Verifying a higher-order, concurrent, stateful library

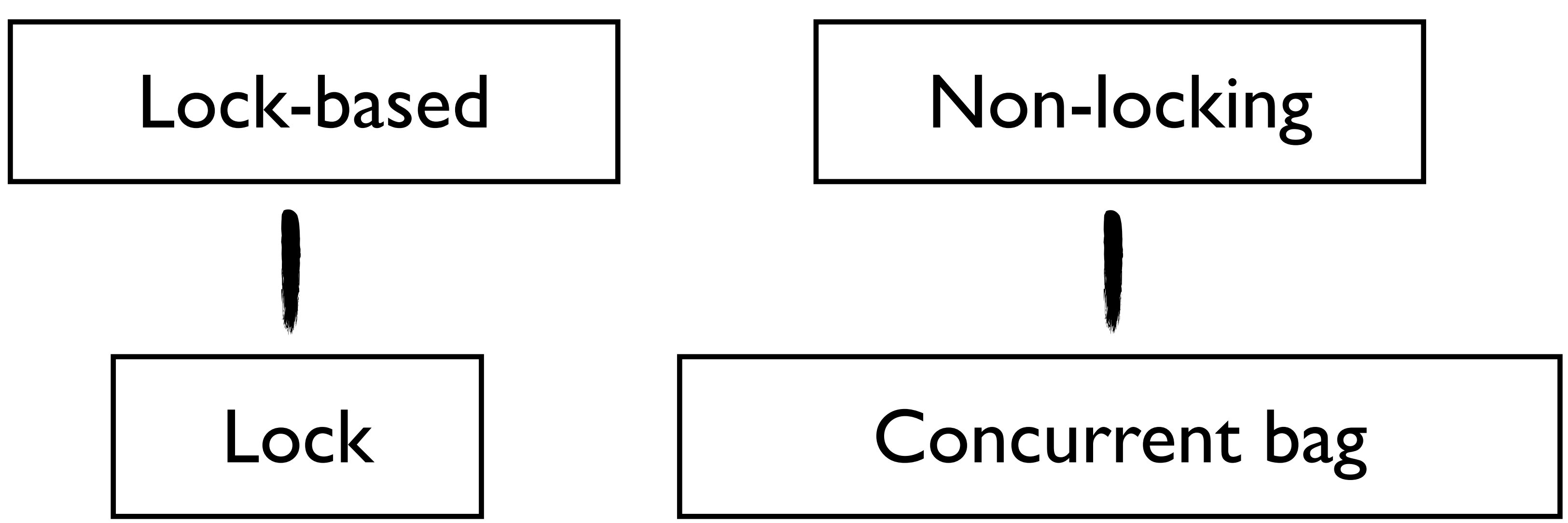
Kasper Svendsen, Lars Birkedal and Matthew Parkinson

September 9, 2012
HOPE 2012

A case study ...

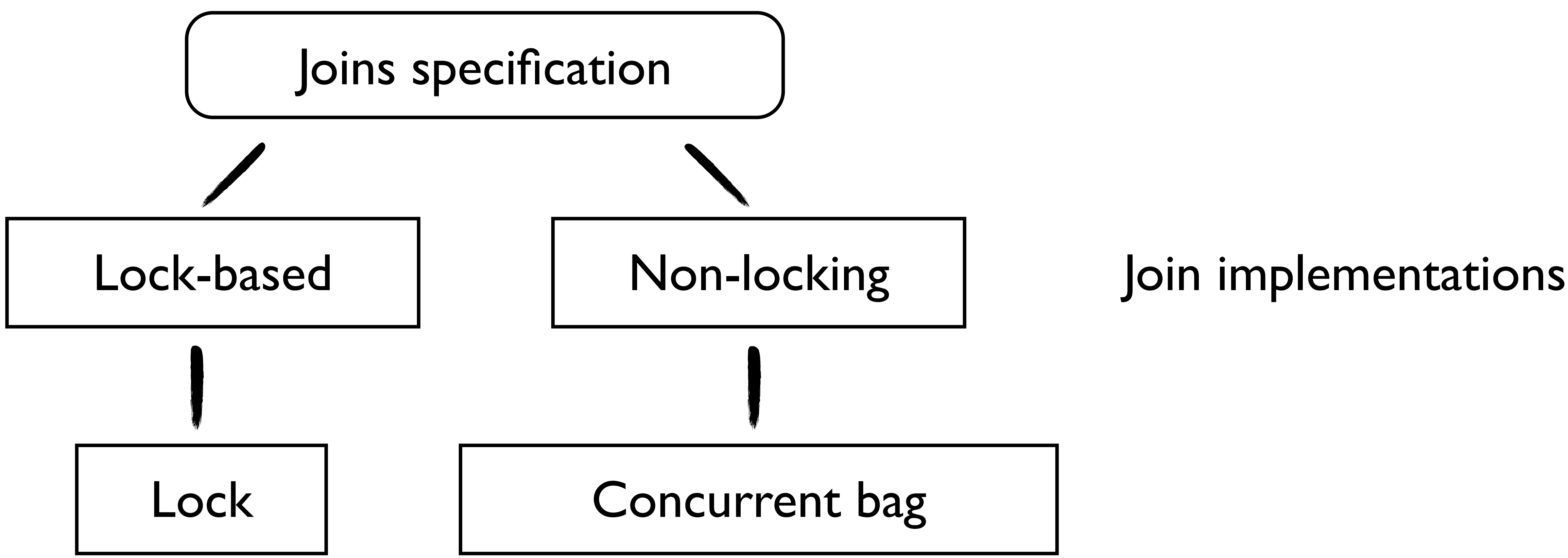
- C# Joins library [Russo, Turon & Russo]
  - declarative way of defining synchronization primitives, based on the join calculus [Fournet & Gonthier]
  - combines higher-order features with state, concurrency, recursion through the store and fine-grained synchronization
  - small (150 lines of C#) realistic library

A case study in modularity

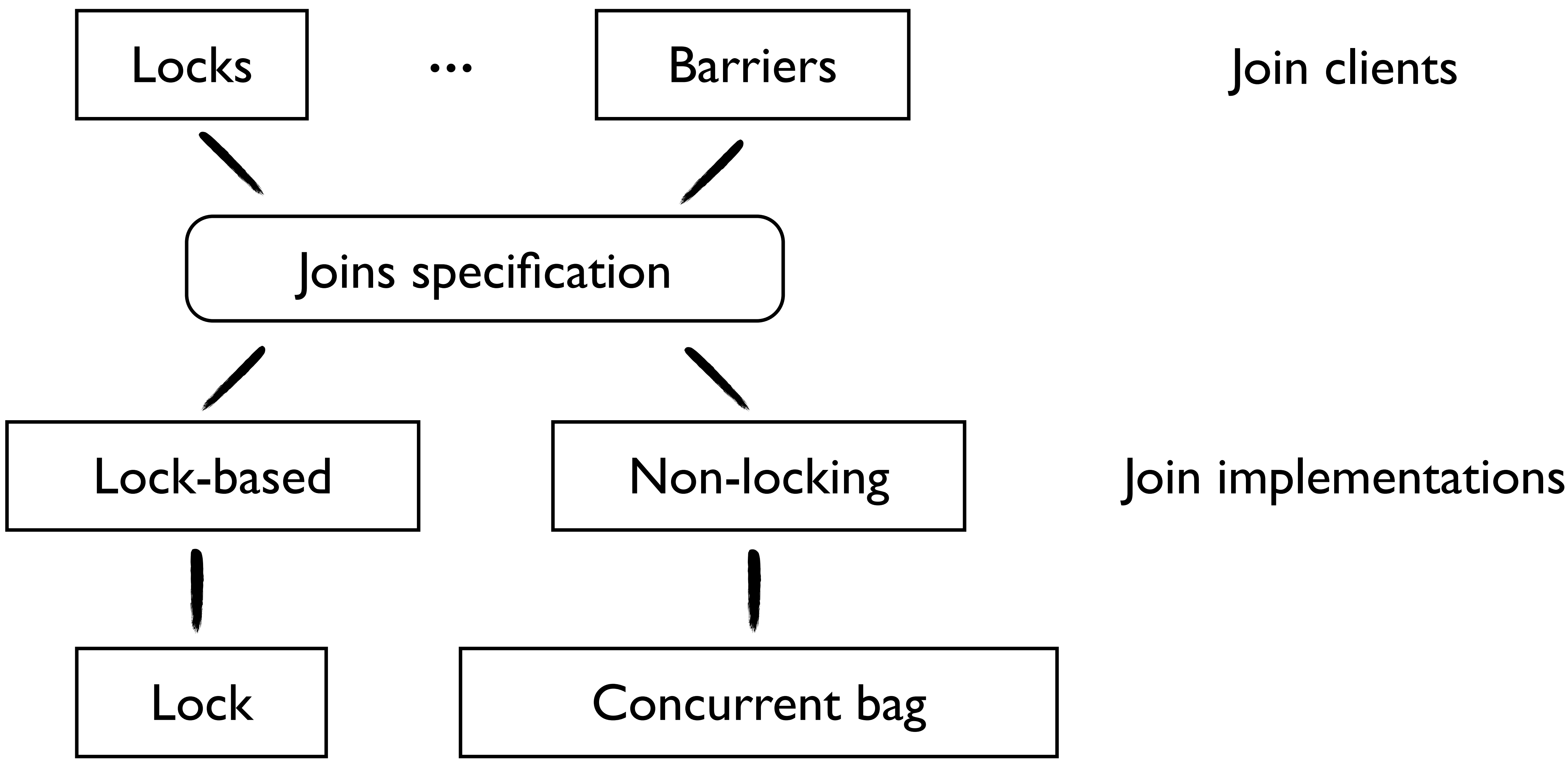


Join implementations

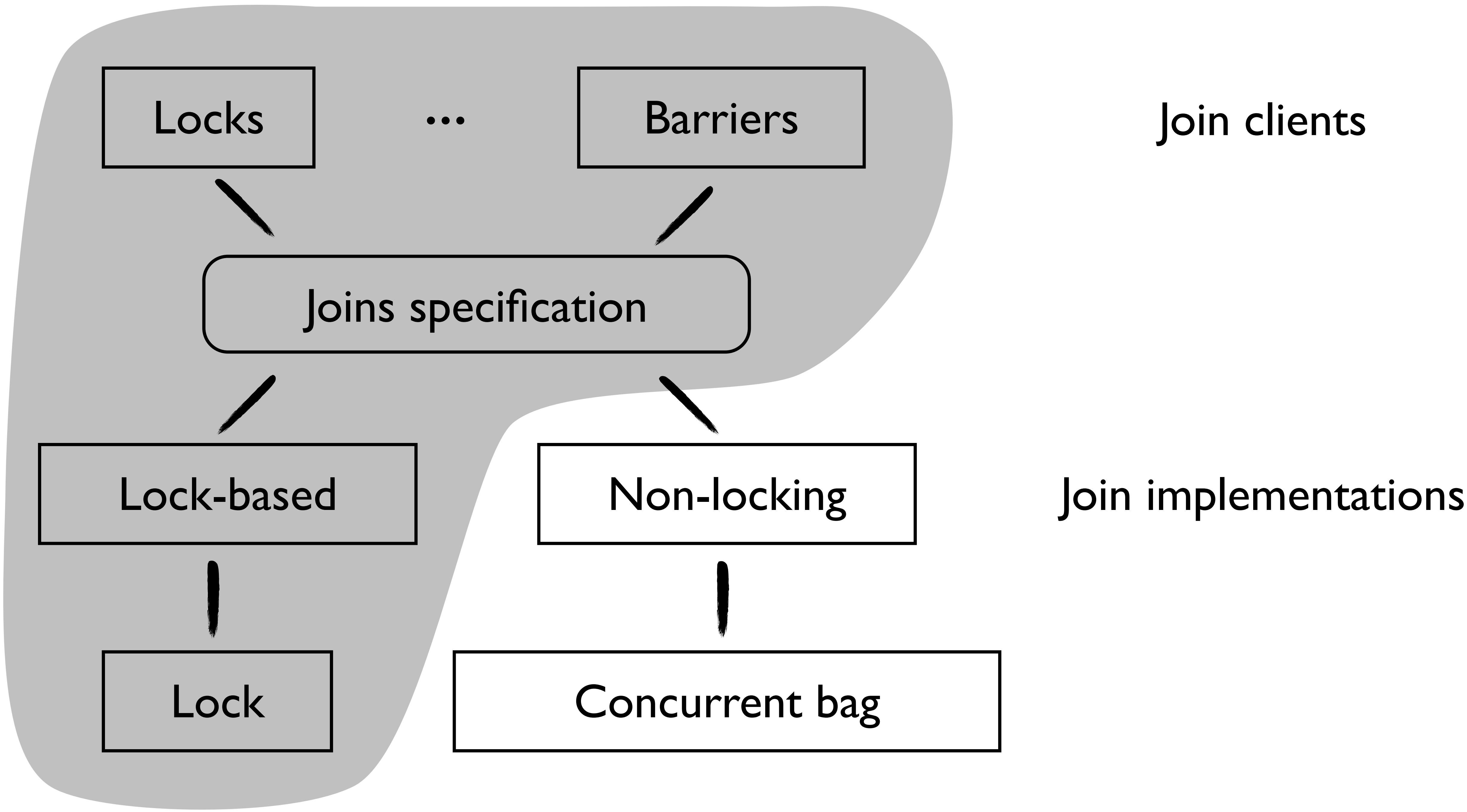
A case study in modularity



A case study in modularity



A case study in modularity



Joins example

```
class RWLock {
  public SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel unused, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join join = new Join();
    // ... initialize channels ...

    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared(); });
    join.When(acqW).And(unused).Do(() => { writer(); });
    join.When(relW).And(writer).Do(() => { unused(); });
    join.When(relR).And(shared).Do(() => {
      if (--readers == 0) unused() else shared(); });

    unused();
  }
}
```

Joins example

```
class RWLock {
  public SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel unused, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join join = new Join();
    // ... initialize channels ...

    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared(); });
    join.When(acqW).And(unused).Do(() => { writer(); });
    join.When(relW).And(writer).Do(() => { unused(); });
    join.When(relR).And(shared).Do(() => {
      if (--readers == 0) unused() else shared(); });

    unused();
  }
}
```

channels

Joins example

```
class RWLock {
  public SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel unused, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join join = new Join();
    // ... initialize channels ...

    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared(); });
    join.When(acqW).And(unused).Do(() => { writer(); });
    join.When(relW).And(writer).Do(() => { unused(); });
    join.When(relR).And(shared).Do(() => {
      if (--readers == 0) unused() else shared(); });

    unused();
  }
}
```

channels

chord

Joins example

```
class RWLock {
  public SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel unused, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join join = new Join();
    // ... initialize channels ...

    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared(); });
    join.When(acqW).And(unused).Do(() => { writer(); });
    join.When(relW).And(writer).Do(() => { unused(); });
    join.When(relR).And(shared).Do(() => {
      if (--readers == 0) unused() else shared(); });

    unused();
  }
}
```

channels

pattern

chord

Joins example

```
class RWLock {
  public SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel unused, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join join = new Join();
    // ... initialize channels ...

    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared(); });
    join.When(acqW).And(unused).Do(() => { writer(); });
    join.When(relW).And(writer).Do(() => { unused(); });
    join.When(relR).And(shared).Do(() => {
      if (--readers == 0) unused() else shared(); });

    unused();
  }
}
```

channels

pattern

chord

continuation

Joins example

```
class RWLock {
  public SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel unused, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join join = new Join();
    // ... initialize channels ...

    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared(); });
    join.When(acqW).And(unused).Do(() => { writer(); });
    join.When(relW).And(writer).Do(() => { unused(); });
    join.When(relR).And(shared).Do(() => {
      if (--readers == 0) unused() else shared(); });

    unused();
  }
}
```

channels

pattern

chord

continuation

send a message on
the unused channel

A reader/writer lock

```
class RWLock {
  public SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel unused, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join join = new Join();
    // ... initialize channels ...

    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared(); });
    join.When(acqW).And(unused).Do(() => { writer(); });
    join.When(relW).And(writer).Do(() => { unused(); });
    join.When(relR).And(shared).Do(() => {
      if (--readers == 0) unused() else shared(); });

    unused();
  }
}
```

A reader/writer lock

synchronous channels to acquire and release the lock

```
class RWLock {
  public SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel unused, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join join = new Join();
    // ... initialize channels ...

    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared(); });
    join.When(acqW).And(unused).Do(() => { writer(); });
    join.When(relW).And(writer).Do(() => { unused(); });
    join.When(relR).And(shared).Do(() => {
      if (--readers == 0) unused() else shared(); });

    unused();
  }
}
```

A reader/writer lock

synchronous channels to acquire and release the lock

asynchronous channels encode the state of the lock

```
class RWLock {
  public SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel unused, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join join = new Join();
    // ... initialize channels ...

    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared(); });
    join.When(acqW).And(unused).Do(() => { writer(); });
    join.When(relW).And(writer).Do(() => { unused(); });
    join.When(relR).And(shared).Do(() => {
      if (--readers == 0) unused() else shared(); });

    unused();
  }
}
```

A reader/writer lock

synchronous channels to acquire and release the lock

asynchronous channels encode the state of the lock

each chord matches and sends exactly one asynchronous message

```
class RWLock {
  public SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel unused, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join join = new Join();
    // ... initialize channels ...

    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared(); });
    join.When(acqW).And(unused).Do(() => { writer(); });
    join.When(relW).And(writer).Do(() => { unused(); });
    join.When(relR).And(shared).Do(() => {
      if (--readers == 0) unused() else shared(); });

    unused();
  }
}
```

A reader/writer lock

synchronous channels to acquire and release the lock

asynchronous channels encode the state of the lock

```
class RWLock {
  public SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel unused, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join join = new Join();
    // ... initialize channels ...

    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared(); });
    join.When(acqW).And(unused).Do(() => { writer(); });
    join.When(relW).And(writer).Do(() => { unused(); });
    join.When(relR).And(shared).Do(() => {
      if (--readers == 0) unused() else shared(); });

    unused();
  }
}
```

each chord matches and sends exactly one asynchronous message

initially, there is exactly one pending asynchronous message

Verification challenges

```
class RWLock {
  public SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel unused, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join join = new Join();
    // ... initialize channels ...

    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared(); });
    join.When(acqW).And(unused).Do(() => { writer(); });
    join.When(relW).And(writer).Do(() => { unused(); });
    join.When(relR).And(shared).Do(() => {
      if (--readers == 0) unused() else shared(); });

    unused();
  }
}
```

Verification challenges

```
class RWLock {
  public SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel unused, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join join = new Join();
    // ... initialize channels ...

    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared(); });
    join.When(acqW).And(unused).Do(() => { writer(); });
    join.When(relW).And(writer).Do(() => { unused(); });
    join.When(relR).And(shared).Do(() => {
      if (--readers == 0) unused() else shared(); });

    unused();
  }
}
```

state effect

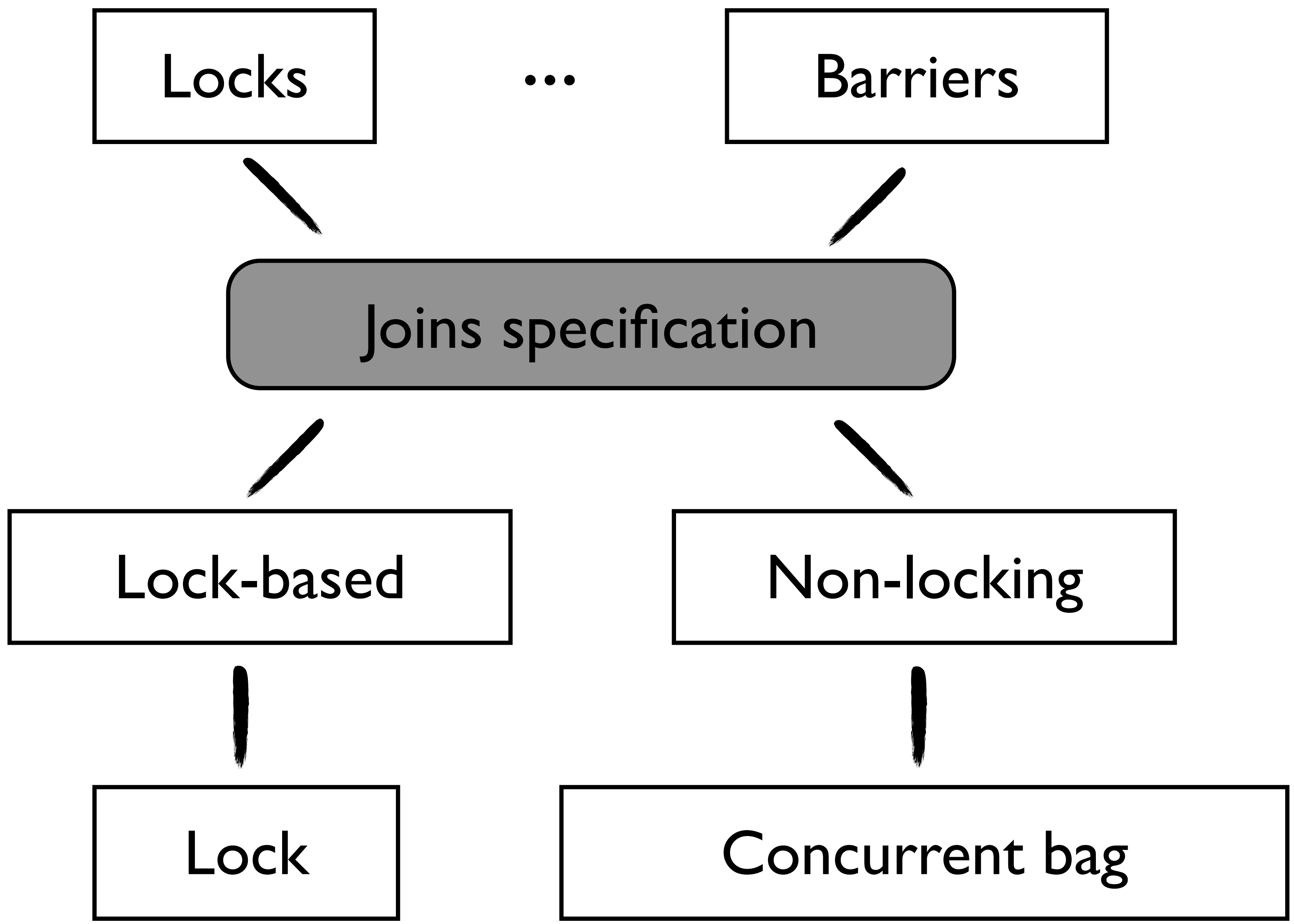
Verification challenges

```
class RWLock {
  public SyncChannel acqR, acqW, relR, relW;
  private AsyncChannel unused, shared, writer;
  private int readers = 0;

  public RWLock() {
    Join join = new Join();
    // ... initialize channels ...

    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    join.When(acqR).And(shared).Do(() => { readers++; shared(); });
    join.When(acqW).And(unused).Do(() => { writer(); });
    join.When(relW).And(writer).Do(() => { unused(); });
    join.When(relR).And(shared).Do(() => {
      if (--readers == 0) unused() else shared(); });

    unused();
  }
}
```

reentrant continuation

state effect

Joins specification



Specification

- Requirements:
  - Ownership transfer
  - Stateful reentrant continuations
- Restrict attention to non-self-modifying clients

Ideas

- Let clients pick an ownership protocol for each channel
  - The channel pre-condition describes the resources the sender is required to transfer to the recipient upon sending a message
  - The channel post-condition describes the resources the recipient is required to transfer to the sender upon receiving the message
  - The channel post-condition of asynchronous channels must be emp
- Prove chords obey the ownership protocol, assuming channels obey the ownership protocol (to support reentrancy)

Specification

- Send a message on channel c (async or sync)

$$\{\mathsf{join}(P, Q, j) * \mathsf{chan}(\mathsf{c}, j) * P(\mathsf{c})\}$$
$$\mathsf{c}()$$
$$\{\mathsf{join}(P, Q, j) * \mathsf{chan}(\mathsf{c}, j) * Q(\mathsf{c})\}$$

Specification

- Send a message on channel c (async or sync)

family of channel pre- and post-conditions, indexed by channels

$$\{\mathsf{join}(P, Q, j) * \mathsf{chan}(\mathsf{c}, j) * P(\mathsf{c})\}$$

$$\mathsf{c}()$$

$$\{\mathsf{join}(P, Q, j) * \mathsf{chan}(\mathsf{c}, j) * Q(\mathsf{c})\}$$

Specification

- Send a message on channel c (async or sync)

family of channel pre- and post-conditions, indexed by channels

transfer channel pre-condition from client to join instance

$$\{\mathsf{join}(P, Q, j) * \mathsf{chan}(\mathsf{c}, j) * P(\mathsf{c})\}$$
$$\mathsf{c}()$$
$$\{\mathsf{join}(P, Q, j) * \mathsf{chan}(\mathsf{c}, j) * Q(\mathsf{c})\}$$

transfer channel post-condition from join instance to client

Specification

- Send a message on channel c (async or sync)

family of channel pre- and post-conditions, indexed by channels

transfer channel pre-condition from client to join instance

$$\{\mathsf{join}(P, Q, j) * \mathsf{chan}(\mathsf{c}, j) * P(\mathsf{c})\}$$
$$\mathsf{c}()$$
$$\{\mathsf{join}(P, Q, j) * \mathsf{chan}(\mathsf{c}, j) * Q(\mathsf{c})\}$$

if c is an asynchronous channel, then channel post-condition must be emp

transfer channel post-condition from join instance to client

Specification

- Register a new chord with pattern p and continuation b

$$\left\{ \begin{array}{l} \mathsf{join}_{\mathsf{init\text{-}pat}}(P, Q, j) * \mathsf{pattern}(\mathsf{p}, j, X) \\ \quad * \ \mathsf{b} \mapsto \ \begin{array}{l} \{\circledast_{x \in X} P(x) * \mathsf{join}(P, Q, j)\} \\ \{\circledast_{x \in X} Q(x) * \mathsf{join}(P, Q, j)\} \end{array} \end{array} \right\}$$

$$\texttt{p.Do}(\mathsf{b})$$

$$\{\mathsf{join}_{\mathsf{init\text{-}pat}}(P, Q, j)\}$$

Specification

- Register a new chord with pattern p and continuation b

pattern p matches the multiset of channels X

$$\left\{ \begin{array}{l} \mathsf{join}_{\text{init-pat}}(P, Q, j) * \mathsf{pattern}(\mathsf{p}, j, X) \\ \quad * \ \mathsf{b} \mapsto \ \{\circledast_{x \in X} P(x) * \mathsf{join}(P, Q, j)\} \\ \qquad\qquad\quad \{\circledast_{x \in X} Q(x) * \mathsf{join}(P, Q, j)\} \end{array} \right\}$$

```
p.Do(b)
```

$$\{\mathsf{join}_{\text{init-pat}}(P, Q, j)\}$$

Specification

- Register a new chord with pattern p and continuation b

pattern p matches the multiset of channels X

$$\left\{\begin{array}{l}\mathsf{join}_{\mathsf{init\text{-}pat}}(P, Q, j) * \mathsf{pattern}(\mathsf{p}, j, X) \\ \quad * \;\; \mathsf{b} \mapsto \;\; \dfrac{\{\circledast_{x \in X} P(x) * \mathsf{join}(P, Q, j)\}}{\{\circledast_{x \in X} Q(x) * \mathsf{join}(P, Q, j)\}}\end{array}\right\}$$

$$\texttt{p.Do(b)}$$

$$\{\mathsf{join}_{\mathsf{init\text{-}pat}}(P, Q, j)\}$$

resources senders must transfer to recipient

Specification

- Register a new chord with pattern p and continuation b

pattern p matches the multiset of channels X

$$\left\{\begin{array}{l} \mathsf{join}_{\mathsf{init\text{-}pat}}(P, Q, j) * \mathsf{pattern}(\mathsf{p}, j, X) \\ \quad * \ \mathsf{b} \mapsto \ \{\underline{\circledast_{x \in X} P(x)} * \mathsf{join}(P, Q, j)\} \\ \qquad\qquad\quad \{\underline{\circledast_{x \in X} Q(x)} * \mathsf{join}(P, Q, j)\} \end{array}\right\}$$

$$\texttt{p.Do}(\mathsf{b})$$

$$\{\mathsf{join}_{\mathsf{init\text{-}pat}}(P, Q, j)\}$$

resources senders must transfer to recipient

resources recipient must transfer to senders

Specification

- Register a new chord with pattern p and continuation b

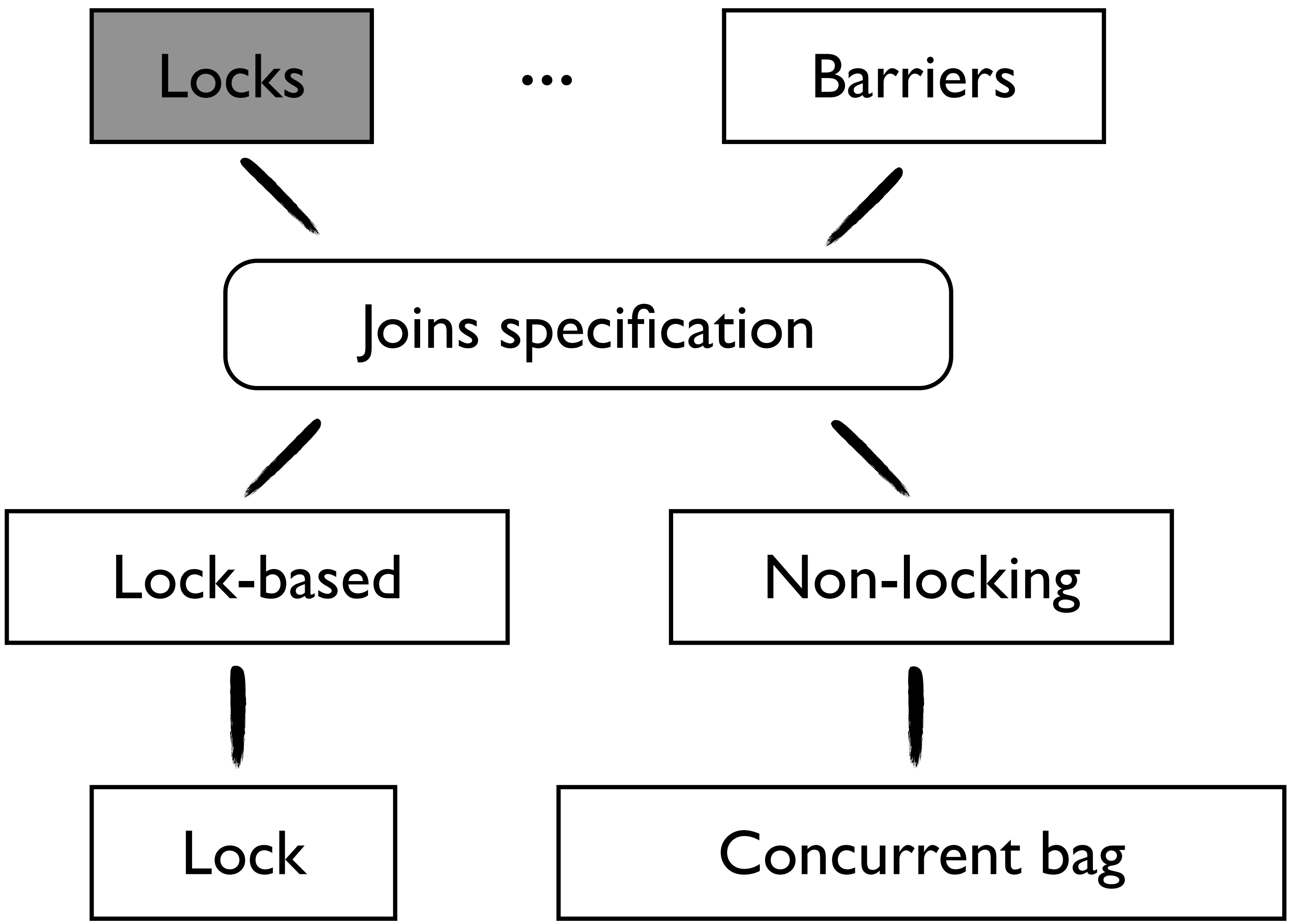
$$\left\{\begin{array}{l}\mathsf{join}_{\mathsf{init\text{-}pat}}(P,Q,j) * \mathsf{pattern}(\mathsf{p},j,X) \\ \quad * \;\; \mathsf{b} \mapsto \;\; \{\circledast_{x\in X} P(x) * \mathsf{join}(P,Q,j)\} \\ \qquad\qquad\quad \{\circledast_{x\in X} Q(x) * \mathsf{join}(P,Q,j)\}\end{array}\right\}$$

$$\texttt{p.Do(b)}$$

$$\{\mathsf{join}_{\mathsf{init\text{-}pat}}(P,Q,j)\}$$

the continuation is allowed to assume channels obey their ownership protocol

Verifying Clients



Reader/Writer lock

- Given resource invariants R and R_{ro} (picked by client) s.t.

$$\forall n \in \mathbf{N}.\ R(n) \Leftrightarrow R_{ro} * R(n+1)$$

  - R_{ro} : read permission to underlying resource
  - R(0): write permission to underlying resource
  - R(n): resource after splitting off n read permissions

Reader/Writer lock

- Given resource invariants R and R_{ro} (picked by client) s.t.

$$\forall n \in \mathbf{N}.\ R(n) \Leftrightarrow R_{ro} * R(n+1)$$

  - R_{ro} : read permission to underlying resource
  - R(0): write permission to underlying resource
  - R(n): resource after splitting off n read permissions

- The reader/writer lock satisfies the following specification

$$\{emp\}\quad \texttt{acqR}()\quad \{R_{ro}\} \qquad\qquad \{R_{ro}\}\quad \texttt{relR}()\quad \{emp\}$$
$$\{emp\}\quad \texttt{acqW}()\quad \{R(0)\} \qquad\qquad \{R(0)\}\quad \texttt{relW}()\quad \{emp\}$$

- Assign pre-conditions to asynchronous channels

$$P(\texttt{unused}) = \texttt{readers} \mapsto 0 * R(0)$$
$$P(\texttt{shared}) = \exists n \in \mathbb{N}.\ \texttt{readers} \mapsto n * R(n) * n > 0$$
$$P(\texttt{writer}) = \texttt{readers} \mapsto 0$$

- Assign pre- and post-conditions to synchronous channels

$P(\texttt{acqR}) = emp$	$Q(\texttt{acqR}) = R_{ro}$
$P(\texttt{acqW}) = emp$	$Q(\texttt{acqW}) = R(0)$
$P(\texttt{relR}) = R_{ro}$	$Q(\texttt{relR}) = emp$
$P(\texttt{relW}) = R(0)$	$Q(\texttt{relW}) = emp$

- Prove chords obey channel ownership protocol

```
class RWLock {
  ...
  public int readers = 0;

  public RWLock() {
    ...
    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    ...
  }
}
```

- Prove chords obey channel ownership protocol

```
class RWLock {
  ...
  public int readers = 0;

  public RWLock() {
    ...
    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    ...
  }
}
```

$\{P(\texttt{acqR}) * P(\texttt{unused}) * join(P, Q, j)\}$

```
readers++


shared();
```

$\{Q(\texttt{acqR}) * Q(\texttt{unused}) * join(P, Q, j)\}$

- Prove chords obey channel ownership protocol

```
class RWLock {
  ...
  public int readers = 0;

  public RWLock() {
    ...
    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    ...
  }
}
```

$\{\texttt{readers} \mapsto 0 * R(0) * join(P, Q, j)\}$

  `readers++`

$\{\texttt{readers} \mapsto 1 * R(1) * R_{ro} * join(P, Q, j)\}$

  `shared();`

$\{R_{ro} * join(P, Q, j)\}$

- Prove chords obey channel ownership protocol

```
class RWLock {
  ...
  public int readers = 0;

  public RWLock() {
    ...
    join.When(acqR).And(unused).Do(() => { readers++; shared(); });
    ...
  }
}
```

$\{\texttt{readers} \mapsto 0 * R(0) * join(P, Q, j)\}$

$\texttt{readers++}$

$\{\texttt{readers} \mapsto 1 * R(1) * R_{ro} * join(P, Q, j)\}$

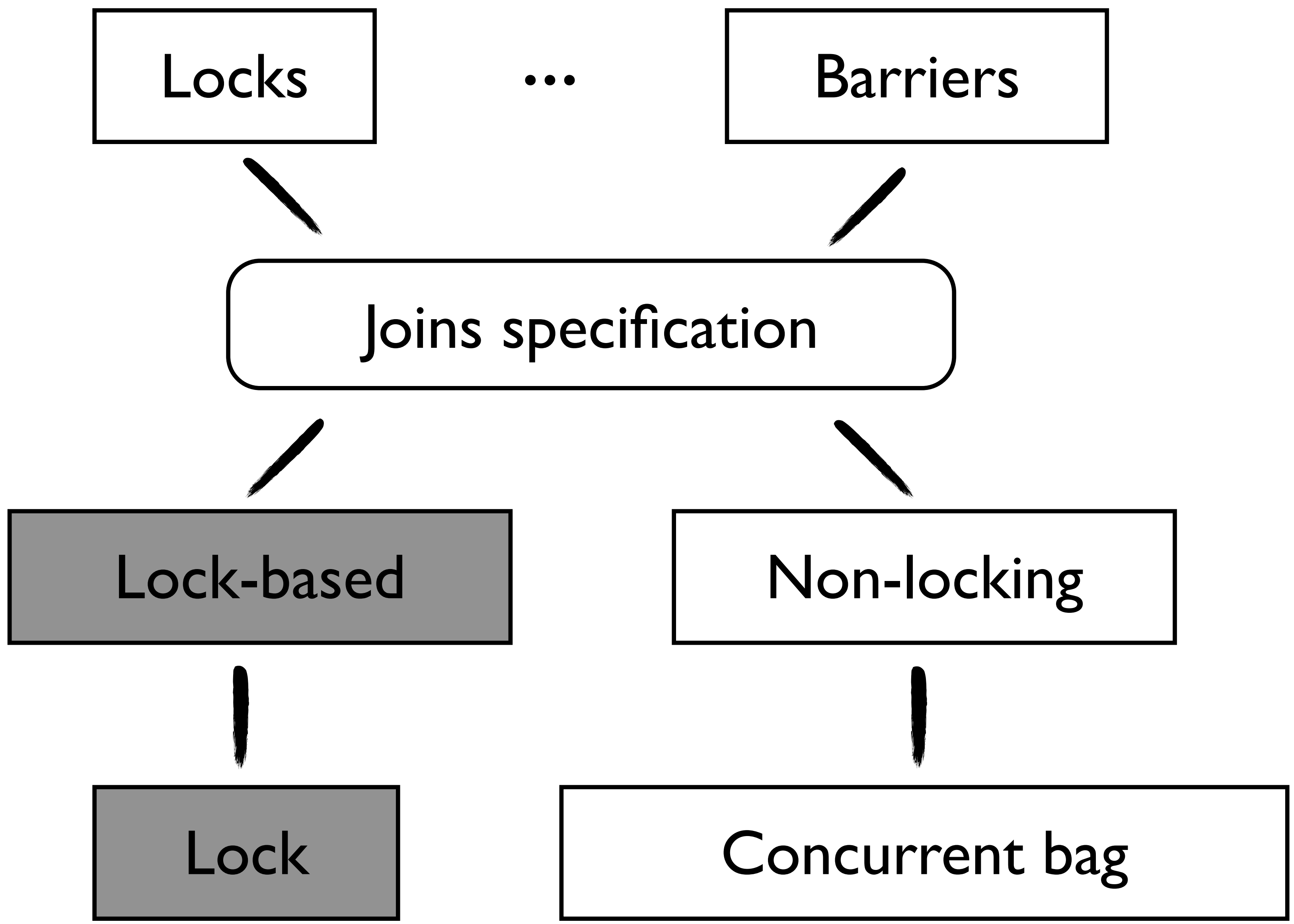
$\texttt{shared();}$

$\{R_{ro} * join(P, Q, j)\}$

$P(\texttt{shared}) = \exists n \in \mathbb{N}_+.$

$\texttt{readers} \mapsto n * R(n)$

Verifying an Implementation



Verifying an Implementation

- Challenges:
  - High-level join primitives implemented using shared mutable state
  - Definition of recursive representation predicates

Verifying an Implementation

- Challenges:
  - High-level join primitives implemented using shared mutable state
  - Definition of recursive representation predicates

guarded recursion & step-indexed model

Messages

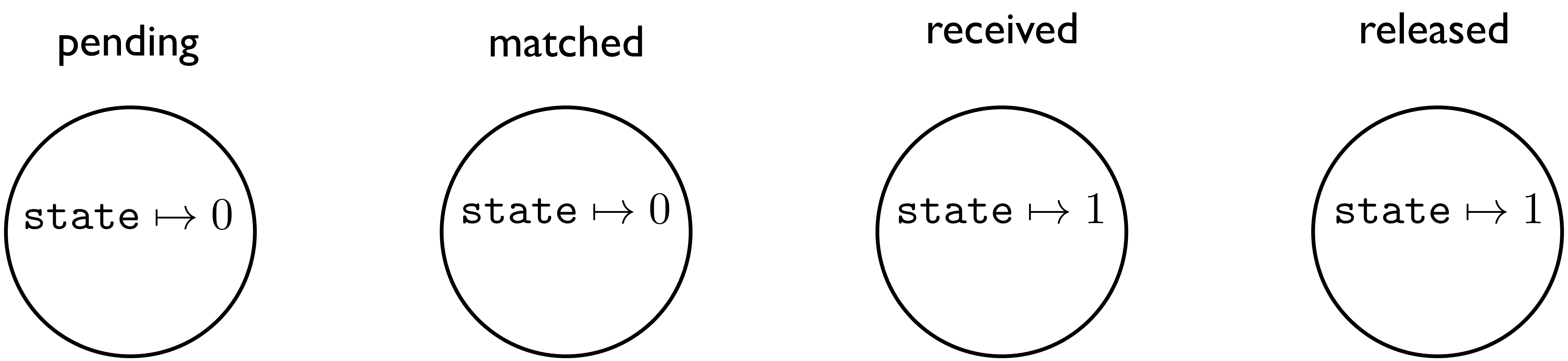
```
class Message {
  public int state;

  public Message() {
    state = 0;
  }

  public void Receive() {
    state = 1;
  }
}
```

Messages

- Assume channel pre- and post-conditions P and Q
- Imagine a message on channel c

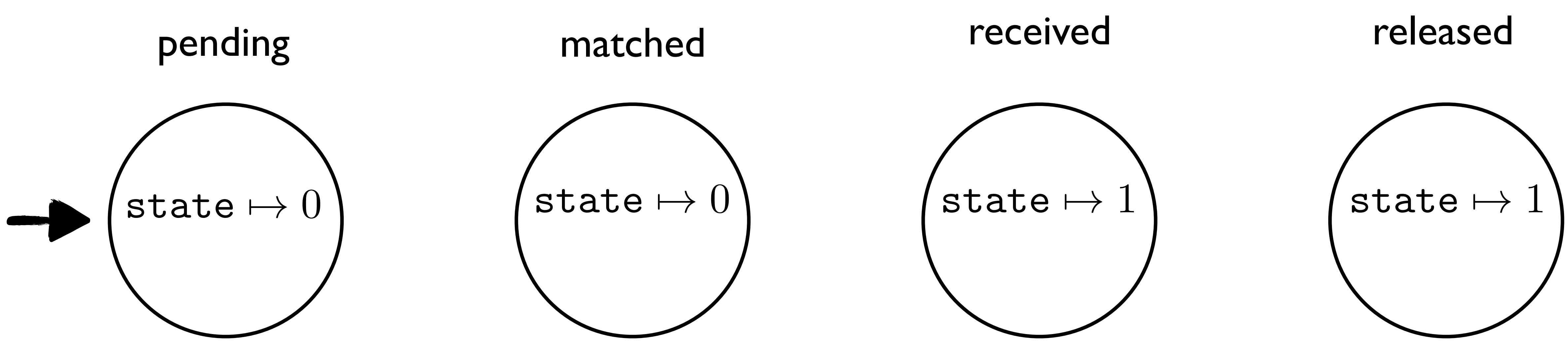
Messages

- Assume channel pre- and post-conditions P and Q
- Imagine a message on channel c



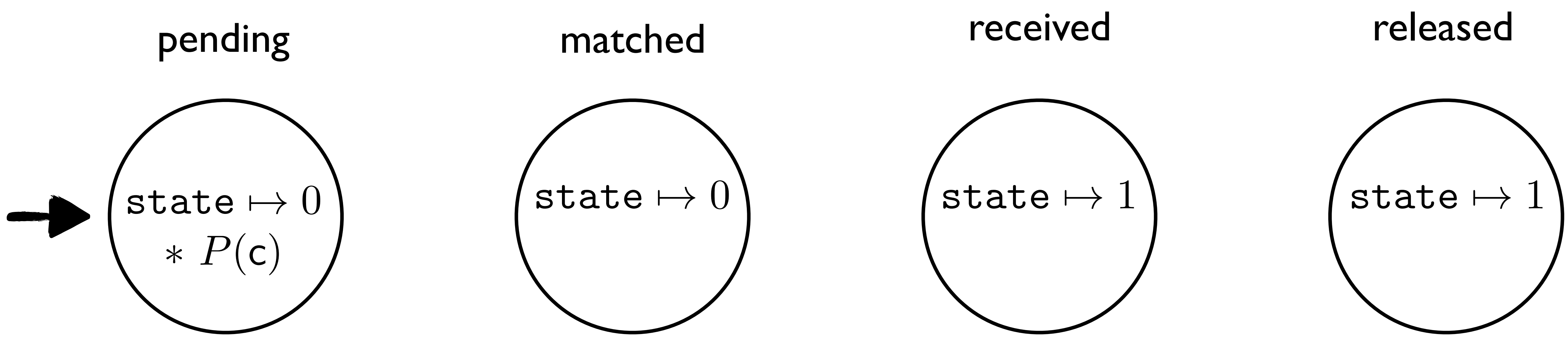
Messages

- Assume channel pre- and post-conditions P and Q
- Imagine a message on channel c



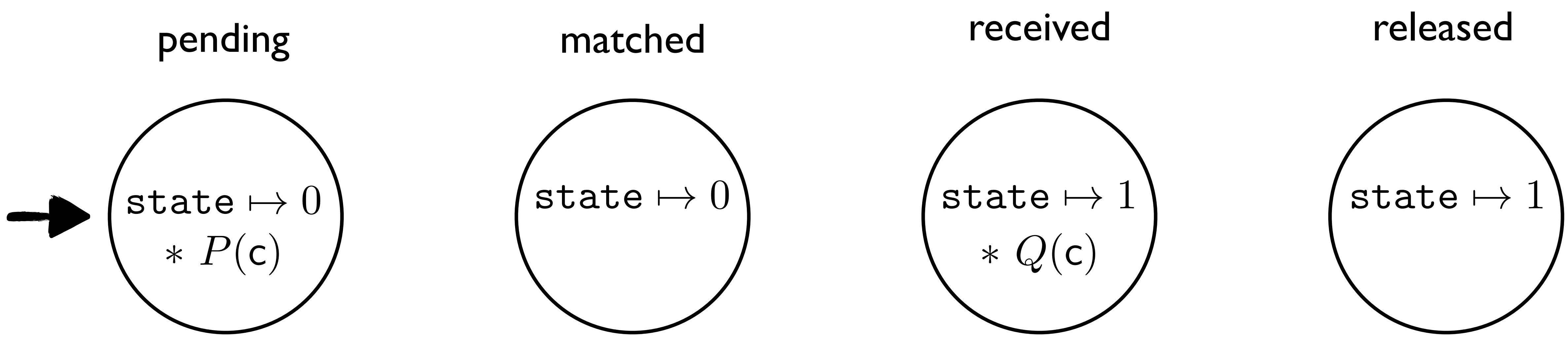
Messages

- Assume channel pre- and post-conditions P and Q
- Imagine a message on channel c



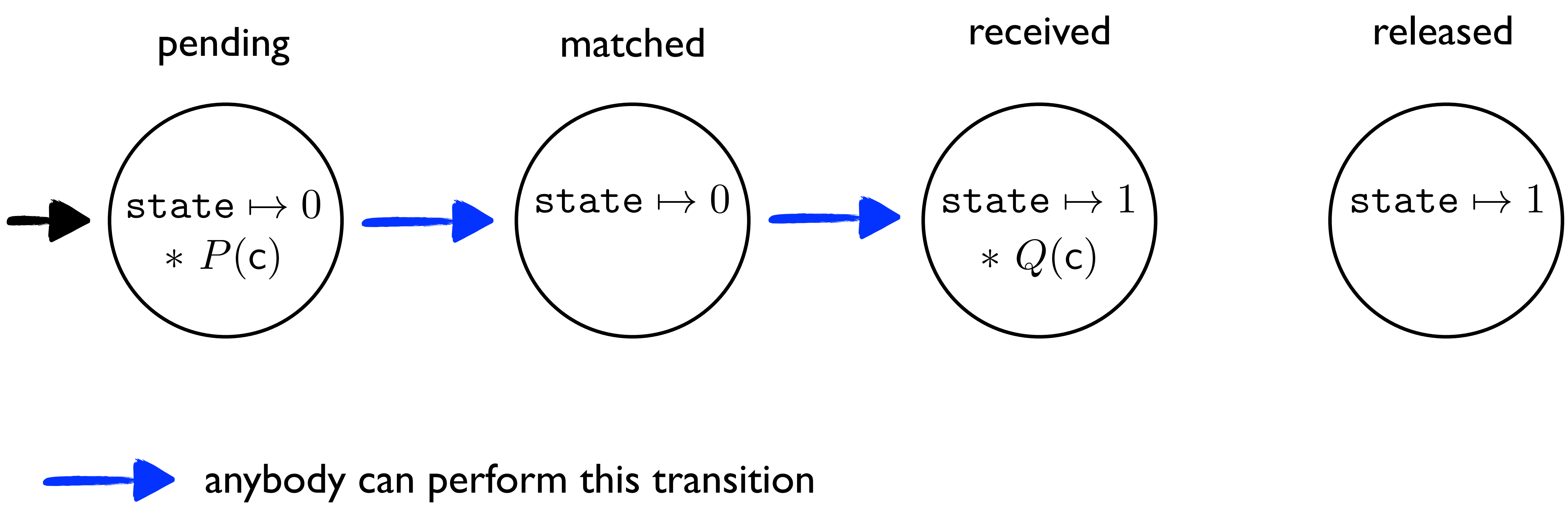
Messages

- Assume channel pre- and post-conditions P and Q
- Imagine a message on channel c



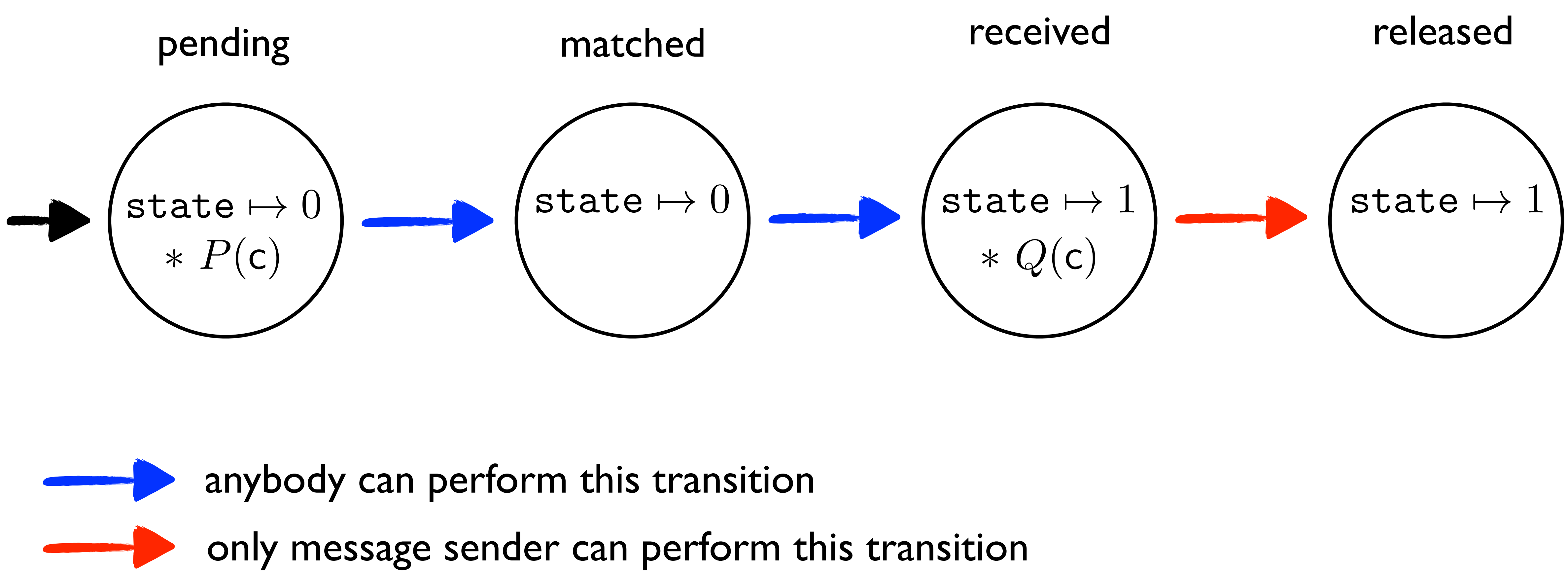
Messages

- Assume channel pre- and post-conditions P and Q
- Imagine a message on channel c



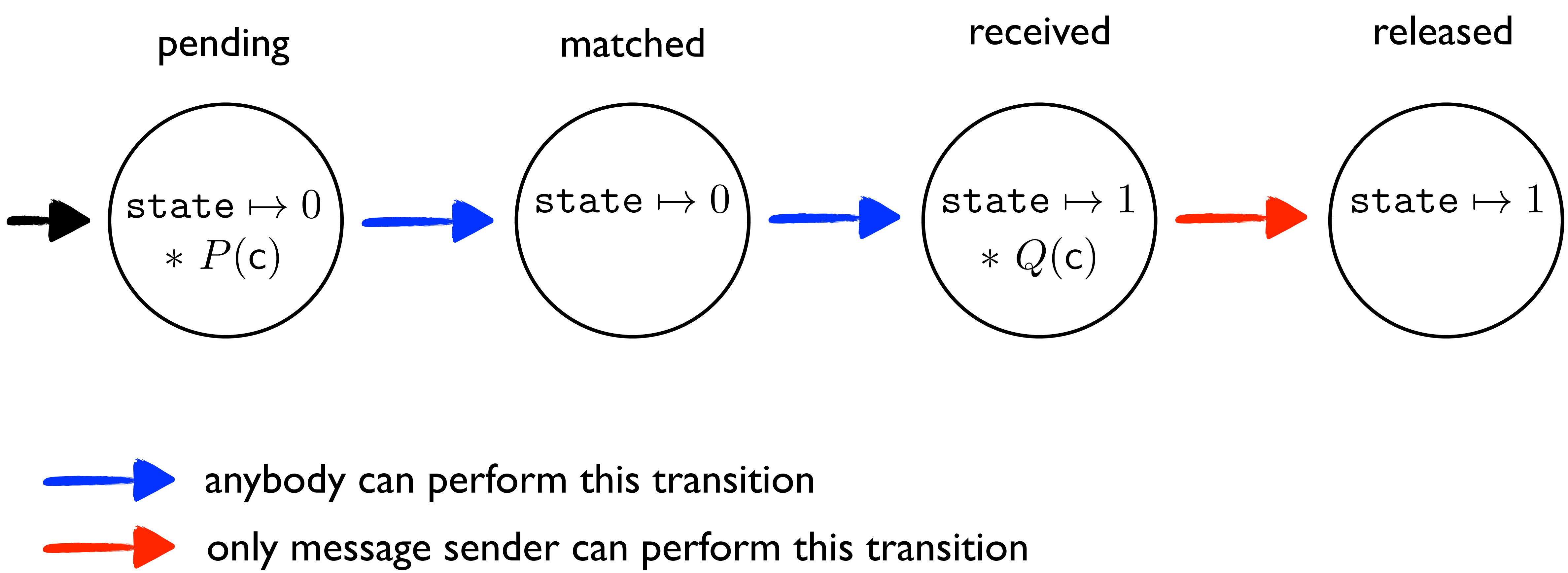
Messages

- Assume channel pre- and post-conditions P and Q
- Imagine a message on channel c



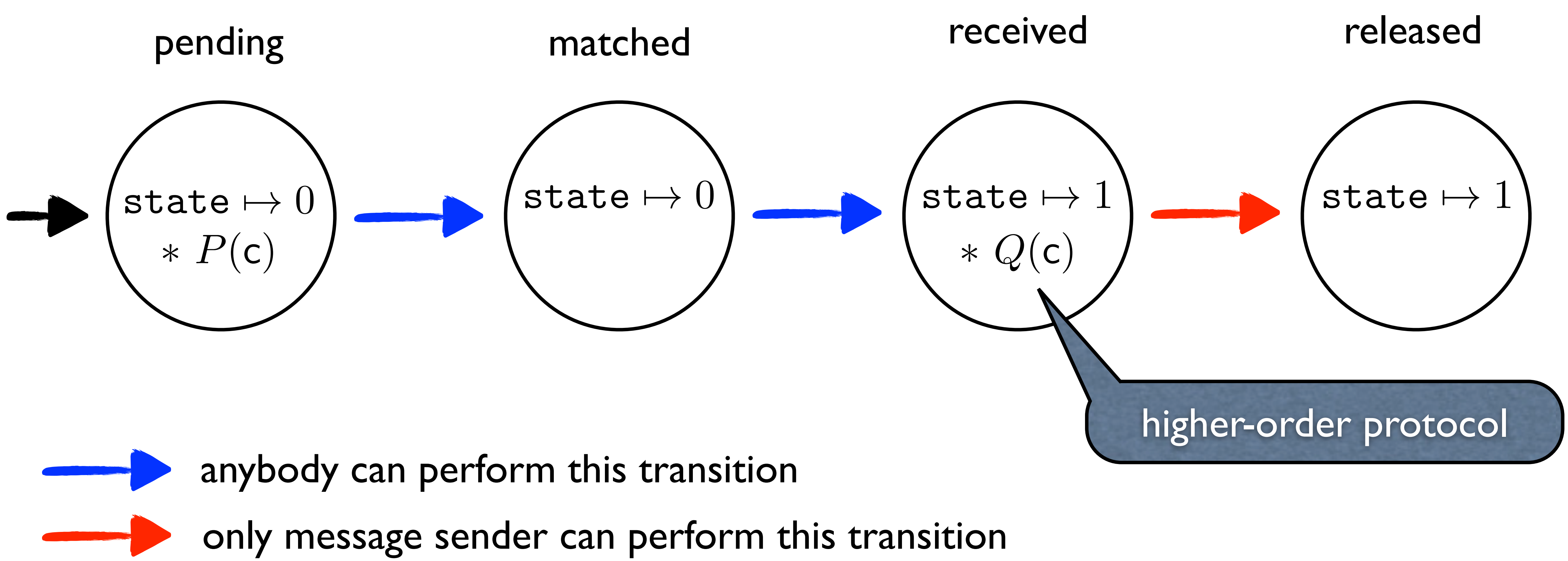
Messages

- Use Concurrent Abstract Predicates [Dinsdale-Young et. al.] to impose this low-level protocol on messages



Messages

- Use Concurrent Abstract Predicates [Dinsdale-Young et. al.] to impose this low-level protocol on messages



HOCAP

- Higher-order protocols are difficult; the previous proposal [Dodds et. al.] from POPL11 is unsound!

HOCAP

- Higher-order protocols are difficult; the previous proposal [Dodds et. al.] from POPL11 is unsound!
- We restrict attention to state-independent higher-order protocols. An assertion P is expressible using state-independent protocols (SIPs) iff

$$\exists R, S : Prop.\ valid\ (P \Leftrightarrow R * S) \wedge noprotocol(R) \wedge nostate(S)$$

invariant under arbitrary changes to protocols

invariant under arbitrary changes to the state

- We require all channel pre- and post-conditions to be expressible using SIPs

Summary

- Verified the lock-based joins implementation against the high-level joins specification
- Verified a couple of classic synchronization primitives using the high-level joins specification
- Given a logic and model for HOCAP with support for state-independent higher-order protocols
- TRs available at www.itu.dk/~kasv

Questions?

Higher-order protocols in CAP

Let

$$P \stackrel{\text{def}}{=} (x \mapsto 0 * (\boxed{y \mapsto 0}_I^r \vee \boxed{y \mapsto 0}_J^r)) \vee (x \mapsto 1 * \boxed{y \mapsto 0}_J^r)$$

where

$$I[\alpha] : y \mapsto 1 \rightsquigarrow y \mapsto 2$$
$$J[\alpha] : y \mapsto 1 \rightsquigarrow y \mapsto 3$$
$$K[\alpha] : P \rightsquigarrow P$$

then P is stable, but $\boxed{P}_K^{r'}$ is not