol is followed on all paths
- Code modifications may introduce new errors

API implementers

- Documentation consistent with actual code
- Consistent runtime tests to protect against misuse
- Shared objects might be modified unexpectedly
- Reentrancy
- Unexpected overriding and open recursion

Implementation of one API will often use other APIs



Automatically check code against protocols

API designers specify API protocols

Interactive protocol violation warnings

Checking protocol compliance is hard

```
s = new BufferedInputStream();
while((c = s.read()) >= 0)
    process(c);
s.close();
```

Problem: What if there are other references to the object c?

Client checking

```
private void process(int c) {
    if(valid(c)) { … }
    else s.close();
}
```

```
private void fill() {
    pos = 0;
    int cnt = underlyingStream.read(…);
    count = pos + cnt;
}
```

Implementation checking

Problems: Does this object use other objects correctly? What if multiple threads are involved?

```
public      ronized int read() {
    if (p    >= count) {
        fill();
        if (pos >= count) return -1;
    }
    return buf[pos++] & 0xff;
}
```

Key Challenges in Previous Work

Previous work provides ***static, modular*** checking of both ***clients*** and ***implementations***

But, previous work had serious limitations:

- Limited tracking of ***aliased*** state
- ***Nondeterministic*** state changes
- ***Concurrency***
- Dynamic ***state tests***
- States with ***representation*** and ***behavior***
- Verifying ***reentrant*** code
- ***Refining*** states in subclasses
- ***Reusing*** superclasses that are in different states
- ***Multi-object*** typestate

Contributions

Previous work provides ***static, modular*** checking of both ***clients*** and ***implementations***

Our contributions

- New modular approaches to tracking ***aliased*** state
- ***Nondeterministic*** state changes
- ***Concurrency***
- Dynamic ***state tests***
- States with ***representation*** and ***behavior***
- Verifying ***reentrant*** code
- ***Refining*** states in subclasses
- ***Reusing*** superclasses that are in different states
- ***Multi-object*** typestate

Outline

Previous work provides ***static, modular*** checking of both ***clients*** and ***implementations***

Our contributions

- New modular approaches to tracking ***aliased*** state
- ***Nondeterministic*** state changes
- ***Concurrency***
- Dynamic ***state tests***
- States with ***representation*** and ***behavior***

Typestate Specification

states open, closed
class StreamProtocol {
    true ⇒ **unique**(*this*) **in** open
    public StreamProtocol() { … }

    full(*this*) **in** open
    ⇒ **full**(*this*) **in** open
    public int read() { … }

    full(*this*) **in** open
    ⇒ **full**(*this*) **in** closed
    public void close() { … }
}

- Declare states open, closed
- Constructor returns **unique** permission to open stream
- Read requires **full** (exclusive write) access to open stream
- Close transitions from open to closed

Typestate Verification

```
states open, closed
class StreamProtocol {
    true ⇒ unique(this) in open
    public StreamProtocol() { … }

    full(this) in open
    ⇒ full(this) in open
    public int read() { … }

    full(this) in open
    ⇒ full(this) in closed
    public void close() { … }
}
```

```
StreamProtocol s = new StreamProtocol();
        unique(s) in open
while(s.available() > 0)
    s.read();     // precondition satisfied
        unique(s) in open
s.close();
        unique(s) in closed
s.read();     // error: require open state
```

Modular Typestate Verification

```
states open, closed
class StreamProtocol {
    true ⇒ unique(this) in open
    public StreamProtocol() { … }

    full(this) in open
    ⇒ full(this) in open
    public int read() { … }

    full(this) in open
    ⇒ full(this) in closed
    public void close() { … }
}
```

```
    full(s) in open ⇒ full(s) in open
void process(StreamProtocol s) {
        full(s) in open
    s.read();    // precondition satisfied
        full(s) in open
}

StreamProtocol s = new StreamProtocol();
        unique(s) in open
while(s.available() > 0)
    process(s); // precondition satisfied
        unique(s) in open
s.close();
        unique(s) in closed
```

DEMONSTRATION - PLURAL

- Plural.test.StreamProtocol

Implementation Verification

states open, closed

class StreamWrapper {

invariant open: **full**(*str*) **in** open

invariant closed: **full**(*str*) **in** closed

private StreamProtocol str;

full(*s*) **in** open ⇒ **unique**(*this*) **in** open

StreamWrapper(StreamProtocol s)

{

unpacked(*this*, **unique**) ⊗

full(*s*) **in** open

str = s;

pack *this* **to** open;

}

full(*this*) **in** open ⇒ **full**(*this*) **in** closed

public void close() {

full(*this*) **in** open

unpack *this*;

unpacked(*this*, **full**) ⊗

full(*str*) **in** open

str.close(); *// precondition satisfied*

unpacked(*this*, **full**) ⊗

full(*str*) **in** closed

pack *this* **to** closed;

full(*this*) **in** closed

}

Implementation Verification (2)

```
states open, closed
class StreamWrapper extends
    StreamProtocol {
    invariant open: super in open
    invariant closed: super in closed
```

- The example can also be done using inheritance
  - Subclass has permission to superclass state
  - Superclass in open when subclass is in open
- New contribution: Subclass and superclass can be in different states
  - E.g. subclass reads file to end and closes it
  - Then superclass is closed when subclass is still open

Access Permission Taxonomy

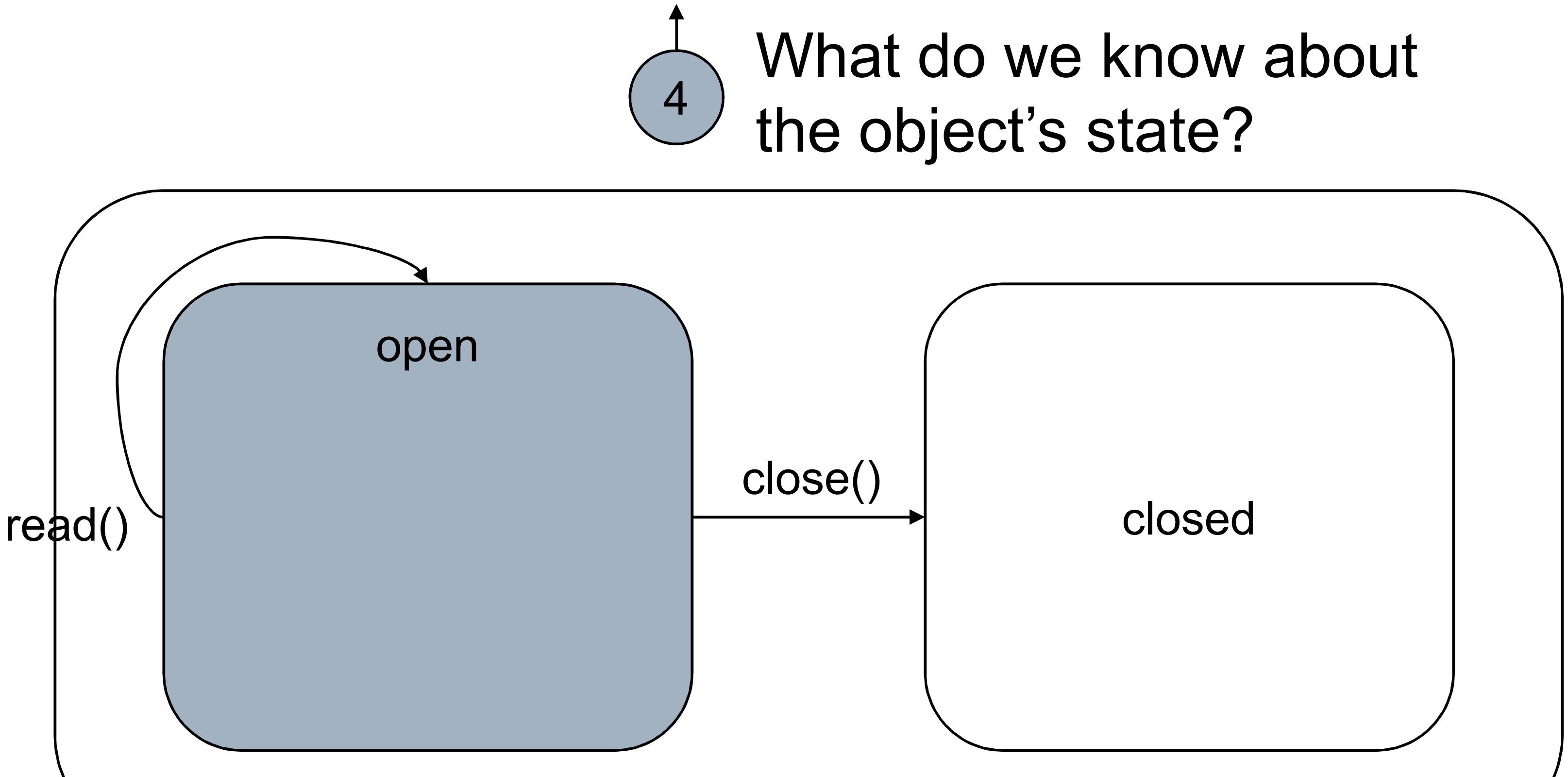
Example: share(s), where s is a program variable

1 What kinds of references exist?

	Current reference	
Other references	Read/write	Read-only
None	**unique**	—
Read/write	**share**	**pure**
Read-only	**full**	**immutable**

State Information

Example: share(s) **in** open



What do we know about the object's state?

State information changes with every operation

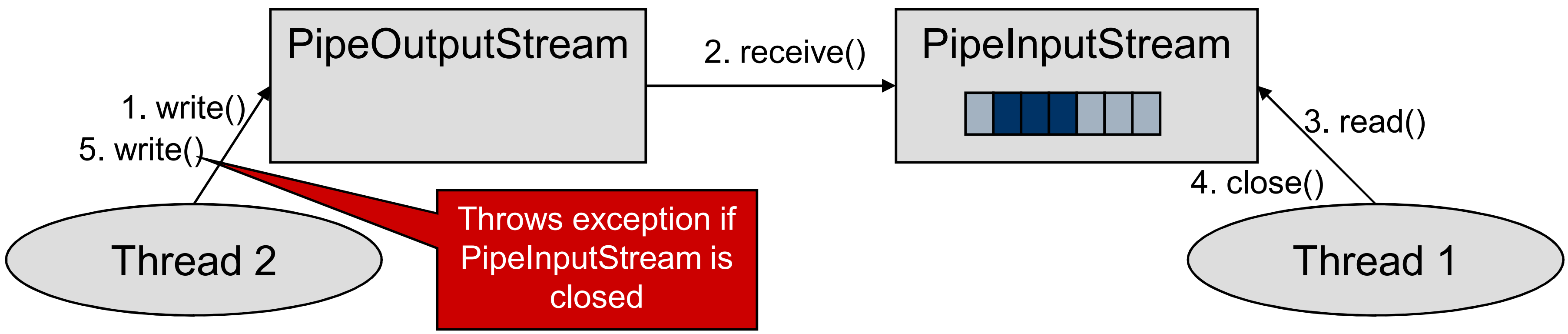
Outline

Previous work provides ***static, modular*** checking of both ***clients*** and ***implementations***

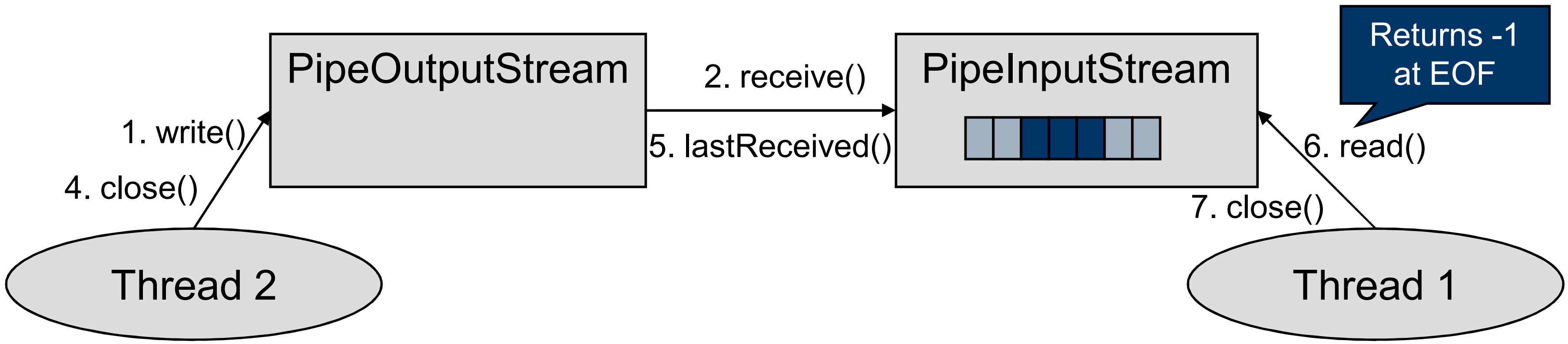
Our contributions

- New modular approaches to tracking ***aliased*** state
- ***Nondeterministic*** state changes
- ***Concurrency***
- Dynamic ***state tests***
- States with ***representation*** and ***behavior***

Pipes in Java



Pipes in Java



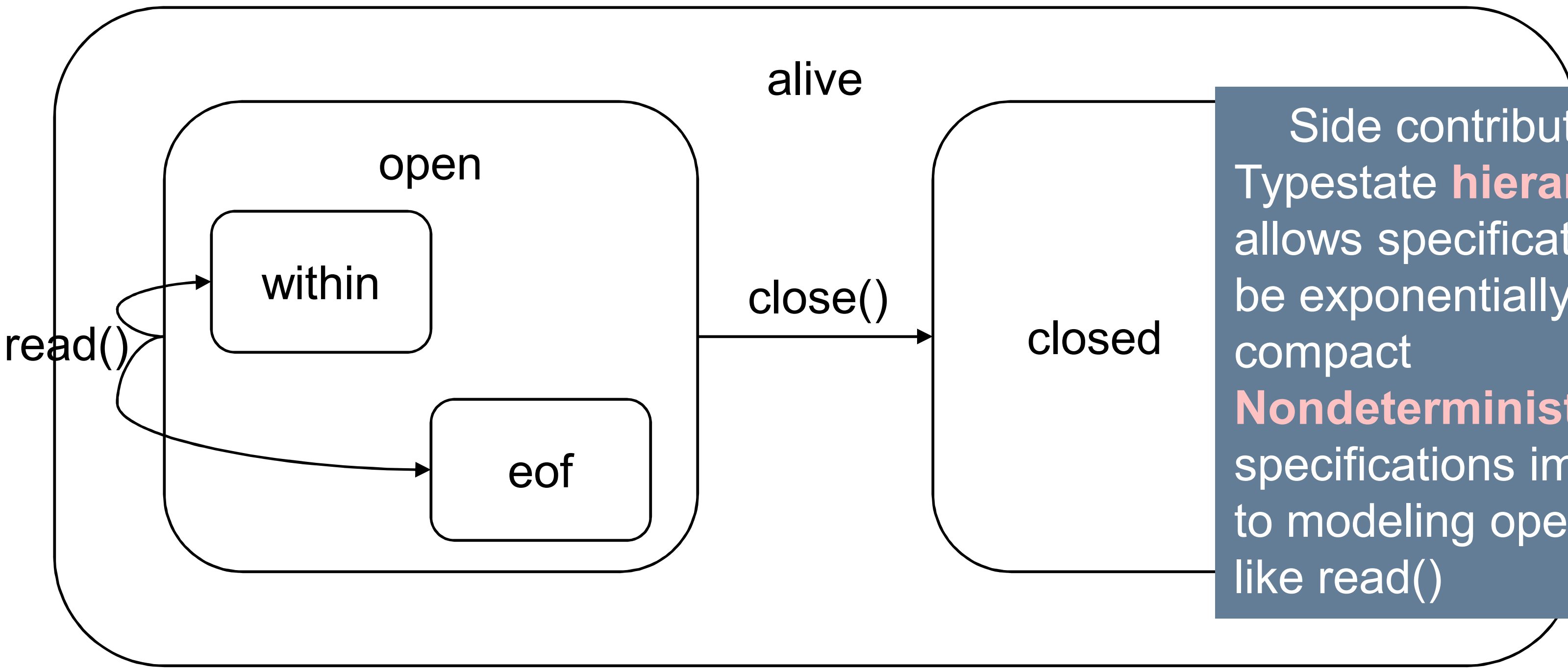
- Key intuition
  - Thread 1 and Thread 2 share the pipe
  - Thread 1 can't close until Thread 2 gives Thread 1 the permission to do so
  - This occurs through close() -> lastReceived() -> read() returning -1
- Challenge
  - Buffer is shared between threads
  - Alias analysis is typically either too imprecise or unscalable to track shared state
  - Need local reasoning – verify that if streams are used correctly, exception will never be thrown

Invariant-Carrying Permissions

- Each permission carries an invariant
  - Invariant == guaranteed state
  - Set up on **unique** reference, cannot be changed once reference is aliased
  - Defaults to alive (the universal state)

- Like assume-guarantee reasoning
  - All aliases can assume the state
  - All aliases must guarantee they don't leave the state
  - But we use it to deal with aliasing, not just concurrency

Typestate Hierarchy

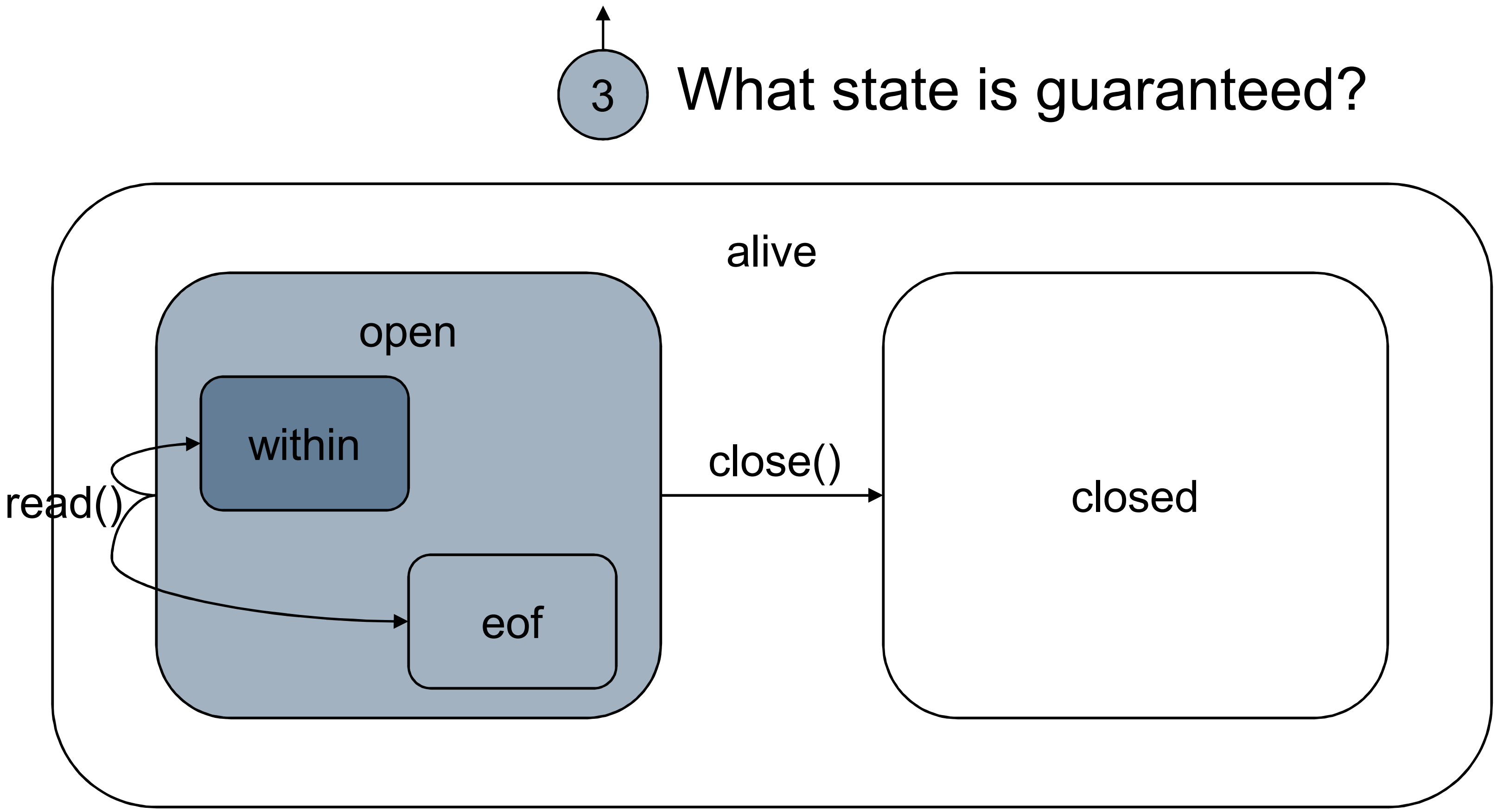


Side contributions:
Typestate **hierarchy** allows specifications to be exponentially more compact
Nondeterministic specifications important to modeling operations like read()

Subtypes can further refine existing states

State Guarantees

Example: share(s, open) **in** within

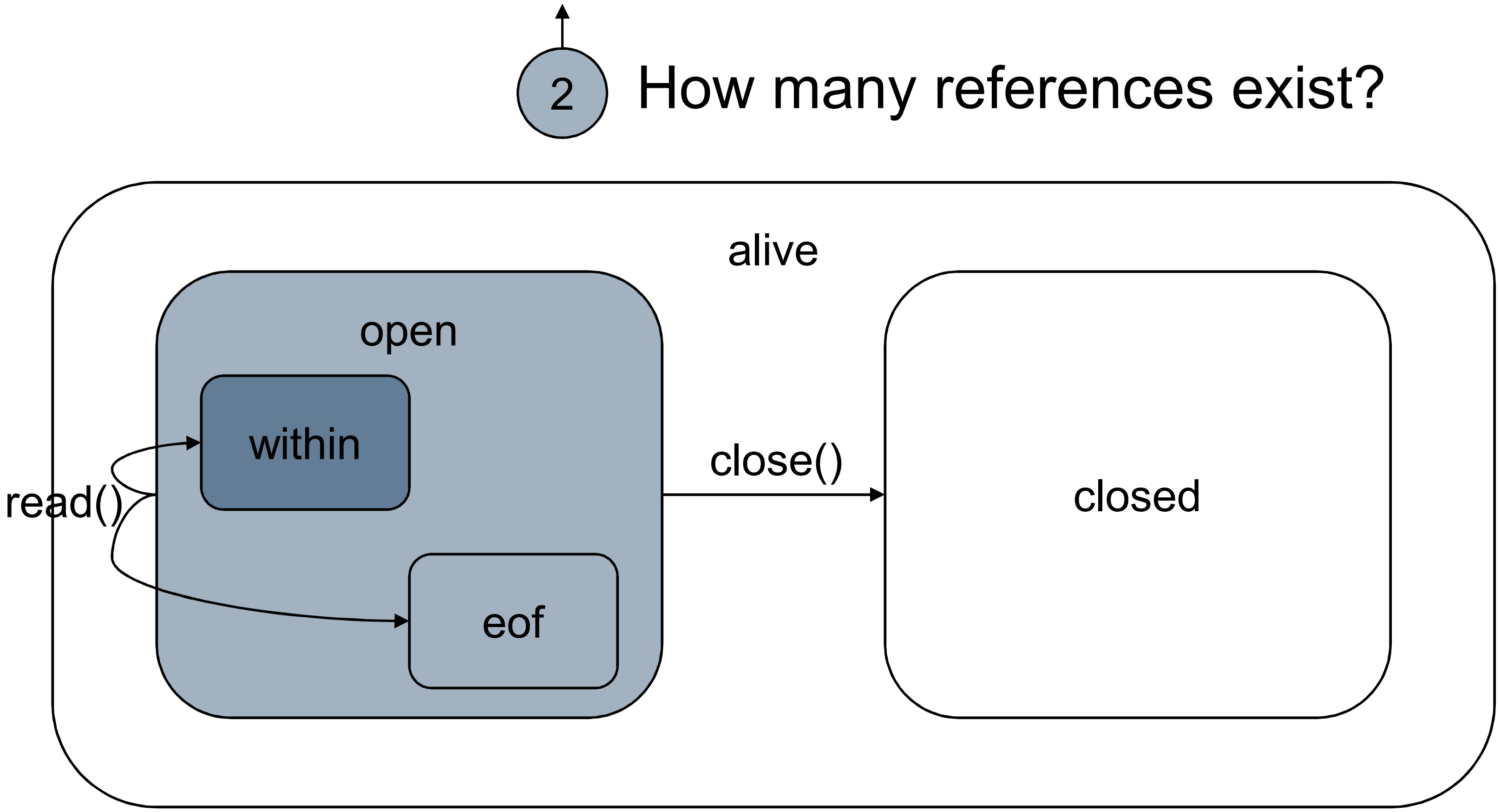


Temporary Invariants

- We want to eventually break the state guarantee and close the pipe
- Solution: fractional permissions [Boyland '03]
  - **unique** reference ⇔ whole (1.0) fraction
  - Splitting operation divides permission
    - ½ to each thread
    - Set up state guarantee: pipes remain open
  - Recombination adds fractions
    - When we restore a whole fraction, we can break the state guarantee

Fractional Permissions

Example: share(s, ½, open) **in** within



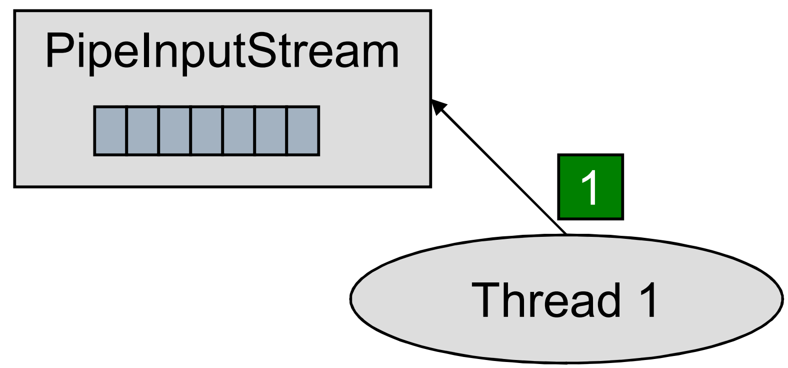
Access Permissions = state + aliasing

Example: share(s, ½, open) **in** within

1 2 3 4

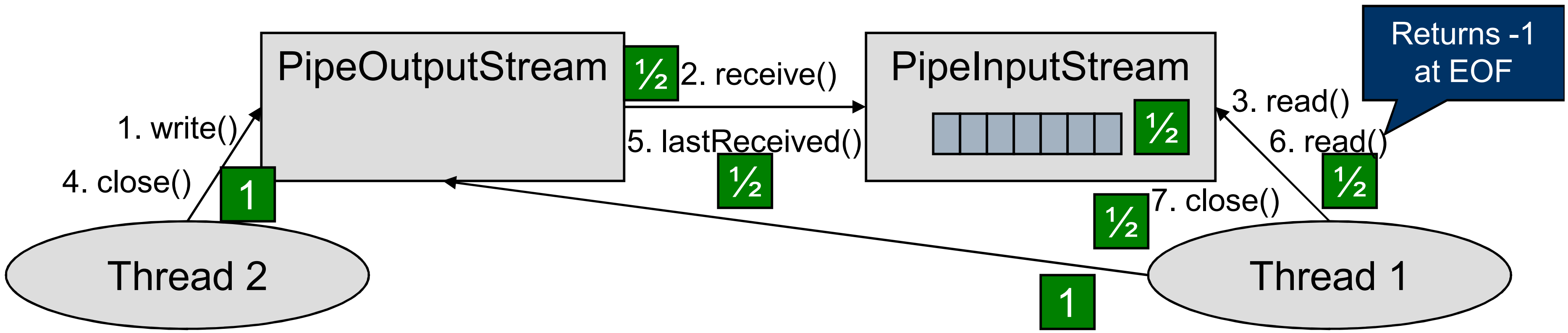
1. What kinds of references exist?
2. How many references exist?
3. What state is guaranteed?
4. What do we know about the object's state at a given point?

Pipes in Java



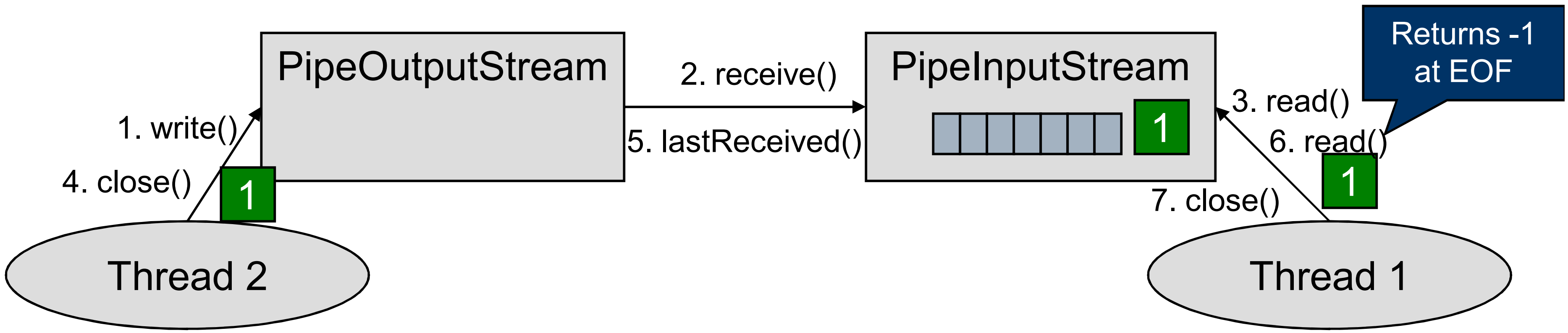
- Key intuition
  - Thread 1 and Thread 2 share the pipe
  - Thread 1 can't close until Thread 2 gives Thread 1 the permission to do so
  - This occurs through close() -> lastReceived() -> read() returning -1

Pipes in Java



- Key intuition
  - Thread 1 and Thread 2 share the pipe
  - Thread 1 can't close until Thread 2 gives Thread 1 the permission to do so
  - This occurs through close() -> lastReceived() -> read() returning -1
- Split permission in half
  - State guarantee: pipes remain open

Pipes in Java



- Key intuition
  - Thread 1 and Thread 2 share the pipe
  - Thread 1 can't close until Thread 2 gives Thread 1 the permission to do so
  - This occurs through close() -> lastReceived() -> read() returning -1
- Split permission in half
  - State guarantee: pipes remain open
- Contribution: coordinating two clients that mutate state
  - In Boyland's system a fraction grants read-only access
  - Here, Thread 1 and Thread 2 change PipeInputStream's state, but they can't close it until their permissions are combined

Invariant-Carrying Permissions

- The state guarantee is really an invariant
  - Carried along in the permissions
  - Customized to the particular object, and potentially temporary
    - Compare class invariants, true for all objects at all times
- Key insight
  - Nowhere need we track exact heap structure
  - Each client can assume the invariant of the object
  - Each client must ensure the invariant is preserved
- Compared to previous approaches
  - Logical approaches: must track heap structure of each reader/writer pair
    - May be difficult if we have many pipes
    - Prohibits separate verification, composition
  - Ownership: does not help as state is not owned
    - Once again, must specify shape of heap
  - Previous permission-based approaches: cannot express
    - Require a unique writer

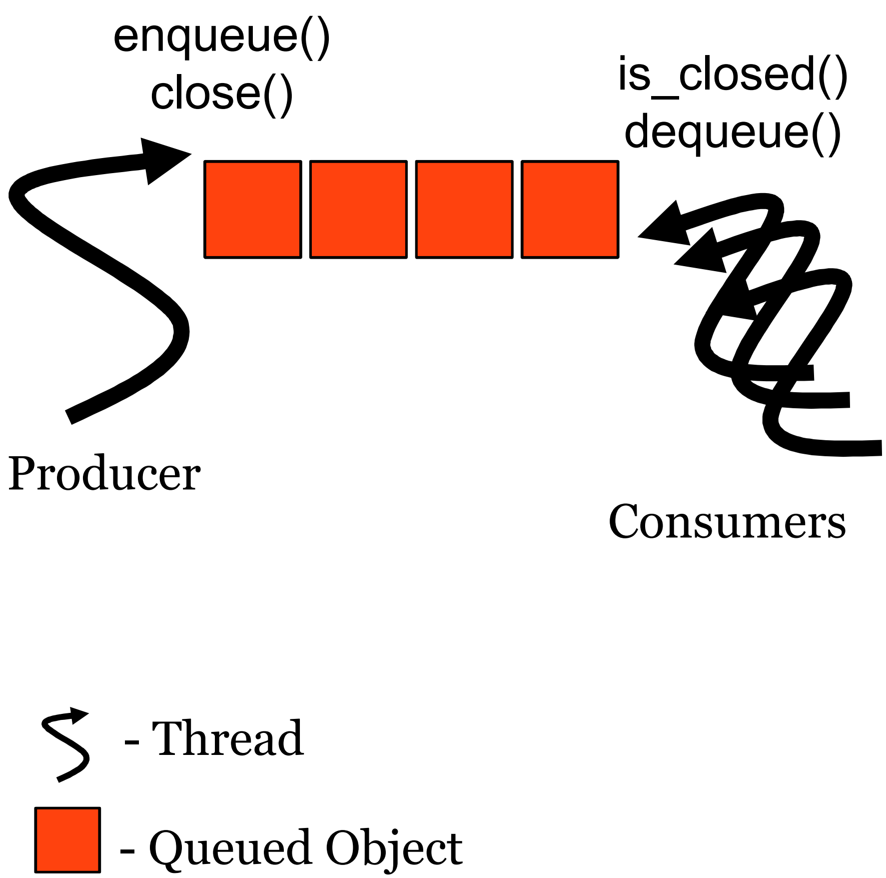
Outline

Previous work provides ***static, modular*** checking of both ***clients*** and ***implementations***

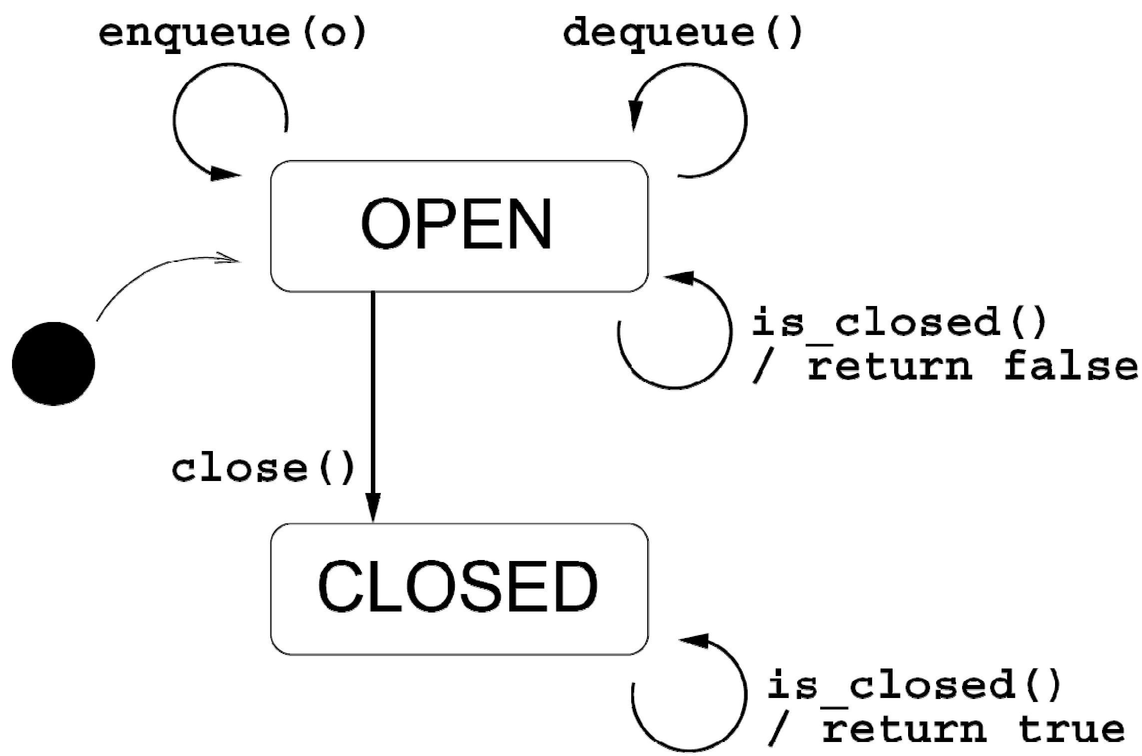
Our contributions

- New modular approaches to tracking ***aliased*** state
- ***Nondeterministic*** state changes
- ***Concurrency***
- Dynamic ***state tests***
- States with ***representation*** and ***behavior***

Queue: Runtime View & Protocol



Queue: Protocol



S - state

- initial state

`foo()` - state transition

`/ do_something` - transition action

`is_closed` is an example of a **dynamic state test**. The return value can be tested to gain knowledge of the Queue's state.

Race Condition in Consumer

```
final Blocking_queue queue = new Blocking_queue();

(new Thread() {
  @Override
  public void run() {
    while( !queue.is_closed() )
      System.out.println("Got object: "+queue.dequeue());
    // Important shut-down code…
  }}).start();

for( int i=0;i<5;i++ )
  queue.enqueue("Object " + i);

queue.close();
```

Where is the race condition?

Race Condition in Consumer

```java
final Blocking_queue queue = new Blocking_queue();

(new Thread() {
  @Override
  public void run() {
    while( !queue.is_closed() )
      System.out.println("Got object: "+queue.dequeue());
    // Important shut-down code…
  }}).start();

for( int i=0;i<5;i++ )
  queue.enqueue("Object " + i);

queue.close();
```

Race!

Potential Race in Producer

```
final Blocking_queue queue = new Blocking_queue();

(...).start();  // queue escapes to thread

for( int i=0;i<5;i++ )
  queue.enqueue("Object " + i);

queue.close();
```

Potential Race in Producer

```
final Blocking_queue queue = new Blocking_queue();

(...).start();  // queue escapes to thread

for( int i=0;i<5;i++ )
  queue.enqueue("Object " + i);

queue.close();
```

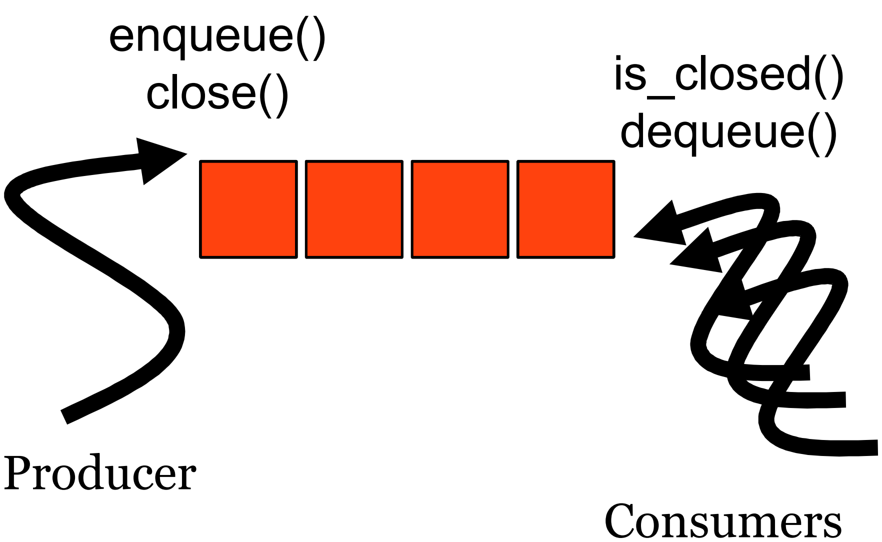
Queue must be OPEN!

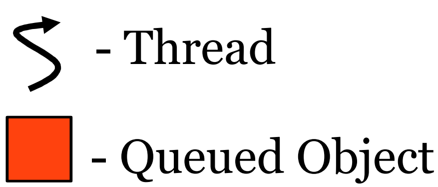
Potential Race in Producer

```
final Blocking_queue queue = new Blocking_queue();

(...).start();  // queue escapes to thread

for( int i=0;i<5;i+
  queue.enqueue("Obj

queue.close();
```

We must somehow encode:
The producer is ‘in control’
of the protocol!

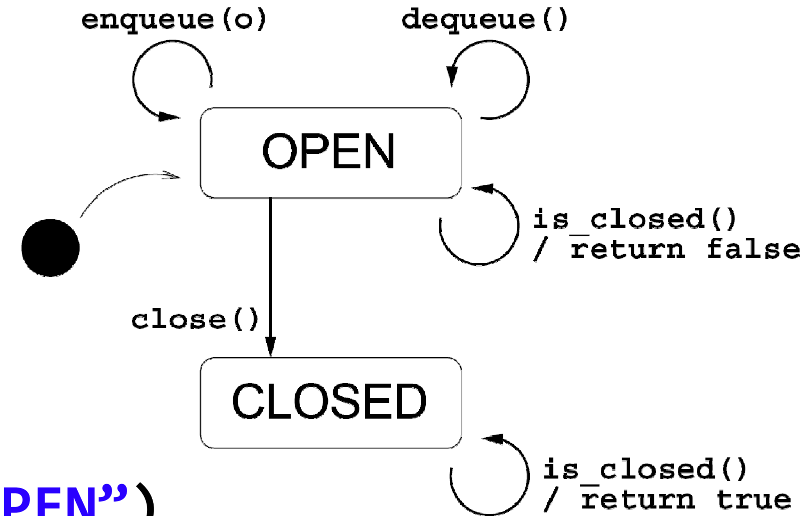
Queue: Runtime View & Protocol





Queue Method Signatures

```
@Full(requires="OPEN", ensures="OPEN")
void enqueue(@Share Object o)

@Full(requires="OPEN", ensures="CLOSED")
void close()

@Pure
@TrueIndicates("CLOSED")
@FalseIndicates("OPEN")
boolean is_closed()

@Pure(requires="OPEN",ensures="OPEN")
Object dequeue()
```



Client-Side Verification: No Races on Abstract State

- Track permissions and state of references through method body
- At method call sites, use pre/post-conditions
- Discard object state if permission indicates concurrent modification
  - @Pure or @Share
- Unless inside atomic block!

Verification with Permissions

```
final Blocking_queue queue = new Blocking_queue();

(...).start();

for( int i=0;i<5;i++ )
  queue.enqueue("Object " + i);

queue.close();
```

Verification with Permissions

```
final Blocking_queue queue = new Blocking_queue();

(...).start();

for( int i=0;i<5;i++ )
  queue.enqueue("Object " + i);

queue.close();
```

@Unique(queue)
in OPEN

Verification with Permissions

```
final Blocking_queue queue = new Blocking_queue();

(...).start();

for( int i=0;i<5;i++ )
  queue.enqueue("Object " + i);

queue.close();
```

@Full(queue) in
OPEN

Verification with Permissions

```
final Blocking_queue queue = new Blocking_queue();

(...).start();

for( int i=0;i<5;i++ )
  queue.enqueue("Object " + i);

queue.close();
```

Method
precondition
met

Verification with Permissions

```
final Blocking_queue queue = new Blocking_queue();

(...).start();

for( int i=0;i<5;i++ )
  queue.enqueue("Object " + i);

queue.close();
```

Method precondition met

Consumer Verification

```
@Override
public void run() {
  while( !queue.is_closed() )
    System.out.println("Got object: “ +
                       queue.dequeue());
  // Important shut-down code…
}
```

Consumer Verification

```
@Override
public void run() {
  while( !queue.is_closed() )
    System.out.println("Got object: “ +
                                    queue.dequeue());
  // Important shut-down code…
}
```

@Pure(queue)

from class
invariant…

Consumer Verification

```
@Override
public void run() {
  while( !queue.is_closed() )
    System.out.println("Got object: “ +
                                   queue.dequeue());
  // Important shut-down code…
}
```

@Pure(queue)
in
OPEN

Consumer Verification

```
@Override
public void run() {
  while( !queue.is_closed() )
    System.out.println("Got object: “ +
                       queue.dequeue());
  // Important shut-down code…
}
```

@Pure(queue)
in
~~OPEN~~

Consumer Verification

```
@Override
public void run() {
  while( !queue.is_closed() )
    System.out.println("Got object: “ +
                                   queue.dequeue());
  // Important shut-down code…
}
```

ERROR!

But with ‘atomic’

```
@Override
public void run() {
  while( true ) {
    atomic: {
      if( !queue.is_closed() )
        System.out.println("Got object: “+queue.dequeue());
      else
        return;
    }
  }
  // Important shut-down code…
}
```

Because of atomic, no need to forget current state

@Pure(queue)
in
OPEN

State Transition Not Atomic

```
class Blocking_queue {
  // Class definition...
  public void close() {
    atomic: { elements = null; }
    // ...
    atomic: { closed = true; }
  }
}
```

State Transition Not Atomic

```
class Blocking_queue {
  // Class definition...
  public void close() {
    atomic: { elements = null; }
    // ...
    atomic: { closed = true; }
  }
}
```

Can be observed in inconsistent state!

Implementation-Side Verification: Transitions are Atomic

- States can be annotated with concrete invariants
  - Predicates over fields
- Use packing/unpacking for modular verification
  - Invariants must be reestablished before method returns

- Unpacking a **`@Full`**, **`@Pure`**, or **`@Share`** object must be within an atomic block

Verification Example

```
@ClassStates({
  @State(name="CLOSED",
    inv="closed == true * elements == null"), ...})
class Blocking_queue {
  private List elements;
  private boolean closed;
  // ...
  @Full(requires="OPEN",ensures="CLOSED")
  void close() {
    atomic: { elements = null; }
    // ...
    atomic: { closed = true; }
  }
  // ...
}
```

Verification Example

```
@ClassStates({
  @State(name="CLOSED",
    inv="closed == true * elements == null"), ...})
class Blocking_queue {
  private List elements;
  private boolean closed;
  // ...
  @Full(requires="OPEN",ensures="CLOSED")
  void close() {
    atomic: { elements = null; }
    // ...
    atomic: { closed = true; }
  }
  // ...
}
```

Verification Example

```
@ClassStates({
  @State(name="CLOSED",
    inv="closed == true * elements == null"), ...})
class Blocking_queue {
  private List elements;
  private boolean closed;
  // ...
  @Full(requires="OPEN",ensures="CLOSED")
  void close() {
    atomic: { elements = null; }
    // ...
    atomic: { closed = true; }
  }
  // ...
}
```

Unpacks from OPEN state.

Verification Example

```
@ClassStates({
  @State(name="CLOSED",
    inv="closed == true * elements == null"), ...})
class Blocking_queue {
  private List elements;
  private boolean closed;
  // ...
  @Full(requires="OPEN",ensures="CLOSED")
  void close() {
    atomic: { elements = null; }
    // ...
    atomic: { closed = true; }
  }
  // ...
}
```

Packs to CLOSED state.

Verification Example

```
@ClassStates({
  @State(name="CLOSED",
    inv="closed == true * elements == null"), ...})
class Blocking_queue {
  private List elements;
  private boolean closed;
  // ...
  @Full(requires="OPEN",ensures="CLOSED")
  void close() {
    atomic: { elements = null; }
    //
    atomic: { closed = true; }
  }
  // ...
}
```

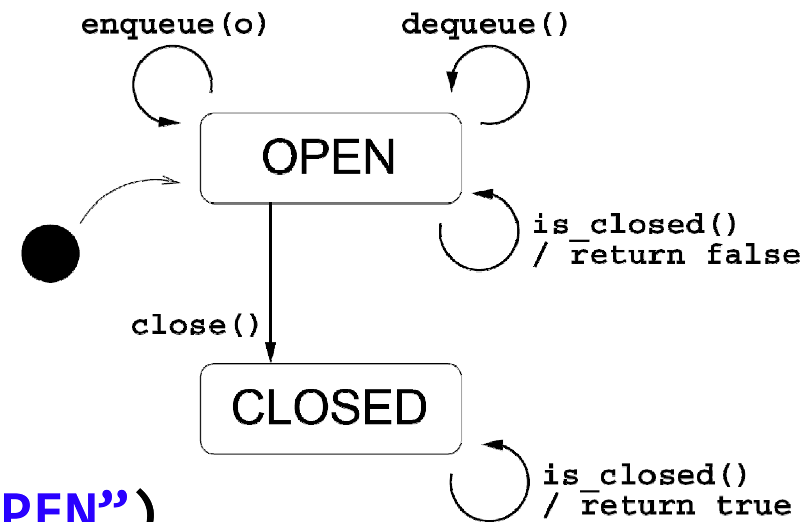
Error! Atomic block ends while receiver unpacked

Something is fishy!

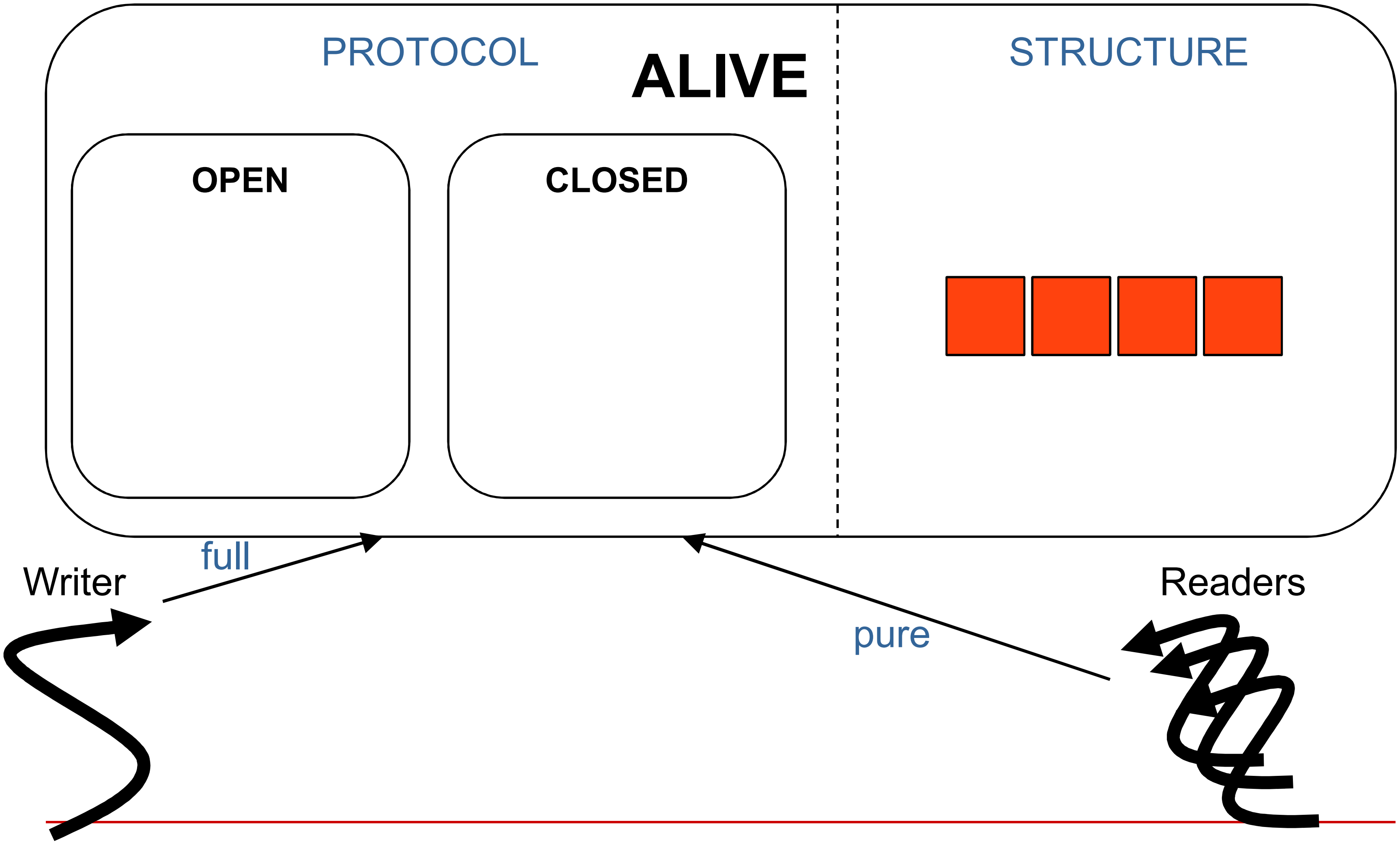
```
@Full(requires="OPEN", ensures="OPEN")
void enqueue(@Share Object o)

@Full(requires="OPEN", ensures="CLOSED")
void close(

@Pure
@TrueIndic
@FalseIn
boolean is_closed()

@Pure(requires="OPEN",ensures="OPEN")
Object dequeue()
```

@Pure means we can't change the Queue.

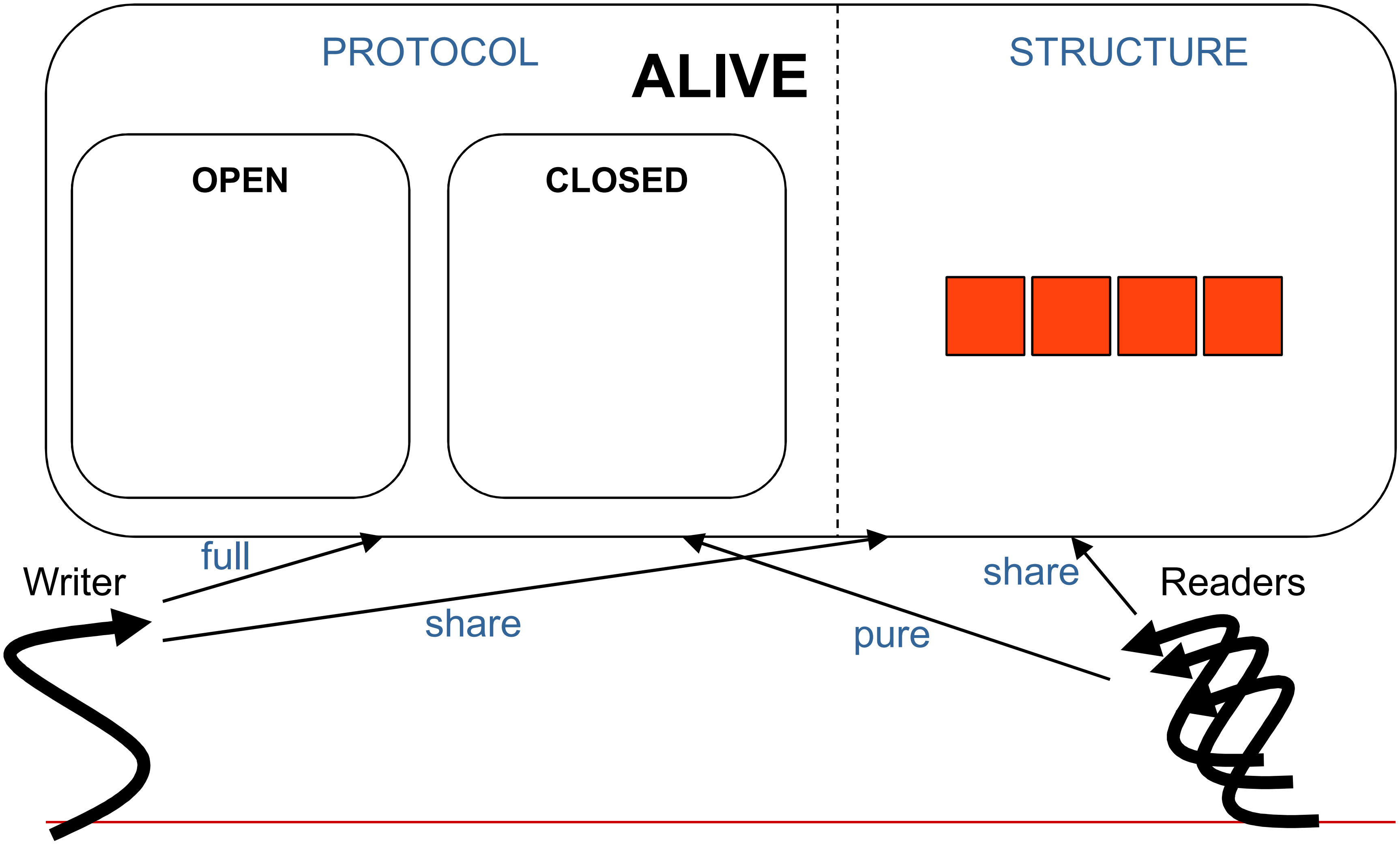
But dequeue must affect the buffer!



State Dimensions and Fields



State Dimensions and Fields



Queue: The Full Specification

```
@Refine({
  @States(dim="STRUCTURE", value={"STRUCTURESTATE"}),
  @States(dim="PROTOCOL", value= {"CLOSED", "OPEN"})
})
@ClassStates({
  @State(name="STRUCTURE",
   inv="share(elements) * reject_enqueue_requests ==
    true => full(this,PROTOCOL) in OPEN"),
  @State(name="OPEN", inv="closed == false"),
  @State(name="CLOSED", inv="closed == true")
})
public class Blocking_queue
{
```

STRUCTURE dimension holds array.

PROTOCOL dimension determines if Queue is open.

Queue: The Full Specification

```
@Share(value="STRUCTURE")
@Full(requires="OPEN", ensures="OPEN",
    value="PROTOCOL")
void enqueue( Object new_element )

@Perm(requires="full(this,PROTOCOL) in OPEN *
    share(this,STRUCTURE)")
void enqueue_final_item(Object elm)

@Perm(requires="share(this!fr,STRUCTURE) *
      pure(this!fr,PROTOCOL) in OPEN",
  ensures="share(this!fr,STRUCTURE) *
           pure(this!fr,PROTOCOL)")
public Object dequeue( )
```

dequeue takes a pure PROTOCOL permission but a share STRUCTURE permission

Queue: The Full Specification

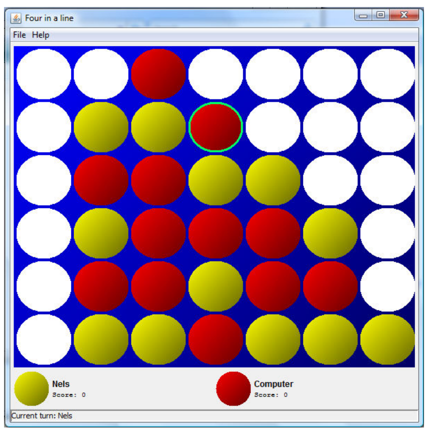
```
@Pure(fieldAccess=true,value="PROTOCOL")
@TrueIndicates("CLOSED")
@FalseIndicates("OPEN")
boolean is_closed()

@Full(fieldAccess=true,value="PROTOCOL",
      requires="OPEN",ensures="CLOSED")
void close()
```

TOOL DEMONSTRATION

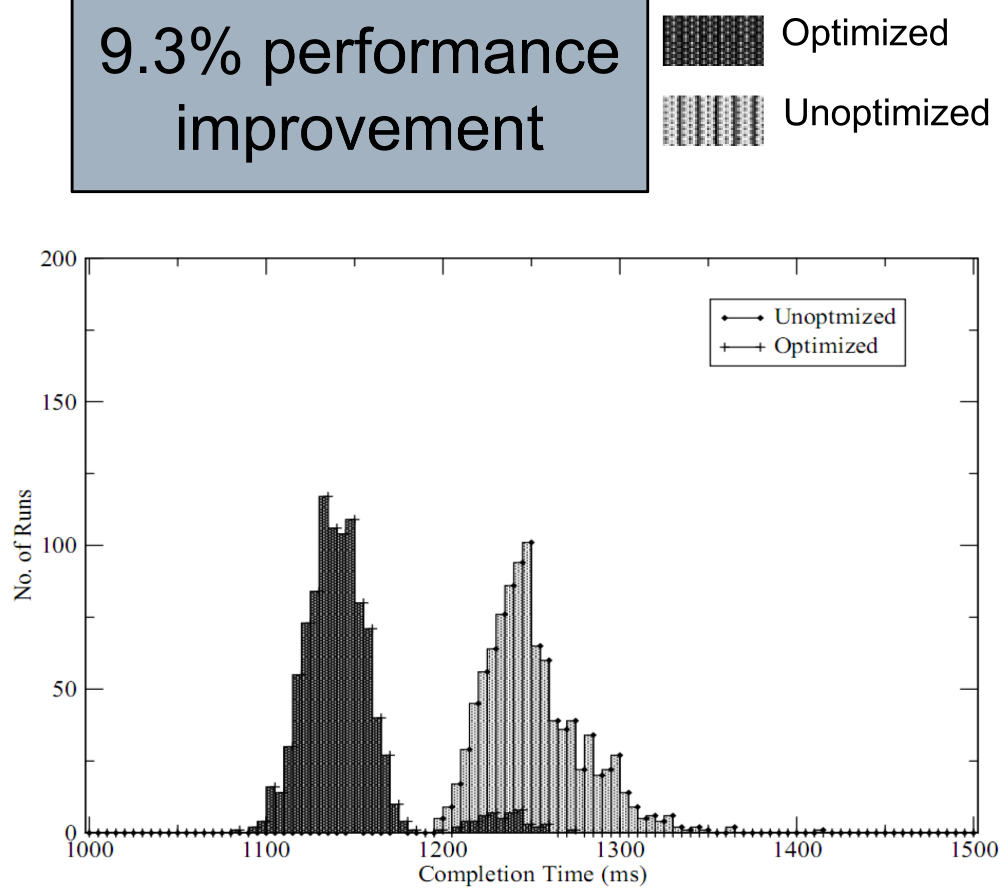
- Queue

Permissions can Help Optimize STM!

- Idea: can avoid synchronization overhead on **unique** and **immutable** objects
  - Also some savings on **full**
- 4InALine benchmark
  - Numbers from an 2x quad-core Intel Xeon machine



9.3% performance improvement



Contributions: Atomicity and Typestate

- First approach to verifying correct use of *atomic* block
  - Ensures typestate properties hold in a concurrent system
  - Ensures freedom from semantic races, not just syntactic races
    - Up to semantics that can be encoded as typestate
  - Demonstrates properties of *atomic*
    - Simplicity – no need to track which lock protects which state
    - Compositionality – can verify typestate in non-hierarchical data structures
      - No current automated system can do this with locks

- Implemented, proven sound
  - Client and implementation-side typestate verification

- Permissions aid optimization
  - Substantial reduction in STM overhead

Outline

Previous work provides ***static, modular*** checking of both ***clients*** and ***implementations***

Our contributions

- New modular approaches to tracking ***aliased*** state
- ***Nondeterministic*** state changes
- ***Concurrency***
- Dynamic ***state tests***
- ***Experience*** with Plural
- States with ***representation*** and ***behavior***

APIs can be annotated quickly

- Annotated 4 Java standard APIs
  - Java Database Connectivity (JDBC)
  - Collections (Lists, Sets, Maps, Iterators)
  - Regular Expressions
  - Exceptions

Example: Java Database Connectivity (JDBC)

5 main interfaces took us about a week to annotate

Interfaces	Total lines	Increase	Methods	Annotations
5	9,866	10.4%	440	838

Mostly informal documentation

Recurring API patterns (could be captured by Plural)

	Dynamic State Tests	Dependent Objects	Method Cases
Description	Methods return value indicates object state	Many objects depend on state of another object	Method behavior different depending on object state
JDBC ResultSet example	next(), isClosed()	Result sets depend on statement to remain open	setter methods (82/187 total)
Found in APIs (studied 4)	JDBC, Collections, Regex	JDBC, Collections, Regex	JDBC, Collections
Support in existing work	Support rare (e.g. Vault, Size props.)	Some support in global analyses	Only supported in JML / Spec#

Practical protocol verification approaches should support these patterns

Case studies illustrate viability of verification approach

	Apache Beehive	PMD
Size	Small: ~2,000 total lines in 12 classes	Large: 38.5 KLOC in 446 classes
Protocols checked	Deep: 4 specified APIs incl. JDBC	Simple: Correct iterator usage
Annotations	66 (~1 per method)	15
Tool runtime (3.2GHz, 2GB RAM)	188 ms / method	62 ms / method
Warnings (false)	9 (5)	3 (3)

Plural can analyze one method at a time

3 warnings from impure call in typically pure Iterator method,
1 from field access in wrong method

Unspecified but correct iterator usage

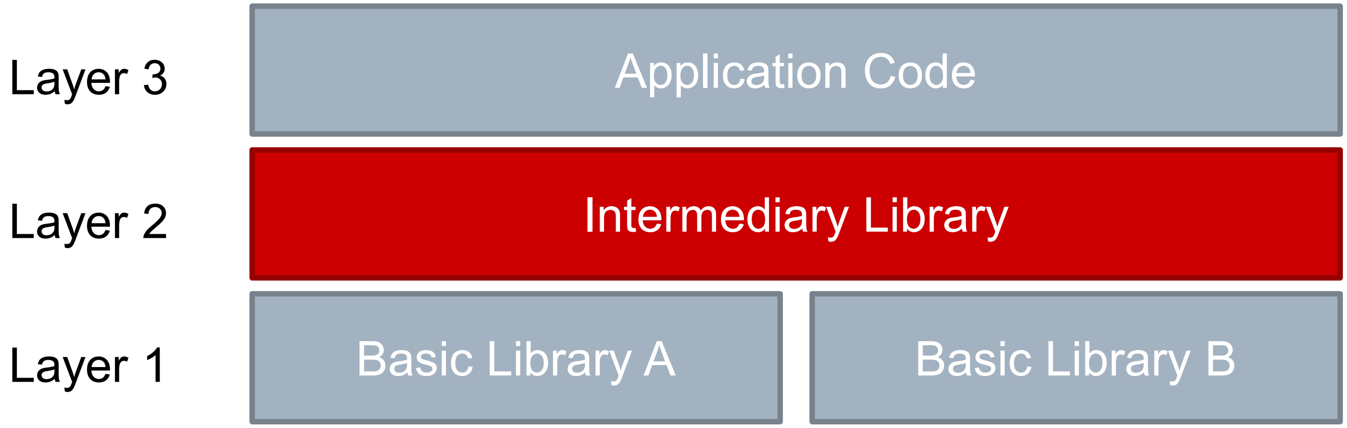
Tool usage observations

- Incremental benefit
  - APIs can be annotated independently
  - Simple protocols are simple to check
- Iterative annotation process
  - Annotate methods that call APIs, then their callers
  - Annotate methods interfaces for clients
    - Later check the method implementation
- Implement one protocol with another
  - Example: Beehive iterator over result set

Plural's modular checking of individual methods gives us these for free

Modularity allows analyzing large systems

- Modularity = analyze part of a program independently from the rest
  - Allows compositional reasoning
  - Essential for creating reusable components
  - Allows analyzing individual classes interactively
  - Ensures scalability to large programs



Study Conclusions

- Empirical evaluation of Plural
  - JDBC largest protocol specification case study we know
  - Annotation overhead on the level of types
  - 1 false positive per 400 lines (Beehive) or less
- 3 challenging recurring patterns
  - Dynamic state tests, dependent objects, method cases
- Iterator's hasNext has effects in practice
- In our ECOOP'09 paper
  - Many interesting details of JDBC specification
  - Details on annotations and imprecision sources

Outline

Previous work provides ***static, modular*** checking of both ***clients*** and ***implementations***

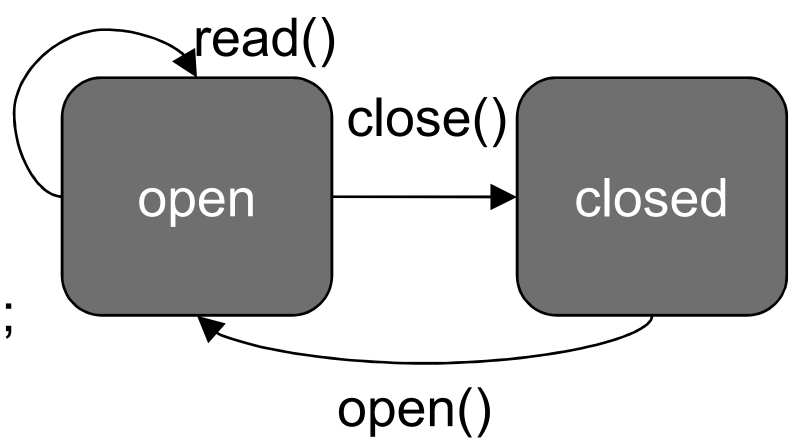
Our contributions

- New modular approaches to tracking ***aliased*** state
- ***Nondeterministic*** state changes
- ***Concurrency***
- Dynamic ***state tests***
- ***Experience*** with Plural
- States with ***representation*** and ***behavior***

Typestate-Oriented Programming

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

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

State transition



Different representation

New methods

Typestate-Oriented Programming

- Definition: A **programming paradigm** in which:

  programs are made up of dynamically created **objects**,
    - Compare: embedded system CASE tools

  each object has a **typestate** that is **changeable** and **statically trackable**,
    - Compare: plain OO classes
    - Compare: dynamically typed state proposals (actors, roles, modes, …) or the State design pattern

  and each typestate has an **interface**, **representation**, and **behavior**.
    - Compare: typestate analysis on top of OO

- In our model interface, representation, and behavior change with an object's typestate, but object identity does not
  - Related: class change proposals (e.g. Fickle)

Why Put Typestate in the Language?

- Language influences thought [Boroditsky ’09]
  - Language support encourages engineers to **think** about states
    - Better designs, better documentation, more effective reuse
- Improved library specification and verification
  - Typestates define when you can call read()
  - Make constraints that are only implicit today, explicit
- Expressive modeling
  - If a field is not needed, it does not exist
  - Methods can be overridden for each state
- Simpler reasoning
  - Without state: fileResource non-**null** if File is open, **null** if closed
  - With state: fileResource always non-**null**
    - But only exists in the FileOpen state

Implementing Typestate Changes

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

Typestate change primitive

Values must be specified for each new field

:

Parametric Polymorphism

```
state Collection {
    type TElem;

    void add(TElem>>none e);

    TElem removeAny();



}
```

Type parameter must now include state and permission

Adding an element to the collection removes the client's permission to it (e.g. to ensure unique objects are unaliased)

If we want to get an element, we must remove it from the collection (to avoid aliasing).

Current Work: Typestate-Oriented Programming

PLAID is a new typestate-oriented programming language

Features:

- Java-like syntax, as presented in this talk
- Permissions describe aliasing on all objects
- Concurrency-by-default execution model
  - See "Concurrency By Default" Onward! '09 companion paper
- Gradual types
- Advanced modularity constructs (e.g. abstract types)
- Composition mechanism similar to traits (replaces inheritance)

Conclusions

Typestate increasing in importance

- Libraries, frameworks dominate modern software

Our work addresses pragmatic challenges

- New approaches to verifying typestate of aliased objects
  - Read/write permission abstractions w/state guarantees
- Concurrent state
  - Assure freedom from semantic races
- Nondeterminism
  - Dynamic checks to recover static information
- Modeling state representation and behavior
  - PLAID language supports first-class typestates

We have built practical tools and gathered experience

Try Plural at http://code.google.com/p/pluralism/