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

- Lock-based
  - Lock
- Non-locking
  - Concurrent bag

Join implementations
A case study in modularity

- Joins specification
  - Lock-based
    - Lock
  - Non-locking
    - Concurrent bag
A case study in modularity

Locks → Joins specification → Lock-based → Lock

... → Joins specification → Non-locking → Concurrent bag

Barriers

Join clients

Join implementations
A case study in modularity
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();
    }
}
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();
    }
}
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();
    }
}
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();
    }
}
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

channels
pattern
chord
continuation
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();
    }
}
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

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();
    }
}
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();
    }
}
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();
    }
}
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();
    }
}
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 specification

- Locks
- ... (omitted)
- Barriers

- Lock-based
  - Lock
- Non-locking
  - Concurrent bag
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 empty

• 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)

\[ \left\{ \text{join}(P, Q, j) \ast \text{chan}(c, j) \ast P(c) \right\} \]
\[ c() \]
\[ \left\{ \text{join}(P, Q, j) \ast \text{chan}(c, j) \ast Q(c) \right\} \]
Specification

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

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

\[
\{ \text{join}(P, Q, j) \ast \text{chan}(c, j) \ast P(c) \}
\]

\[
c()
\]

\[
\{ \text{join}(P, Q, j) \ast \text{chan}(c, j) \ast Q(c) \}
\]
Specification

• Send a message on channel $c$ (async or sync)

\[
\{\text{join}(P, Q, j) \times \text{chan}(c, j) \times P(c)\}
\]

\[
\{\text{join}(P, Q, j) \times \text{chan}(c, j) \times Q(c)\}
\]
Specification

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

\{\text{join}(P, Q, j) \ast \text{chan}(c, j) \ast P(c)\}
\text{c()}
\{\text{join}(P, Q, j) \ast \text{chan}(c, j) \ast Q(c)\}

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

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

transfer channel pre-condition from client to join instance

transfer channel post-condition from join instance to client
Specification

- Register a new chord with pattern $p$ and continuation $b$

\[
\begin{align*}
\text{join}_{\text{init-pat}}(P, Q, j) & \ast \text{pattern}(p, j, X) \\
& \ast b \mapsto \left\{ \bigotimes_{x \in X} P(x) \ast \text{join}(P, Q, j) \right\} \\
& \quad \left\{ \bigotimes_{x \in X} Q(x) \ast \text{join}(P, Q, j) \right\} \\
p \cdot \text{Do}(b) \\
\{ \text{join}_{\text{init-pat}}(P, Q, j) \}
\end{align*}
\]
Specification

• Register a new chord with pattern \( p \) and continuation \( b \)

\[
\begin{align*}
\text{pattern } p & \text{ matches the multiset of channels } X \\
\{ \text{join}_{\text{init-pat}}(P, Q, j) \} & \star \{ \text{pattern}(p, j, X) \} \\
* \ b & \mapsto \{ \bigotimes_{x \in X} P(x) \star \text{join}(P, Q, j) \} \\
& \quad \{ \bigotimes_{x \in X} Q(x) \star \text{join}(P, Q, j) \} \\
\{ \text{p.Do}(b) \} & \{ \text{join}_{\text{init-pat}}(P, Q, j) \}
\end{align*}
\]
Specification

- Register a new chord with pattern p and continuation b

\[
\begin{align*}
\text{join}_{\text{init-pat}}(P, Q, j) & \ast \text{pattern}(p, j, X) \\
\ast b & \mapsto \begin{cases} 
\{ \bigotimes_{x \in X} P(x) \ast \text{join}(P, Q, j) \} \\
\{ \bigotimes_{x \in X} Q(x) \ast \text{join}(P, Q, j) \} 
\end{cases} \\
p. \text{Do}(b) \\
\{ \text{join}_{\text{init-pat}}(P, Q, j) \}
\end{align*}
\]

- pattern p matches the multiset of channels X
- resources senders must transfer to recipient
Specification

- Register a new chord with pattern $p$ and continuation $b$

\[
\{ \text{join} \text{\textunderscore init\textunderscore pat}(P, Q, j) \} * \{ \text{pattern}(p, j, X) \}
\]

\[
* b \mapsto \{ \bigotimes_{x \in X} P(x) * \text{join}(P, Q, j) \}, \bigotimes_{x \in X} Q(x) * \text{join}(P, Q, j) \}\}
\]  
\[
p. \text{Do}(b)
\]

- Resources senters must transfer to recipient
- Resources recipient must transfer to senders
Specification

- Register a new chord with pattern \( p \) and continuation \( b \)

\[
\begin{align*}
\text{join}_{\text{init-pat}}(P, Q, j) & \ast \text{pattern}(p, j, X) \\
\ast b & \mapsto \{ \bigotimes_{x \in X} P(x) \ast \text{join}(P, Q, j) \} \\
& \cup \{ \bigotimes_{x \in X} Q(x) \ast \text{join}(P, Q, j) \} \\
p.\text{Do}(b) & \\
\{ \text{join}_{\text{init-pat}}(P, Q, j) \} \\
\end{align*}
\]

the continuation is allowed to assume channels obey their ownership protocol
Verifying Clients

- Locks
- Barriers
- Joins specification
  - Lock-based
    - Lock
  - Non-locking
    - Concurrent bag
Reader/Writer lock

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

  $$\forall n \in \mathbb{N}. R(n) \iff R_{ro} \ast R(n + 1)$$

- $R_{ro}$: 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.
  \[ \forall n \in \mathbb{N}. \ R(n) \Leftrightarrow R_{ro} \star R(n + 1) \]
  - $R_{ro}$: 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

  \[
  \begin{array}{ll}
  \{emp\} & \text{acqR()} \ {R_{ro}} \\
  \{emp\} & \text{acqW()} \ {R(0)} \\
  \{emp\} & \text{relR()} \ {emp} \\
  \{R(0)\} & \text{relW()} \ {emp}
  \end{array}
  \]
• Assign pre-conditions to asynchronous channels

\[ P(\text{unused}) = \text{readers} \mapsto 0 \ast R(0) \]
\[ P(\text{shared}) = \exists n \in \mathbb{N}. \text{readers} \mapsto n \ast R(n) \ast n > 0 \]
\[ P(\text{writer}) = \text{readers} \mapsto 0 \]

• Assign pre- and post-conditions to synchronous channels

\[ P(\text{acqR}) = \text{emp} \quad Q(\text{acqR}) = R_{ro} \]
\[ P(\text{acqW}) = \text{emp} \quad Q(\text{acqW}) = R(0) \]
\[ P(\text{relR}) = R_{ro} \quad Q(\text{relR}) = \text{emp} \]
\[ P(\text{relW}) = R(0) \quad Q(\text{relW}) = \text{emp} \]
• 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(acqR) \land P(unused) \land join(P, Q, j)\}
readers++

shared();
\{Q(acqR) \land Q(unused) \land 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(); });
    }

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

    {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(); });
    }
}

\[
\begin{align*}
\{ \text{readers} & \mapsto 0 \ast R(0) \ast \text{join}(P, Q, j) \} \\
\text{readers} & \mapsto \text{readers} + 1 \\
\{ \text{readers} & \mapsto 1 \ast R(1) \ast R_{ro} \ast \text{join}(P, Q, j) \} \\
\text{shared}() & \\
\{ R_{ro} \ast \text{join}(P, Q, j) \} \\
P(\text{shared}) & = \exists n \in \mathbb{N}_+. \\
\text{readers} & \mapsto n \ast R(n)
\end{align*}
\]
Verifying an Implementation

- Locks
- Lock-based
  - Lock
- Non-locking
  - Concurrent bag
- ... (omitted)
- Barriers
  - Joins specification
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
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

pending

state $\mapsto 0$

received

state $\mapsto 1$
Messages

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

<table>
<thead>
<tr>
<th>State</th>
<th>Pending</th>
<th>Matched</th>
<th>Received</th>
<th>Released</th>
</tr>
</thead>
<tbody>
<tr>
<td>$state \leftrightarrow 0$</td>
<td>$state \leftrightarrow 0$</td>
<td>$state \leftrightarrow 1$</td>
<td>$state \leftrightarrow 1$</td>
<td></td>
</tr>
</tbody>
</table>
Messages

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

![Diagram showing states: pending (state $\rightarrow 0$), matched (state $\rightarrow 0$), received (state $\rightarrow 1$), released (state $\rightarrow 1$).]
Messages

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

```
- pending: state → 0
  * P(c)
- matched: state → 0
- received: state → 1
- released: state → 1
```
Assume channel pre- and post-conditions P and Q

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

\[
\begin{align*}
\text{pending} & \quad \text{matched} & \quad \text{received} & \quad \text{released} \\
\text{state} & \mapsto 0 & \text{state} & \mapsto 0 & \text{state} & \mapsto 1 & \text{state} & \mapsto 1 \\
* P(c) & & * Q(c) & & & & \\
\end{align*}
\]

\[\text{anybody can perform this transition}\]
• Assume channel pre- and post-conditions P and Q
• Imagine a message on channel c
- Use Concurrent Abstract Predicates [Dinsdale-Young et. al.] to impose this low-level protocol on messages.
• Use Concurrent Abstract Predicates [Dinsdale-Young et. al.] to impose this low-level protocol on messages

\[ \text{state} \rightarrow 0 \quad \exists \ P(c) \]

\[ \text{matched} \rightarrow 0 \]

\[ \text{received} \rightarrow 0 \quad \exists \ Q(c) \]

\[ \text{released} \rightarrow 1 \]

anybody can perform this transition

only message sender can perform this transition
• Higher-order protocols are difficult; the previous proposal [Dodds et. al.] from POPL 11 is unsound!
HOCAST

• 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

$$\exists R, S : Prop. \text{ valid } (P \Leftrightarrow R \ast S) \land \text{noprotocol}(R) \land \text{nostate}(S)$$

• 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

\[ P \overset{\text{def}}{=} (x \mapsto 0 \ast (y \mapsto 0)^r_I \lor y \mapsto 0)^r_J) \lor (x \mapsto 1 \ast y \mapsto 0)^r_J \]

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 \( P^r_K \) is not