Take-Aways

- Protocols define temporal ordering of events
  - Can often be captured with state machines
- Protocol analysis needs to pay attention to
  - Interprocedural control flow
  - Aliasing of objects
- Disjoint sets and capabilities can handle aliasing correctly
Agenda

- Example protocols
  - Modeling protocols as state machines
- Protocol analysis approaches
  - Annotations vs. interprocedural analyses
- Aliasing challenges
  - Tracking aliases in methods and fields
- Protocol implementation checking
Streams can be read until they’re closed

public interface InputStream {
    public int read();
    public void close();
}

Stream sample client
InputStream f = new FileInputStream(…);
int c = f.read();  // read first character
while(c >= 0) {
    // do something with c
    c = f.read();  // read next character
}
f.close();

Stream protocol state machine
open
→ read()
→ closed

close()
Sockets go through a well-defined sequence of states

```java
@States({"created", "connected", "closed"})
public class Socket {
  @Creates("created")
  public Socket()

  @ChangesState("created", "connected")
  public void connect(…)

  @InState("connected")
  public InputStream getInputStream()

  @InState("connected")
  public OutputStream getOutputStream()

  @ChangesState("connected", "closed")
  public void close();
}
```

Java Socket protocol

![Java Socket protocol diagram](image)
Java Applets have a funny back edge

Java Applet protocol

created

init()

initialized

start()

running

stop()

stopped

start()

destroy()

destroyed

Crystal3 analyses have the same back edge.

Unawareness of this back edge can lead to outdated error reports.
Protocols constrain temporal ordering of events

- Protocols define restrictions on which methods can be called when
- Clients have to follow protocols in order to avoid runtime errors
- Protocols can often be modeled as state machines
Protocol documentation...

- Protocols are informally documented
  - Example: java.io.InputStream
    - Detailed Javadoc for every method
  - Example: java.net.Socket
    - Exceptions describe when methods cannot be called

- Not always complete and precise
...formalized in various ways

<table>
<thead>
<tr>
<th>Formalization</th>
<th>Socket example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotations on classes and methods</td>
<td><code>@States(“created”, “connected”, “closed”)</code></td>
</tr>
<tr>
<td></td>
<td><code>public class Socket {</code></td>
</tr>
<tr>
<td></td>
<td><code>    @Creates(“created”)</code></td>
</tr>
<tr>
<td></td>
<td><code>    public Socket()</code></td>
</tr>
<tr>
<td></td>
<td><code>    @ChangesState(“created”, “connected”)</code></td>
</tr>
<tr>
<td></td>
<td><code>    public void connect(…)</code></td>
</tr>
<tr>
<td></td>
<td><code>)</code></td>
</tr>
<tr>
<td>Regular expressions</td>
<td>`connect (getInputStream</td>
</tr>
<tr>
<td>State machine defined in one place (similar to Metal)</td>
<td><code>created : connect(…) -&gt; connected</code></td>
</tr>
<tr>
<td></td>
<td><code>connected :</code></td>
</tr>
<tr>
<td></td>
<td><code>    getInputStream() -&gt; connected</code></td>
</tr>
<tr>
<td></td>
<td>`</td>
</tr>
</tbody>
</table>

We will use annotations on classes and methods
Agenda

- Example protocols
  - Modeling protocols as state machines

Protocol analysis approaches
  - Annotations vs. interprocedural analyses

- Aliasing challenges
  - Tracking aliases in methods and fields

- Protocol implementation checking
Protocol analysis tracks states of variables

Socket sock = new Socket();
sock.connect(new InetSocketAddress("www.cs.cmu.edu",80));
InputStream in = sock.getInputStream();
sock.close();

<table>
<thead>
<tr>
<th>Post-state</th>
<th>Created</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Connected</td>
</tr>
<tr>
<td></td>
<td>Connected</td>
</tr>
<tr>
<td></td>
<td>Closed</td>
</tr>
</tbody>
</table>

- What if `sock` is assigned to another variable?
- What if `sock` is assigned to a field?
- What if `sock` is passed to another method?
public class SocketClient {

private String readSocket(Socket s) {
    InputStream in = s.getInputStream();
    ... // read and return string
}

public String readRemoteData() {
    Socket sock = new Socket();
    sock.connect(new InetSocketAddress("www.cs.cmu.edu", 80));
    String result = readSocket(sock);
    sock.close();
    return result;
}  }

Need to handle inter-procedural control flow
Interprocedural analysis techniques

- Need to handle inter-procedural control flow
  - Every method call could potentially affect analysis results
  - Need to figure out what happens in called methods

- Some possible approaches
  - Default assumptions
  - Interprocedural CFG
  - More annotations
Defaults too inflexible for protocol analysis

- **Simple approach:** **default assumptions**
  - Assumption about method parameters and result
  - Check that call and return sites respect the default
  - Example: Maybe-null assumption in null analysis (HW6)
    - Assume that method parameters may be null
    - Check methods with that assumption
    - All call and return sites automatically maybe-null

- **No reasonable default for protocol analysis**
  - “Any” state too imprecise (lots of false positives)
  - Optimistic assumption (a particular state) might be wrong a lot of the times
Interprocedural CFG “inlines” method calls

Interprocedural CFG
- Pretend that called methods are part of current method
- Every method appears once
- Problem: scalability
- One big CFG for the entire program

BEGIN
sock = new Socket();
sock.connect(...);
readSocket(sock);
sock.close();
END

BEGIN
s.getInputStream();
...
END

Interprocedural CFG hard to use at scale
Assume and Check Annotations

- Annotations
  - Starting dataflow value for all parameters
  - Dataflow value for result

- Verification
  - Initial info: starting value for parameters
  - Verify result $\sqsubseteq$ annotation$_{result}$
    - Ending value for result obeys annotation
  - Verify arg $\sqsubseteq$ annotation$_{arg}$
    - Actual arguments obey annotations on formal parameter

```java
String readSocket(@InState("connected") Socket s) {
    InputStream in = s.getInputStream();
    ... }
```
Agenda

- Example protocols
  - Modeling protocols as state machines
- Protocol analysis approaches
  - Annotations vs. interprocedural analyses
- Aliasing challenges
  - Tracking aliases in methods and fields
- Protocol implementation checking
Looks familiar? Aliasing is a problem that you can easily have

<table>
<thead>
<tr>
<th>Code</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SimpleProtocolTest t1 = new SimpleProtocolTest();</code></td>
<td>a</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td><code>SimpleProtocolTest t2 = new SimpleProtocolTest();</code></td>
<td>a</td>
<td>a</td>
<td>--</td>
</tr>
<tr>
<td><code>SimpleProtocolTest t3 = t1;</code></td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td><code>t1.aToB();</code></td>
<td>b</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>// t1 alias t3 in b, t2 in a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>t1 = t2;</code></td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>// t3 in b, t1 alias t2 in a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>t1.aToB();</code></td>
<td>b</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td><code>t3.bToC();</code></td>
<td>b</td>
<td>a</td>
<td>ERR</td>
</tr>
<tr>
<td><code>t2.inB();</code></td>
<td>b</td>
<td>ERR</td>
<td></td>
</tr>
<tr>
<td>// t1 alias t2 in b, t3 in c</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Aliasing** = multiple names for the same thing

Spurious warnings
Track local aliases as disjoint sets (aka equivalence classes)

- Track aliased variables as disjoint sets
  - Lattice information
    - \( A = \{ S_1, ..., S_n \} \)
    - \( S_1, ..., S_n \) disjoint sets of variables
  - Copy instructions \( x = y \)
    - Get y’s aliases \( S \in A \) where \( y \in S \)
    - Add \( x \) to \( S \) (and remove it from any other set)
  - Object allocations \( x = \text{new } C(...) \)
    - Remove \( x \) from existing sets
    - \( A = A \cup \{ x \} \) (i.e., add new set with just \( x \))
    - (Need to also set initial state for \( x \))

- Track state for each disjoint set
  - Method calls \( x = y.m(...) \)
    - Get y’s aliases \( S = \{ y_1, ..., y_n \} \) where \( y \in S \)
    - Update \( S \)’s state according to \( m \)’s spec
Disjoint sets correctly handle local aliases in example

<table>
<thead>
<tr>
<th>aliasing</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleProtocolTest t1 = new SimpleProtocolTest();</td>
<td>{t1}</td>
<td>a</td>
<td>--</td>
</tr>
<tr>
<td>SimpleProtocolTest t2 = new SimpleProtocolTest();</td>
<td>{t1}, {t2}</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>SimpleProtocolTest t3 = t1;</td>
<td>{t1,t3}, {t2}</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>t1.aToB(); // t1 alias t3 in b, t2 in a</td>
<td>{t1,t3}, {t2}</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>t1 = t2; // t3 in b, t1 alias t2 in a</td>
<td>{t1,t2}, {t3}</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>t1.aToB();</td>
<td>{t1,t2}, {t3}</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>t3.bToC();</td>
<td>{t1,t2}, {t3}</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>t2.inB(); // t1 alias t2 in b, t3 in c</td>
<td>{t1,t2}, {t3}</td>
<td>b</td>
<td>b</td>
</tr>
</tbody>
</table>

States of aliased variables are updated correctly
Calling other methods can affect fields

```java
public class AliasingFun() {
    @InState("b") private SimpleProtocolTest t2;

    private void callField() {
        t2.inB();
    }

    public void aliasingFun() {
        SimpleProtocolTest t1 = new SimpleProtocolTest();
        t1.aToB();
        internal(t1); t1.bToC();
        callField(); ...
    }
}
```

Field annotation makes this call go through

```
field annotation makes this call go through
```

```
```

This call violates t2's annotation

```
t2 is actually in "c" when called
```

```
```

```
t2 aliases t and t1
```

Fields hold on to objects beyond duration of methods

```
Fields hold on to objects beyond duration of methods
```

Our approach so far does not issue any warnings

```
Our approach so far does not issue any warnings
```
Aliasing through fields different from local variables

- Aliasing in local variables affects current method only
  - We can handle that with disjoint sets
- Fields hold on to objects
  - Assignment to field in one method can affect other methods
  - Changing state of local variable can inadvertently change state of field
- Other situations with similar problems?
Capabilities track whether an object is accessible

- **Capabilities**: Access objects only if not stored in a field
- Exactly one capability for each object
  - Can call methods only if capability available
    - `x.m(…)` only valid if caller has capability for `x`
  - Capability created with **new**
  - Field assignments `x.f = y`
    - “Capture” capability for `y`
- Annotate methods with capabilities
  - `@Captured` if capability needed but not returned
  - `@Borrowed` if capability needed and returned
Capabilities correctly handle field assignments and method calls

```java
public class AliasingFun() {
    @InState("b") private SimpleProtocolTest t;

    private void callField() {
        t.inB();
    }

    public void aliasingFun() {
        SimpleProtocolTest t1 = new SimpleProtocolTest();
        t1.aToB();
        internal(t1);
        t1.bToC();
        callField();
    }

    private void internal(@Captured SimpleProtocolTest t) {
        t2 = t;
    }

    private void internal(@Borrowed SimpleProtocolTest t) {
    }

    private void internal(@Captured SimpleProtocolTest t) {
    }

    Error: No capability for t1
```
Disjoint sets and capabilities can handle aliasing correctly

- Track disjoint sets of local aliases
  - Handle copies between local variables
- One capability for each object
  - Handle assignments to fields
- Capability annotations on methods
  - Handle aliasing during method calls

Capabilities are sometimes not enough

- Source calls receive(byte) to deposit characters
- ReceivedLast() signals no more characters

Pipe is modified through two independent aliases

- Reader calls read() to retrieve characters
- Reader calls close() to close the pipe
- Unsafe to call close() before source finished

read() returns -1
Permissions for shared access

- Permissions generalize capabilities
  - Permission required for all object access
  - Many permissions to the same object can exist
  - But keep track of how many permissions there are
- Unique(x) is the only existing permission for object referenced by x
  - Similar to capability for x
- Half(x) is one of two permissions for x
  - Half(x) + Half(x) = Unique(x)
Permissions in pipe example

- Source calls `receive(byte)` to deposit characters
- `ReceivedLast()` signals no more characters

- Reader calls `read()` to retrieve characters
- Reader calls `close()` to close the pipe
- Unsafe to call `close()` before source finished

Half(snk) → Half(s)

Unique(s)
Half + Half => Unique

Pipe

within  eof  closed

read() returns -1

Change to eof with Half permission
Unique permission needed to close the pipe
Agenda

- Example protocols
  - Modeling protocols as state machines
- Protocol analysis approaches
  - Annotations vs. interprocedural analyses
- Aliasing challenges
  - Tracking aliases in methods and fields

Protocol implementation checking
Implementation checking tracks changes to fields

- So far we looked at clients
  - Code calling methods on sockets etc.
  - Assumed that declared protocol was right
- Checking protocol implementations
  - Does *this* change state as declared?
  - **State changes = field manipulations**
    - Protocols ensure that “something” happened already (or has not happened yet)
    - “Something” can (only) be recorded in fields
State invariants define states in terms of fields

- **State invariants** constrain fields...
  - Constraints on field values
    - E.g., greater than zero or non-null
  - Expected state of referenced object
    - E.g., underlying stream should be “within” or “eof”
  - ...but only while in a particular state
    - close() will change fields accordingly

```java
public class BufferedInputStream {
    private InputStream in;
    private byte[] buffer;
    private int pos, count;
    // open: in instate (within | eof) &&
    //       buffer != null &&
    //       0 ≤ pos ≤ count &&
    //       count ≤ buffer.length
    // closed: in == null && buffer == null
```
Don’t forget aliasing…!

```java
public class BufferedInputStream {
    private InputStream in;
    private byte[] buffer;
    private int pos, count;

    // open: in instate (within | eof) &&
    //       buffer != null &&
    //       0 ≤ pos ≤ count &&
    //       count ≤ buffer.length

    // closed: in == null && buffer == null

    // closed: in == null && buffer == null
}
```

What happens when the underlying stream calls back to the buffer?

As it turns out, such a re-entrant callback can violate count’s invariant, leading to an access to buffer outside its bounds.