Multris: Functional Verification of Multiparty Message Passing in Separation Logic

Jonas Kastberg Hinrichsen
Aarhus University

Jules Jacobs
Cornell University

Robbert Krebbers
Radboud University Nijmegen

Jesper Bengtson  Daniël Louwrink  Léon Gondelman
Mário Pereira  Amin Timany  Lars Birkedal
Me, Actris, and The Iris Workshop

Future work

- Semantic model of Session Types via logical relations
  \[ r \rightarrow \text{Val} \rightarrow \text{iProp} \]
  \[ \int \lambda v. \exists n. v = n \]
  \[ \int \rho \Rightarrow \text{??} \]
- Multi-party Dependent Separation Protocols (Based on [Honda et al., POPL'08])
- Linearity of channels through Iron
  - Preventing dropping of channel obligation
- Communication between distributed systems

[POPL'20] Actris, 1st Iris Workshop
[CPP'21] Semantic Session Types
[LMCS'22] Actris 2.0, 2nd Iris Workshop
[ICFP'23a] Actris in Distributed Systems, 2nd/3rd Iris Workshop (Léon/Me)
[ICFP'23b] MiniActris, 3rd Iris Workshop (Jules)
[POPL'24] LinearActris
Multris = Multiparty Actris

Actris = Verification system for message passing in Iris
Message Passing

Well-structured approach to writing concurrent (/distributed) programs
- Individual components behave as individual actors
- Actors interact based on predetermined global protocol
- We consider reliable channels: Messages are never duplicated or reordered

Message passing is not a silver bullet
- Often mixed with other programming mechanisms
  - Such as: shared memory, higher-order functions, recursion
- Many bugs happen when these mechanisms intersect
- We want functional verification that spans these intersections

Actris: program logic for verifying message passing programs
- Actris (via Iris) supports all of the above

But what about multiparty message passing?
Multiparty Message Passing

Multiparty message passing
- Message passing with dependent interactions between multiple actors
- Like a game of telephone! Or leader election

Dependencies are hard to get right
- Few results exists for functional verification
- Multiple unsound results in the literature

Idea: Modify Actris to support multiparty message passing
- Inheriting verification alongside other programming mechanisms
- Inheriting foundationally proven soundness theorem (via Iris)

Scope: Synchronous message passing in shared memory
- Synchronous: Sender and receiver block until exchange
- Shared memory: Channels implemented via references in ML-like language
Multiparty Message Passing in Shared Memory

Multiparty channels in shared memory:

- `new_chan(n)` Creates a multiparty channel with $n$ parties, returning a tuple $(c_0, ..., c_{(n-1)})$ of endpoints.
- $c_i[j].send(v)$ Sends a value $v$ via endpoint $c_i$ to party $j$ (synchronously).
- $c_i[j].recv()$ Receives a value via endpoint $c_i$ from party $j$.

Example Program: Roundtrip

```plaintext
let (c_0, c_1, c_2) = new_chan(3) in
fork {let x = c_1[0].recv() in c_1[2].send(x + 1)};
fork {let x = c_2[1].recv() in c_2[0].send(x + 1)};
c_0[1].send(40); let x = c_0[2].recv() in assert(x = 42)
```
Safety and Functional Correctness

Example Program: Roundtrip

```ocaml
let (c0, c1, c2) = new_chan(3) in
fork {let x = c1[0].recv() in c1[2].send(x + 1)};
fork {let x = c2[1].recv() in c2[0].send(x + 1)};
c0[1].send(40); let x = c0[2].recv() in assert(x = 42)
```

Goal: Prove crash-freedom (safety) and verify asserts (functional correctness)

<table>
<thead>
<tr>
<th>Safety</th>
<th>Functional Correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type systems</td>
<td>Program logics</td>
</tr>
<tr>
<td>Multiparty session types</td>
<td>???</td>
</tr>
<tr>
<td><code>c0 : ![1]Z. ?[2]Z. end</code></td>
<td></td>
</tr>
<tr>
<td><code>c1 : ?[0]Z. ![2]Z. end</code></td>
<td>???</td>
</tr>
<tr>
<td><code>c2 : ?[1]Z. ![0]Z. end</code></td>
<td></td>
</tr>
</tbody>
</table>

! is send, ? is receive

---

Jonas Kastberg Hinrichsen, Jules Jacobs, and Robbert Krebbers

Functional Verification of Multiparty Message Passing in Separation Logic
Key Idea

**Prior Work:** Binary protocols

- **Session Types:** $!\mathbb{Z} \ldots ?\mathbb{Z}. \text{end}$
- **Actris protocols:** $!(x : \mathbb{Z}) \langle x \rangle. ?\langle x + 2 \rangle. \text{end}$

**Key Idea:** Multiparty protocols!

- **Multiparty Session Types:** $![i]!\mathbb{Z} \ldots ![j]!\mathbb{Z}. \text{end}$
- **Multiparty Actris protocols:** $![i](x : \mathbb{Z}) \langle x \rangle. ![j] \langle x + 2 \rangle. \text{end}$

**Example Program: Roundtrip**

\[
\text{c}_0[1].\text{send}(40); \text{let } x = \text{c}_0[2].\text{recv()} \text{ in assert}(x = 42)
\]

**Challenge:** How to guarantee consistent global communication?
Challenge: How to guarantee consistent global communication?

```plaintext
define (c_0, c_1, c_2) = new_chan(3) in
fork {let x = c_1[0].recv() in c_1[2].send(x + 1)};
fork {let x = c_2[1].recv() in c_2[0].send(x + 1)};
c_0[1].send(40); let x = c_0[2].recv() in assert(x = 42)
```

Prior work: Syntactic duality

- $c_0 : ![1] \mathbb{Z}. ?[2] \mathbb{Z}. \text{end}$
- $c_1 : ?[0] \mathbb{Z}. ![2] \mathbb{Z}. \text{end}$
- $c_2 : ?[1] \mathbb{Z}. ![0] \mathbb{Z}. \text{end}$

This work: Semantic duality

- $c_0 \mapsto ! [1] (x : \mathbb{Z}) \langle x \rangle . ! [2] \langle x + 2 \rangle . \text{end}$
- $c_1 \mapsto ? [0] (x : \mathbb{Z}) \langle x \rangle . ! [2] \langle x + 1 \rangle . \text{end}$
- $c_2 \mapsto ? [1] (x : \mathbb{Z}) \langle x \rangle . ! [0] \langle x + 1 \rangle . \text{end}$

Key Idea: Define and prove consistency via separation logic!
Contributions

Multiparty Actris protocols
- Rich specification language for describing multiparty message passing
- Protocol consistency defined and proven in separation logic

Foundational functional verification via Multris
- Program logic for verifying multiparty message passing in Iris
- Support for language-parametric instantiation of Multiparty Actris

Verification of suite of multiparty programs
- Increasingly intricate variations of the roundtrip program
- Chang and Roberts ring leader election algorithm

Full mechanisation in Coq
- With tactic support for channels primitives and protocol consistency
Roadmap of this talk

Tour of Multiparty Actris
- Multiparty dependent separation protocols and protocol consistency
- Program logic rules
- Verification of suite of roundtrip variations

Verification of Chang and Roberts ring leader election algorithm
- Overview of algorithm
- Ring leader election protocol
- Verification of algorithm

Language-parametricity of Multiparty Actris
- Multiparty Actris ghost theory

Conclusion and Future Work
Tour of Multiparty Actris
Roundtrip Example

Roundtrip program:

```ml
let (c₀, c₁, c₂) = new_chan(3) in
fork {let x = c₁[0].recv() in c₁[2].send(x + 1)};
fork {let x = c₂[1].recv() in c₂[0].send(x + 1)};
c₀[1].send(40); let x = c₀[2].recv() in assert(x = 42)
```

**Goal:** Prove crash-freedom (safety) and verify asserts (functional correctness)
Channel endpoint ownership: \( c \mapsto p \)

Protocols: \(! [i] (\vec{x}:\vec{r}) \langle v \rangle . p \ | \ ?[i] (\vec{x}:\vec{r}) \langle v \rangle . p \ | \ \text{end}\)

Example: \(! [1] (x : \mathbb{Z}) \langle x \rangle . ?[2] \langle x + 2 \rangle . \text{end}\)

Rules:

**Ht-send**

\[\{ c \mapsto ! [i] (\vec{x}:\vec{r}) \langle v \rangle . p \} \cdot c[i].\text{send}(v[\vec{t}/\vec{x}]) \cdot \{ c \mapsto p[\vec{t}/\vec{x}] \}\]

**Ht-recv**

\[\{ c \mapsto ?[i] (\vec{x}:\vec{r}) \langle v \rangle . p \} \cdot c[i].\text{recv()} \cdot \{ w . \exists \vec{t}. w = v[\vec{t}/\vec{x}] \cdot c \mapsto p[\vec{t}/\vec{x}] \}\]

**Ht-new**

\[\{ \text{CONSISTENT } \vec{p} \star |\vec{p}| = n + 1 \} \cdot \text{new-chan}(|\vec{p}|) \cdot \{ (c_0, \ldots, c_n). c_0 \mapsto \vec{p}_0 \star \ldots \star c_n \mapsto \vec{p}_n \}\]
Protocol Consistency

For any synchronised exchange from $i$ to $j$, given the binders of $i$, we must:

1. Instantiate the binders of $j$
2. Prove equality of exchanged values
3. Prove protocol consistency where $i$ and $j$ are updated to their respective tails

Repeat until no more synchronised exchanges exist.

$$(\forall i, j. \text{semantic\_dual } \vec{p} ij)$$

$\text{CONSISTENT } \vec{p}$$

$$\vec{p}_i = ![j](\vec{x}_1 : \vec{\tau}_1) \langle v_1 \rangle \cdot p_1 \leftarrow \vec{p}_j = ?[i](\vec{x}_2 : \vec{\tau}_2) \langle v_2 \rangle \cdot p_2 \leftarrow$$

$\forall \vec{x}_1 : \vec{\tau}_1. \exists \vec{x}_2 : \vec{\tau}_2. v_1 = v_2 \triangleright (\text{CONSISTENT } (\vec{p}[i := p_1][j := p_2]))$$

$\text{semantic\_dual } \vec{p} ij$
Protocol Consistency - Example

Protocol consistency example:

\[
\vec{p}_0 := \text{!}[1] (x : \mathbb{Z}) \langle x \rangle . \text{?}[2] (x + 2) . \text{end}
\]

\[
\vec{p}_1 := \text{?}[0] (x : \mathbb{Z}) \langle x \rangle . \text{!}[2] (x + 1) . \text{end}
\]

\[
\vec{p}_2 := \text{?}[1] (x : \mathbb{Z}) \langle x \rangle . \text{!}[0] (x + 1) . \text{end}
\]

Protocol consistency:

\[
(\forall i,j. \text{semantic\_dual} \vec{p} ij)
\]

\[
\text{CONSISTENT} \vec{p}
\]

\[
\vec{p}_i = \text{!}[j] (x_1 : \tau_1) \langle v_1 \rangle . p_1 \rightarrow* \vec{p}_j = \text{?}[i] (x_2 : \tau_2) \langle v_2 \rangle . p_2 \rightarrow*
\]

\[
\forall x_1 : \tau_1 . \exists x_2 : \tau_2 . v_1 = v_2 \rightarrow (\text{CONSISTENT} (\vec{p}[i := p_1][j := p_2]))
\]

\[
\text{semantic\_dual} \vec{p} ij
\]
Roundtrip Example - Verified

Roundtrip program:

```ocaml
let (c₀, c₁, c₂) = new_chan(3) in
fork {let x = c₁[0].recv() in c₁[2].send(x + 1)};
fork {let x = c₂[1].recv() in c₂[0].send(x + 1)};

let x = c₀[2].recv() in assert(x = 42)
```

Protocols:

```
c₀ ➞ ![1] (x : ℤ) ⟨x⟩.?[2] ⟨x + 2⟩. end


c₁ ➞ ?[0] (x : ℤ) ⟨x⟩.![2] ⟨x + 1⟩. end


c₂ ➞ ?[1] (x : ℤ) ⟨x⟩.![0] ⟨x + 1⟩. end
```

Verified Safety!
Roundtrip Reference Example

Roundtrip reference program:

```plaintext
let (c0, c1, c2) = new-chan(3) in
fork {let ℓ = c1[0].recv() in ℓ ↦ (! ℓ + 1); c1[2].send(ℓ)};
fork {let ℓ = c2[1].recv() in ℓ ↦ (! ℓ + 1); c2[0].send()};
let ℓ = ref 40 in c0[1].send(ℓ); c0[2].recv(); let x = ℓ in assert(x = 42)
```

Goal: Prove crash-freedom (safety) and verify asserts (functional correctness)
Multiparty Actris with Resources

Protocols: $![i] (\vec{x} : \vec{\tau}) \langle v \rangle \{P\}.p \mid ?[i] (\vec{x} : \vec{\tau}) \langle v \rangle \{P\}.p$

Example: $![1] (\ell : \text{Loc}, x : \mathbb{Z}) \langle \ell \rangle \{\ell \mapsto x\}. ?[2] () \langle () \rangle \{\ell \mapsto (x + 2)\}. \text{end}$

Rules:

$H_t$-send

$$\{ c \mapsto ![i] (\vec{x} : \vec{\tau}) \langle v \rangle \{P\}.p \ast P[\vec{t}/\vec{x}] \} \ c[i].\text{send}(v[\vec{t}/\vec{x}]) \ \{ c \mapsto p[\vec{t}/\vec{x}] \}$$

$H_t$-recv

$$\{ c \mapsto ?[i] (\vec{x} : \vec{\tau}) \langle v \rangle \{P\}.p \} \ c[i].\text{recv}() \ \{ w. \exists \vec{t}. \ w = v[\vec{t}/\vec{x}] \ast c \mapsto p[\vec{t}/\vec{x}] \ast P[\vec{t}/\vec{x}] \}$$

$H_t$-new

$$\{ \text{consistent } \vec{p} \ast |\vec{p}| = n + 1 \} \ \text{new\_chan}(|\vec{p}|) \ \{(c_0, \ldots, c_n).c_0 \mapsto \vec{p}_0 \ast \ldots \ast c_n \mapsto \vec{p}_n \}$$
Protocol Consistency with Resources

For any synchronised exchange from $i$ to $j$, given the binders and resources of $i$:

1. Instantiate the binders of $j$
2. Prove equality of exchanged values and the resources of $j$
3. Prove protocol consistency where $i$ and $j$ are updated to their respective tails

Repeat until no more synchronised exchanges exist.

\[ (\forall i,j. \text{semantic\_dual } \vec{p}_{ij}) \]

\[ \text{CONSISTENT } \vec{p} \]

\[ \vec{p}_i = ! [j] (x_1 : \vec{\tau}_1) \langle v_1 \rangle \{ P_1 \} . p_1 \rightarrow \vec{p}_j = ? [i] (x_2 : \vec{\tau}_2) \langle v_2 \rangle \{ P_2 \} . p \rightarrow \]

\[ \forall x_2 : \vec{\tau}_2. P_2 \rightarrow \exists x_1 : \vec{\tau}_1. v_1 = v_2 \cdot P_2 \cdot \triangleright (\text{CONSISTENT } (\vec{p}[i := p_1][j := p_2])) \]

\[ \text{semantic\_dual } \vec{p}_{ij} \]
Protocol Consistency with Resources - Example

Protocol consistency example:

\[ \vec{p}_0 := ! [1] (\ell : \text{Loc}, x : \mathbb{Z}) \langle \ell \rangle \{ \ell \mapsto x \}. ?[2] \langle () \rangle \{ \ell \mapsto (x + 2) \}. \text{end} \]
\[ \vec{p}_1 := ?[0] (\ell : \text{Loc}, x : \mathbb{Z}) \langle \ell \rangle \{ \ell \mapsto x \}. ! [2] \langle \ell \rangle \{ \ell \mapsto (x + 1) \}. \text{end} \]
\[ \vec{p}_2 := ?[1] (\ell : \text{Loc}, x : \mathbb{Z}) \langle \ell \rangle \{ \ell \mapsto x \}. ! [0] \langle () \rangle \{ \ell \mapsto (x + 1) \}. \text{end} \]

Protocol consistency:

\[
\begin{align*}
(\forall i,j. \text{semantic\_dual } \vec{p} i j) \\
\text{CONSISTENT } \vec{p}
\end{align*}
\]

\[
\begin{align*}
\vec{p}_i = ! [j] (\vec{x}_1 : \vec{\tau}_1) \langle v_1 \rangle \{ P_1 \}. p_1 \rightarrow* \vec{p}_j = ?[i] (\vec{x}_2 : \vec{\tau}_2) \langle v_2 \rangle \{ P_2 \}. p_2 \rightarrow* \\
\forall \vec{x}_1 : \vec{\tau}_1. P_1 \rightarrow* \exists \vec{x}_2 : \vec{\tau}_2. v_1 = v_2 \rightarrow* (\text{CONSISTENT } (\vec{p}[i := p_1][j := p_2]))
\end{align*}
\]

\[
\begin{align*}
\text{semantic\_dual } \vec{p} i j
\end{align*}
\]
Roundtrip Reference Example - Verified

Roundtrip reference program:

\[
\text{let } (c_0, c_1, c_2) = \text{new_chan}(3) \text{ in } \\
\text{fork \{let } \ell = c_1[0].\text{recv()} \text{ in } \ell \leftarrow (!\ell + 1); c_1[2].\text{send}(\ell)\}; \\
\text{fork \{let } \ell = c_2[1].\text{recv()} \text{ in } \ell \leftarrow (!\ell + 1); c_2[0].\text{send}()\}; \\
\text{let } \ell = \text{ref} 40 \text{ in } c_0[1].\text{send}(\ell); c_0[2].\text{recv}(); \text{let } x = !\ell \text{ in } \text{assert}(x = 42)
\]

Protocols:

- \(c_0 \leftrightarrow ![1] (\ell : \text{Loc}, x : \mathbb{Z}) \langle \ell \rangle \{ \ell \mapsto x \}. ?[2] \langle () \rangle \{ \ell \mapsto (x + 2) \}. \text{end}\)
- \(c_1 \leftrightarrow \ ?[0] (\ell : \text{Loc}, x : \mathbb{Z}) \langle \ell \rangle \{ \ell \mapsto x \}. ![2] \langle \ell \rangle \{ \ell \mapsto (x + 1) \}. \text{end}\)
- \(c_2 \leftrightarrow \ ?[1] (\ell : \text{Loc}, x : \mathbb{Z}) \langle \ell \rangle \{ \ell \mapsto x \}. ![0] \langle () \rangle \{ \ell \mapsto (x + 1) \}. \text{end}\)
Protocol Consistency - Recursion

Protocols are contractive in the tail:

\( \mu \text{rec}. ![1] (\ell : \text{Loc}, x : \mathbb{Z}) \langle \ell \rangle \{ \ell \mapsto x \}. ?[2] \langle () \} \{ \ell \mapsto (x + 2) \}. \text{rec} \)

Protocols:

\[
\begin{align*}
\vec{p}_0 &= \mu \text{rec}. ![1] (\ell : \text{Loc}, x : \mathbb{Z}) \langle \ell \rangle \{ \ell \mapsto x \}. ?[2] \langle () \} \{ \ell \mapsto (x + 2) \}. \text{rec} \\
\vec{p}_1 &= \mu \text{rec}. ?[0] (\ell : \text{Loc}, x : \mathbb{Z}) \langle \ell \rangle \{ \ell \mapsto x \}. ![2] \langle \ell \} \{ \ell \mapsto (x + 1) \}. \text{rec} \\
\vec{p}_2 &= \mu \text{rec}. ?[1] (\ell : \text{Loc}, x : \mathbb{Z}) \langle \ell \rangle \{ \ell \mapsto x \}. ![0] \langle () \} \{ \ell \mapsto (x + 1) \}. \text{rec}
\end{align*}
\]

Recursion via Löb induction (▷):

\[
\begin{align*}
\vec{p}_i &= ![j] (\vec{x}_1 : \vec{\tau}_1) \langle \vec{v}_1 \} \{ P_1 \}. P_1 \dashv \vdash \vec{p}_j = ?[i] (\vec{x}_2 : \vec{\tau}_2) \langle \vec{v}_2 \} \{ P_2 \}. P_2 \dashv \vdash \\
\forall \vec{x}_1 : \vec{\tau}_1. P_1 \dashv \vdash \exists \vec{x}_2 : \vec{\tau}_2. \vec{v}_1 = \vec{v}_2 \iff P_2 \dashv \vdash (\text{consistent} (\vec{p}[i := p_1][j := p_2]))
\end{align*}
\]

\text{semantic\_dual \vec{p}_i j}
Protocol Consistency - Framing

Consider the replacement of process 1 with a forwarder:

\[
\text{let } v = c_1[0].\texttt{recv()} \text{ in } c_1[1].\texttt{send}(v)
\]

Protocols:

\[
\vec{p}_0 = \mu \texttt{rec. } ![1] (\ell : \text{Loc}, x : \mathbb{Z}) \langle \ell \rangle \{ \ell \mapsto x \}. ?[2] \langle () \rangle \{ \ell \mapsto (x + 1) \}. \texttt{rec}
\]
\[
\vec{p}_1 = \mu \texttt{rec. } ?[0] (v : \text{Val}) \langle v \rangle . ! [2] \langle v \rangle . \texttt{rec}
\]
\[
\vec{p}_2 = \mu \texttt{rec. } ?[1] (\ell : \text{Loc}, x : \mathbb{Z}) \langle \ell \rangle \{ \ell \mapsto x \}. ! [0] \langle () \rangle \{ \ell \mapsto (x + 1) \}. \texttt{rec}
\]

Protocol consistency owns resources while in transit:

\[
\vec{p}_i = ! [j] (x^i : \tau^i_1) \langle v_1 \rangle \{ P_1 \}. p_1 \rightarrow \vec{p}_j = ?[i] (x^i : \tau^i_2) \langle v_2 \rangle \{ P_2 \}. p_2 \rightarrow
\]
\[
\forall x^i_1 : \tau^i_1. P_1 \rightarrow \exists x^i_2 : \tau^i_2. v_1 = v_2 \rightarrow P_2 \rightarrow \triangleright (\text{CONSISTENT } (\vec{p}[i := p_1][j := p_2]))
\]

\[\text{semantic dual } \vec{p}_{ij}\]
Consider the extension of process 1 with a rerouter:

```
let (v, b) = c_1[0].recv() in c_1[if b then 2 else 3].send(v)
```

Protocols:

\[
\vec{p}_0 = \mu \text{rec. } ![1](\ell : \text{Loc}, x : \mathbb{Z}, b : \mathbb{B}) \langle (\ell, b) \rangle \{\ell \mapsto x\}. rec \\
\vec{p}_1 = \mu \text{rec. } ?[0](v : \text{Val}, b : \mathbb{B}) \langle (v, b) \rangle . ! [\text{if } b \text{ then } 2 \text{ else } 3] \langle v \rangle . rec \\
\vec{p}_2, \vec{p}_3 = \mu \text{rec. } ![1](\ell : \text{Loc}, x : \mathbb{Z}) \langle \ell \rangle \{\ell \mapsto x\}. ! [0] \langle () \rangle \{\ell \mapsto (x + 1)\}. rec
\]

We can do case analysis on the binders:

\[
\vec{p}_i = ![j] (x'_1 : \tau_1) \langle v_1 \rangle \{P_1\}. p_1 \ast \vec{p}_j = ?[i] (x'_2 : \tau_2) \langle v_2 \rangle \{P_2\}. p_2 \ast \\
\forall x'_1 : \tau_1. P_1 \ast \exists x'_2 : \tau_2. v_1 = v_2 \ast P_2 \ast \triangleright (\text{CONSISTENT } (\vec{p}[i := p_1][j := p_2])) \\
\]

\[\text{semantic_dual } \vec{p}_{ij}\]
Benchmark:
Chang and Roberts
Ring Leader Election
Leader Election

Consider $n$ uniquely identifiable actors in a network. Leader election is an algorithm that upon satisfies:

- **Uniqueness**: There is exactly one actor that considers itself as leader.
- **Agreement**: All other actors know who the leader is.
- **Termination**: The algorithm finishes in finite time*.

**Goal**: Prove uniqueness and agreement.

**Observation**: We prove partial correctness so termination is out of scope.

We lift the properties to functional correctness as:

- **Uniqueness**: The leader can proceed with elevated permissions (resources).
- **Agreement**: Participants following interaction can depend on knowing leader.
Chang and Roberts Ring Leader Election - Overview
Chang and Roberts Ring Leader Election - Algorithm

Consider $n$ actors, with unique id’s, arranged in a ring

- Ex1: $0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 0$
- Ex2: $0 \rightarrow 2, 2 \rightarrow 1, 1 \rightarrow 0$

Actors are tagged as participating or not; everyone starts untagged

- Tag as participating whenever any message is sent

Message types are election($i'$) (1) and elected($i'$) (2)

Received election($i'$) messages are compared to the receivers id $i$ and

- If $i' > i$, send election($i'$) (1.1)
- If $i' = i$, we are elected, send elected($i$) (1.2)
- If we are not participating, send election($i$) (1.3)
- If we are already participating, do nothing (1.4)

Received elected($i'$) messages are compared to the participants id $i$ and

- If $i' = i$, terminate by returning $i'$ (2.1)
- If $i' \neq i$, send elected($i'$), and terminate by returning $i'$ (2.2)
We encode election\((i)\) as \textbf{inl} \(i\) and elected\((i)\) as \textbf{inr} \(i\).

We write \(i_l\) and \(i_r\) for the left and right participants of participant \(i\).

The leader election process can then be implemented as follows:

\[
\text{process } c_i \triangleq \text{rec } \text{rec isp } = \\
\text{match } c[i_r].\text{recv}() \text{ with } \\
| \text{inl} \ i' \Rightarrow \begin{cases} 
\text{if } i < i' \text{ then } c[i].\text{send(inl} \ i'); \text{rec true} & (1.1) \\
\text{else if } i = i' \text{ then } c[i].\text{send(inr} \ i); \text{rec false} & (1.2) \\
\text{else if } \text{isp} \text{ then } \text{rec true} & (1.3) \\
\text{else } c[i].\text{send(inl} \ i); \text{rec true} & (1.4) \\
\end{cases} \\
| \text{inr} \ i' \Rightarrow \begin{cases} 
\text{if } i = i' \text{ then } i' & (2.1) \\
\text{else } c[i].\text{send(inr} \ i'); i' & (2.2) \\
\end{cases} \\
\text{end}
\]
Chang and Roberts Ring Leader Election - Validation

Procedure for starting the election:

\[
\text{init } c_i \triangleq c[i].\text{send(inl } i\text{); process } c_i \text{ true}
\]

Closed program example of election:

\[
\text{ring_ref_prog } n \triangleq \\
\text{let } \ell = \text{ref } 42 \text{ in} \\
\text{let } (c_0, \ldots, c_{n-1}) = \text{new_chan}(n) \text{ in} \\
\text{for } (i = 1 \ldots (n - 1)) \left\{ \text{fork } \left\{ \begin{array}{l}
\text{let } i' = \text{process } c_i \text{ i false in} \\
\text{if } i' = i \text{ then free } \ell \text{ else } ()
\end{array} \right\} \right\} ; \\
\text{let } i' = \text{init } c_0 \text{ 0 in if } i' = 0 \text{ then free } \ell \text{ else } ()
\]

Goal: Verify that only one leader is elected (no use-after-free)
Chang and Roberts Ring Leader Election - Protocol

We can define the ring leader election protocol as:

\[
\text{ring}_\text{prot}(i : \mathbb{N})(P : \text{iProp})(p : \mathbb{N} \rightarrow \text{iProto}) : \mathbb{B} \rightarrow \text{iProto} \overset{\Delta}{=} \mu \text{rec}. \lambda (isp : \mathbb{B}).
\]

\[
\begin{align*}
\text{inl}(i' : \mathbb{N}) & \langle i' \rangle \quad \Rightarrow \begin{cases} 
\begin{align*}
\text{if } i < i' & \Rightarrow ![i] \langle \text{inl } i' \rangle . \text{rec } \text{true} \\
\text{else if } i = i' & \Rightarrow ![i] \langle \text{inr } i \rangle . \text{rec } \text{false} \\
\text{else if } isp & \Rightarrow \text{rec } \text{true}
\end{cases} \\
\end{align*}
\end{cases}
\]

\&[i_r]

\[
\begin{align*}
\text{inr}(i' : \mathbb{N}) & \langle i' \rangle \{ i = i' \Rightarrow P \} \Rightarrow \begin{cases} 
\begin{align*}
\text{if } i = i' & \Rightarrow p i' \\
\text{else } & \Rightarrow ![i] \langle \text{inr } i' \rangle . p i'
\end{cases}
\end{cases}
\end{align*}
\]

This lets us verify the following spec for the ring leader process:

\[
\{ c \mapsto \text{ring}_\text{prot } i \ P \ p \ isp \} \text{ process } c \ i \ isp \ \{ i' . c \mapsto (p \ i') \ast (i = i' \Rightarrow P) \}
\]
The protocol for starting an election is an extension of the ring protocol:

\[
\text{init}_\text{prot}(i : \mathbb{N})(P : \text{iProp})(p : \mathbb{N} \rightarrow \text{iProto}) : \text{iProto} \triangleq \\
! [i] (\text{inl } i) \{ P \}. \text{ring}_\text{prot} i P p \text{ true}
\]

With the initial message we yield the \( P \) resource to the network.

With this protocol we can prove the following specification for the starting process:

\[
\{ c \mapsto (\text{init}_\text{prot } i P p) \ast P \} \text{ init } c \ i \ I \ c \mapsto (p \ i') \ast (i = i' \Rightarrow P)\}
\]
We verify the program for 3 participants with the following protocols:

\[
\begin{align*}
c_0 & \to \text{end} \\
c_1 & \to \text{end} \\
c_2 & \to \text{end}
\end{align*}
\]

We can thus verify: \{True\} ring_ref_prog 3 \{True\}
ring_del_prog \( n \triangleq \)

\[
\begin{align*}
\text{let } (c_0, \ldots, c_n) &= \text{new\_chan}(n + 1) \text{ in} \\
\text{fork } \{ \text{let } i' = c_n[0].\text{recv()} \text{ in for}(i = 1 \ldots (n - 1)) \{ \text{assert}(c_n[i].\text{recv()} = i') \} \} ; \\
\text{for}(i = 1 \ldots (n - 1)) \{ \text{fork } \{ \text{let } i' = \text{process } c_i \text{ i false in } c_i[n].\text{send}(i') \} \} ; \\
\text{let } i' = \text{init } c_0 0 \text{ in } c_0[n].\text{send}(i')
\end{align*}
\]

We verify the program for 3 participants and 1 central coordinator:

\[
\begin{align*}
c_0 &\rightarrow \text{ end} \\
c_1 &\rightarrow \text{ end} \\
c_2 &\rightarrow \text{ end} \\
c_3 &\rightarrow \text{ end}
\end{align*}
\]

We can thus verify: \{True\} ring_del_prog 3 \{True\}
Language Parametricity of Multiparty Actris
Multiparty Actris Ghost Theory

We prove language-generic ghost theory rules:

**PROTO-ALLOC**

\[
\text{CONSISTENT } \vec{p} \quad \text{\Rightarrow } \exists \chi \cdot \text{prot_ctx } \chi \mid \vec{p} \mid \ast \quad \text{prot_own } \chi \ i \ p \\
i \mapsto p \in \vec{p}
\]

**PROTO-VALID**

\[
\text{prot_ctx } \chi \ n \quad \text{prot_own } \chi \ i \ p \\
i < n
\]

**PROTO-STEP**

\[
\text{prot_ctx } \chi \ n \quad P_1[\vec{t}_1/\vec{x}_1] \\
\text{prot_own } \chi \ i \ (\land j \] (\vec{x}_1 : \tau_1) \langle v_1 \rangle \{P_1 \}, p_1) \quad \text{prot_own } \chi \ j \ (\land i \] (\vec{x}_2 : \tau_2) \langle v_2 \rangle \{P_2 \}, p_2)
\]

\[
\text{\Rightarrow } \exists (\vec{t}_2 : \tau_2). \text{prot_ctx } \chi \ast \text{prot_own } \chi \ i \ (p_1[\vec{t}_1/\vec{x}_1]) \ast \text{prot_own } \chi \ j \ (p_2[\vec{t}_2/\vec{x}_2]) \ast (v_1[\vec{t}_1/\vec{x}_1]) = (v_2[\vec{t}_2/\vec{x}_2]) \ast P_2[\vec{t}_2/\vec{x}_2]
\]

One can then define \( c \mapsto p \) and prove Hoare triple rules for a given language using the ghost theory

- Such as \( H_t\text{-send} \), \( H_t\text{-recv} \), and \( H_t\text{-new} \)
Conclusion and Future Work
Conclusion

Dependent multiparty protocols are non-trivial to prove sound
- Mismatched dependencies (quantifiers) makes syntactic analysis difficult
- Fullfillment of received resources is tricky

Concurrent separation logic (Iris) is a good fit for multiparty protocols
- Quantifier scopes enable inherent tracking of dependencies
- Separation logic enables framing of resources
- Integration with other features readily available

Automation of protocol consistency proofs is warranted
- Deterministic (often synchronous) protocols are barely manageable
- Brute-force procedure allows for some automation
- Asynchronous protocols would require more efficient techniques
Future Work

Additional features
- Asynchronous communication

More scalable methodology for proving protocol consistency
- Abstraction and Modularity via separation logic
- Automation via model checking?

Semantic Multiparty Session Type System
- Investigate correspondences with syntactic protocol consistency

Deadlock freedom guarantees
- Leverage connectivity graphs for multiparty communication

Multiparty Actris for distributed systems
- Leverage Aneris

And much more?: RefinedActris, Verified Secure MPC, Non-interference, ...
![1] (“Thank you”) {MultrisOverview}.

\( \mu \text{rec.} \) ![1] (q : Question i) ⟨q⟩ {AboutMultris q}.

![i] (a : Answer) ⟨a⟩ {Insightful q a}.