Multris:
Functional Verification of Multiparty Message Passing in Separation Logic

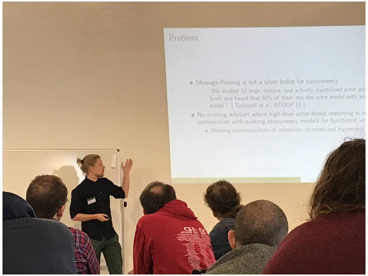
Jonas Kastberg Hinrichsen
Aarhus University

Jules Jacobs
Cornell University

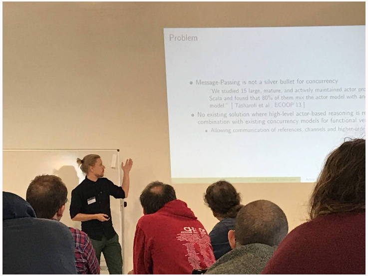
Robbert Krebbers
Radboud University Nijmegen

Jesper Bengtson Daniël Louwrink Léon Gondelman
Mário Pereira Amin Timany Lars Birkedal

Me, Actris, and The Iris Workshop



Me, Actris, and The Iris Workshop



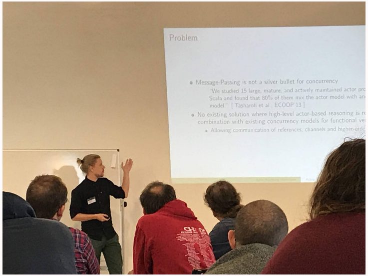
Problem

- Message-Passing is not a silver bullet for concurrency

  "We studied 15 large, mature, and actively maintained actor pr
  Scala and found that 80% of them mix the actor model with an
  model." [Tasharofi et al., ECOOP'13]
- No existing solution where high-level actor-based reasoning is r
  combination with existing concurrency models for functional ve
  - Allowing communication of references, channels and higher-orde

[POPL'20] Actris, *1st Iris Workshop*

Me, Actris, and The Iris Workshop



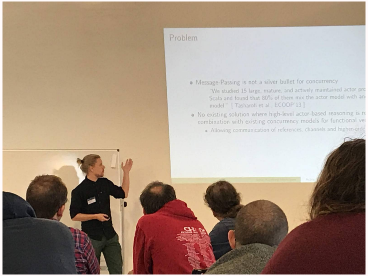
Problem

- Message-Passing is not a silver bullet for concurrency

  "We studied 15 large, mature, and actively maintained actor pro
  Scala and found that 80% of them mix the actor model with an
  model." [Tasharofi et al., ECOOP'13]
- No existing solution where high-level actor-based reasoning is re
  combination with existing concurrency models for functional ve
  - Allowing communication of references, channels and higher-orde

[POPL'20] Actris, *1st Iris Workshop*
[CPP'21] Semantic Session Types

Me, Actris, and The Iris Workshop



Problem

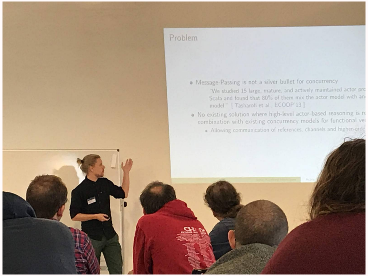
- Message-Passing is not a silver bullet for concurrency

  "We studied 15 large, mature, and actively maintained actor
  Scala and found that 80% of them mix the actor model with
  model." [Tasharofi et al., ECOOP'13]

- No existing solution where high-level actor-based reasoning is
  combination with existing concurrency models for functional
  - Allowing communication of references, channels and

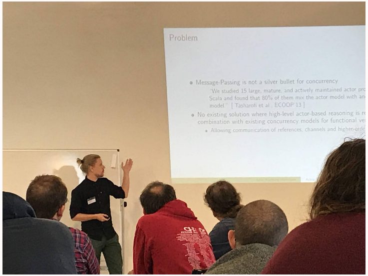
[POPL'20] Actris, *1st Iris Workshop*
[CPP'21] Semantic Session Types
[LMCS'22] Actris 2.0, *2nd Iris Workshop*

Me, Actris, and The Iris Workshop



[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)*

Me, Actris, and The Iris Workshop



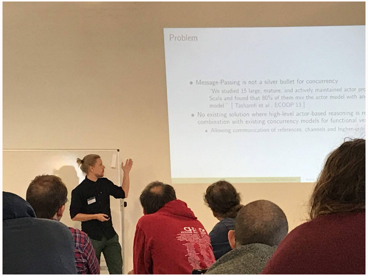
Problem

- Message-Passing is not a silver bullet for concurrency

  "We studied 15 large, mature, and actively maintained actor pro
  Scala and found that 80% of them mix the actor model with an
  model." [Tasharofi et al., ECOOP'13]
- No existing solution where high-level actor-based reasoning is re
  combination with existing concurrency models for functional ve
  - Allowing communication of references, channels and higher-orde

[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)*

Me, Actris, and The Iris Workshop



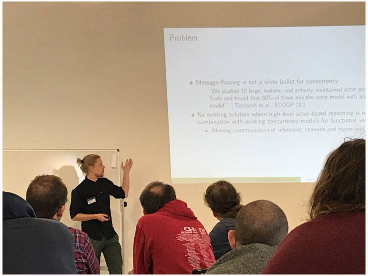
Problem

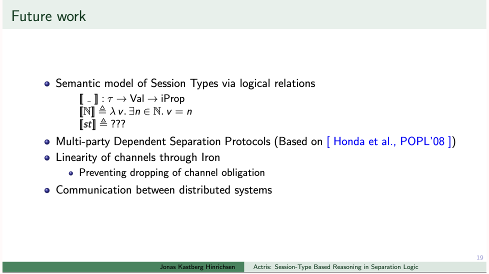
- Message-Passing is not a silver bullet for concurrency

  "We studied 15 large, mature, and actively maintained actor pro
  Scala and found that 80% of them mix the actor model with an
  model." [Tasharofi et al., ECOOP'13]
- No existing solution where high-level actor-based reasoning is re
  combination with existing concurrency models for functional ve
  - Allowing communication of references, channels and higher-orde

[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

Me, Actris, and The Iris Workshop





Future work

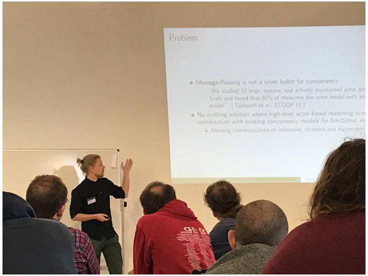
- Semantic model of Session Types via logical relations
  $[\![\_]\!] : \tau \to \mathsf{Val} \to \mathsf{iProp}$
  $[\![\mathbb{N}]\!] \triangleq \lambda\, v.\, \exists n \in \mathbb{N}.\, v = n$
  $[\![st]\!] \triangleq ???$
- Multi-party Dependent Separation Protocols (Based on [Honda et al., POPL'08])
- Linearity of channels through Iron
  - Preventing dropping of channel obligation
- Communication between distributed systems

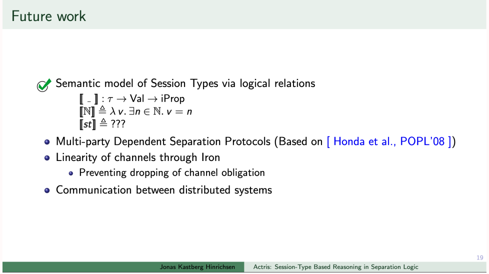
19

Jonas Kastberg Hinrichsen | Actris: Session-Type Based Reasoning in Separation Logic

[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

Me, Actris, and The Iris Workshop





Future work

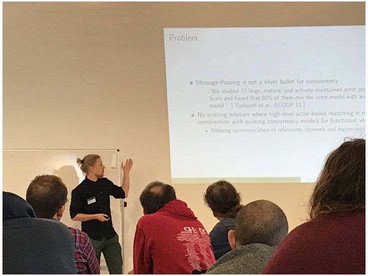
- ✓ Semantic model of Session Types via logical relations
  - $[\![\_]\!] : \tau \rightarrow \mathsf{Val} \rightarrow \mathsf{iProp}$
  - $[\![\mathbb{N}]\!] \triangleq \lambda\, v.\, \exists n \in \mathbb{N}.\, v = n$
  - $[\![st]\!] \triangleq ???$
- Multi-party Dependent Separation Protocols (Based on [Honda et al., POPL'08])
- Linearity of channels through Iron
  - Preventing dropping of channel obligation
- Communication between distributed systems

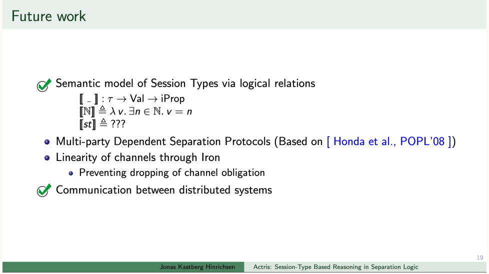
19

Jonas Kastberg Hinrichsen | Actris: Session-Type Based Reasoning in Separation Logic

[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

Me, Actris, and The Iris Workshop




Future work

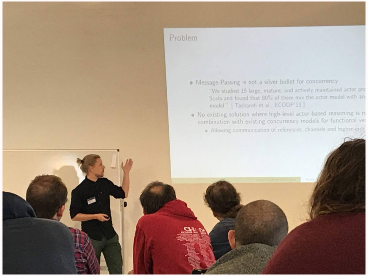
- ✓ Semantic model of Session Types via logical relations
  - $[\![\_]\!] : \tau \to \mathsf{Val} \to \mathsf{iProp}$
  - $[\![\mathbb{N}]\!] \triangleq \lambda\, v.\, \exists n \in \mathbb{N}.\, v = n$
  - $[\![st]\!] \triangleq ???$
- Multi-party Dependent Separation Protocols (Based on [Honda et al., POPL'08])
- Linearity of channels through Iron
  - Preventing dropping of channel obligation
- ✓ Communication between distributed systems

19

Jonas Kastberg Hinrichsen Actris: Session-Type Based Reasoning in Separation Logic

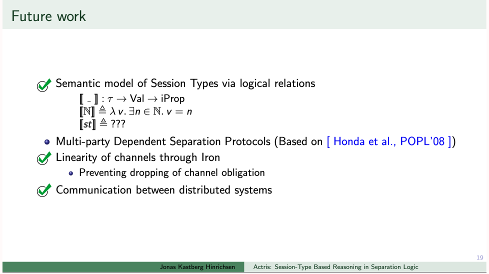
[POPL'20] Actris, *1st Iris Workshop*
[CPP'21] Semantic Session Types
[LMCS'22] Actris 2.0, *2nd Iris Workshop*
[ICFP'23a] Actris in Distributed Systems, *2nd/3rd Iris Workshop (Léon/Me)*
[ICFP'23b] MiniActris, *3rd Iris Workshop (Jules)*
[POPL'24] LinearActris

Me, Actris, and The Iris Workshop



Problem

- Message-Passing is not a silver bullet for concurrency
  "We studied 15 large, mature, and actively maintained actor pro
  Scala and found that 80% of them mix the actor model with an
  model." [Tasharofi et al., ECOOP'13]
- No existing solution where high-level actor-based reasoning is re
  combination with existing concurrency models for functional ve
  - Allowing communication of references, channels and higher-orde



Future work

- ✓ Semantic model of Session Types via logical relations
  $[\![\_]\!] : \tau \rightarrow \mathsf{Val} \rightarrow \mathsf{iProp}$
  $[\![\mathbb{N}]\!] \triangleq \lambda\, v.\, \exists n \in \mathbb{N}.\, v = n$
  $[\![st]\!] \triangleq ???$
- 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

19

Jonas Kastberg Hinrichsen | Actris: Session-Type Based Reasoning in Separation Logic

[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

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

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

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

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

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

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

Multiparty Message Passing

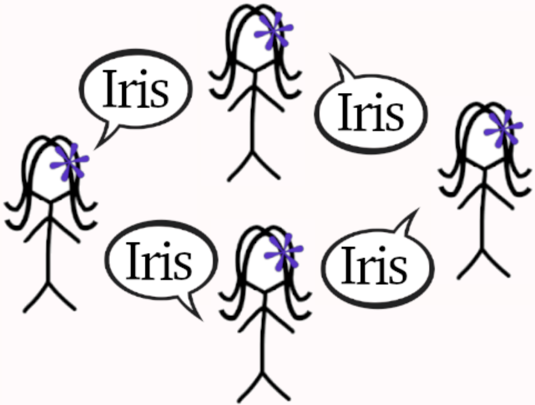
Multiparty message passing

- Message passing with dependent interactions between multiple actors
- Like a game of telephone!

Multiparty Message Passing

Multiparty message passing

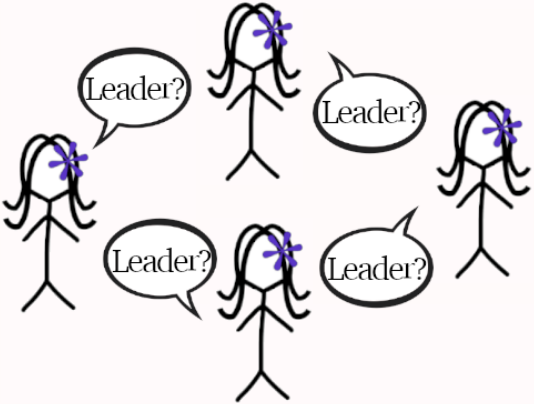
- Message passing with dependent interactions between multiple actors
- Like a game of telephone!



Multiparty Message Passing

Multiparty message passing

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



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

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)

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:

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

Multiparty Message Passing in Shared Memory

Multiparty channels in shared memory:

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

Example Program: Roundtrip

$$
\begin{array}{l}
\texttt{let}\,(c_0, c_1, c_2) = \texttt{new\_chan}(3)\;\texttt{in} \\
\texttt{fork}\;\{\texttt{let}\,x = c_1[0].\texttt{recv}()\;\texttt{in}\;c_1[2].\texttt{send}(x+1)\}\,; \\
\texttt{fork}\;\{\texttt{let}\,x = c_2[1].\texttt{recv}()\;\texttt{in}\;c_2[0].\texttt{send}(x+1)\}\,; \\
c_0[1].\texttt{send}(40);\,\texttt{let}\,x = c_0[2].\texttt{recv}()\;\texttt{in}\;\texttt{assert}(x = 42)
\end{array}
$$

Safety and Functional Correctness

Example Program: Roundtrip

$$\begin{array}{l}
\mathtt{let}\,(c_0, c_1, c_2) = \mathtt{new\_chan}(3)\,\mathtt{in} \\
\mathtt{fork}\ \{\mathtt{let}\,x = c_1[0].\mathtt{recv}()\,\mathtt{in}\,c_1[2].\mathtt{send}(x+1)\}\,; \\
\mathtt{fork}\ \{\mathtt{let}\,x = c_2[1].\mathtt{recv}()\,\mathtt{in}\,c_2[0].\mathtt{send}(x+1)\}\,; \\
c_0[1].\mathtt{send}(40);\,\mathtt{let}\,x = c_0[2].\mathtt{recv}()\,\mathtt{in}\,\mathtt{assert}(x = 42)
\end{array}$$

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

Safety and Functional Correctness

Example Program: Roundtrip

$$\begin{array}{l}
\mathbf{let}\,(c_0, c_1, c_2) = \mathbf{new\_chan}(3)\;\mathbf{in} \\
\mathbf{fork}\;\{\mathbf{let}\,x = c_1[0].\mathbf{recv}()\;\mathbf{in}\;c_1[2].\mathbf{send}(x+1)\}\,; \\
\mathbf{fork}\;\{\mathbf{let}\,x = c_2[1].\mathbf{recv}()\;\mathbf{in}\;c_2[0].\mathbf{send}(x+1)\}\,; \\
c_0[1].\mathbf{send}(40);\,\mathbf{let}\,x = c_0[2].\mathbf{recv}()\;\mathbf{in}\;\mathbf{assert}(x = 42)
\end{array}$$

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

Safety	Functional Correctness
Type systems	Program logics

Safety and Functional Correctness

Example Program: Roundtrip

$$\begin{array}{l}
\mathtt{let}\,(c_0, c_1, c_2) = \mathtt{new\_chan}(3)\ \mathtt{in} \\
\mathtt{fork}\ \{\mathtt{let}\, x = c_1[0].\mathtt{recv}()\ \mathtt{in}\, c_1[2].\mathtt{send}(x+1)\}\,; \\
\mathtt{fork}\ \{\mathtt{let}\, x = c_2[1].\mathtt{recv}()\ \mathtt{in}\, c_2[0].\mathtt{send}(x+1)\}\,; \\
c_0[1].\mathtt{send}(40);\ \mathtt{let}\, x = c_0[2].\mathtt{recv}()\ \mathtt{in}\ \mathtt{assert}(x = 42)
\end{array}$$

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

Safety	**Functional Correctness**
Type systems	Program logics
Multiparty session types	???

Safety and Functional Correctness

Example Program: Roundtrip

$$\begin{array}{l}
\mathtt{let}\ (c_0, c_1, c_2) = \mathtt{new\_chan}(3)\ \mathtt{in} \\
\mathtt{fork}\ \{\mathtt{let}\ x = c_1[0].\mathtt{recv}()\ \mathtt{in}\ c_1[2].\mathtt{send}(x+1)\}\,; \\
\mathtt{fork}\ \{\mathtt{let}\ x = c_2[1].\mathtt{recv}()\ \mathtt{in}\ c_2[0].\mathtt{send}(x+1)\}\,; \\
c_0[1].\mathtt{send}(40);\ \mathtt{let}\ x = c_0[2].\mathtt{recv}()\ \mathtt{in}\ \mathtt{assert}(x = 42)
\end{array}$$

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

Safety	**Functional Correctness**
Type systems	Program logics
Multiparty session types	???
$c_0 : \mathbf{!}[1]\mathbb{Z}.\,\mathbf{?}[2]\mathbb{Z}.\,\mathbf{end}$ $c_1 : \mathbf{?}[0]\mathbb{Z}.\,\mathbf{!}[2]\mathbb{Z}.\,\mathbf{end}$ $c_2 : \mathbf{?}[1]\mathbb{Z}.\,\mathbf{!}[0]\mathbb{Z}.\,\mathbf{end}$	???

! is send, **?** is receive

Key Idea

Prior Work: Binary protocols

- **Session Types:** $!\mathbb{Z}.\,?\mathbb{Z}.\,\textbf{end}$
- **Actris protocols:** $!\,\langle 40\rangle.\,?\langle 42\rangle.\,\textbf{end}$

Key Idea

Prior Work: Binary protocols

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

Key Idea

Prior Work: Binary protocols

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

Key Idea: Multiparty protocols!

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

Key Idea

Prior Work: Binary protocols

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

Key Idea: Multiparty protocols!

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

Example Program: Roundtrip

$$c_0[1].\texttt{send}(40);\, \texttt{let}\, x = c_0[2].\texttt{recv}()\, \texttt{in}\, \texttt{assert}(x = 42)$$

Key Idea

Prior Work: Binary protocols

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

Key Idea: Multiparty protocols!

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

Example Program: Roundtrip

$$c_0[1].\mathtt{send}(40);\, \mathtt{let}\, x = c_0[2].\mathtt{recv}()\, \mathtt{in}\, \mathtt{assert}(x = 42)$$

Challenge: How to guarantee consistent global communication?

Challenge

Challenge: How to guarantee consistent global communication?

$$\begin{array}{l}
\mathtt{\mathbf{let}}\,(c_0, c_1, c_2) = \mathtt{\mathbf{new\_chan}}(3)\ \mathtt{\mathbf{in}} \\
\mathtt{\mathbf{fork}}\ \{\mathtt{\mathbf{let}}\ x = c_1[0].\mathtt{\mathbf{recv}}()\ \mathtt{\mathbf{in}}\ c_1[2].\mathtt{\mathbf{send}}(x + 1)\}\,; \\
\mathtt{\mathbf{fork}}\ \{\mathtt{\mathbf{let}}\ x = c_2[1].\mathtt{\mathbf{recv}}()\ \mathtt{\mathbf{in}}\ c_2[0].\mathtt{\mathbf{send}}(x + 1)\}\,; \\
c_0[1].\mathtt{\mathbf{send}}(40);\ \mathtt{\mathbf{let}}\ x = c_0[2].\mathtt{\mathbf{recv}}()\ \mathtt{\mathbf{in}}\ \mathtt{\mathbf{assert}}(x = 42)
\end{array}$$

Challenge

Challenge: How to guarantee consistent global communication?

$$
\begin{array}{l}
\mathtt{let}\,(c_0, c_1, c_2) = \mathtt{new\_chan}(3)\,\mathtt{in} \\
\mathtt{fork}\ \{\mathtt{let}\,x = c_1[0].\mathtt{recv}()\,\mathtt{in}\,c_1[2].\mathtt{send}(x+1)\}\,; \\
\mathtt{fork}\ \{\mathtt{let}\,x = c_2[1].\mathtt{recv}()\,\mathtt{in}\,c_2[0].\mathtt{send}(x+1)\}\,; \\
c_0[1].\mathtt{send}(40);\,\mathtt{let}\,x = c_0[2].\mathtt{recv}()\,\mathtt{in}\,\mathtt{assert}(x = 42)
\end{array}
$$

Prior work: Syntactic duality

$$
\begin{array}{lcl}
c_0 & : & \mathbf{!}[1]\mathbb{Z}.\,\mathbf{?}[2]\mathbb{Z}.\,\mathbf{end} \\
c_1 & : & \mathbf{?}[0]\mathbb{Z}.\,\mathbf{!}[2]\mathbb{Z}.\,\mathbf{end} \\
c_2 & : & \mathbf{?}[1]\mathbb{Z}.\,\mathbf{!}[0]\mathbb{Z}.\,\mathbf{end}
\end{array}
$$

Challenge

Challenge: How to guarantee consistent global communication?

$$\begin{array}{l}
\mathtt{let}\,(c_0, c_1, c_2) = \mathtt{new\_chan}(3)\;\mathtt{in} \\
\mathtt{fork}\;\{\mathtt{let}\,x = c_1[0].\mathtt{recv}()\;\mathtt{in}\;c_1[2].\mathtt{send}(x+1)\}\,; \\
\mathtt{fork}\;\{\mathtt{let}\,x = c_2[1].\mathtt{recv}()\;\mathtt{in}\;c_2[0].\mathtt{send}(x+1)\}\,; \\
c_0[1].\mathtt{send}(40);\,\mathtt{let}\,x = c_0[2].\mathtt{recv}()\;\mathtt{in}\;\mathtt{assert}(x = 42)
\end{array}$$

Prior work: Syntactic duality

$$\begin{array}{lcl}
c_0 & : & ![1]\mathbb{Z}.\,?[2]\mathbb{Z}.\,\mathbf{end} \\
c_1 & : & ?[0]\mathbb{Z}.\,![2]\mathbb{Z}.\,\mathbf{end} \\
c_2 & : & ?[1]\mathbb{Z}.\,![0]\mathbb{Z}.\,\mathbf{end}
\end{array}$$

This work:

$$c_0 \rightarrowtail\, !\,[1]\,(x:\mathbb{Z})\,\langle x\rangle.\,?[2]\,\langle x+2\rangle.\,\mathbf{end}$$

Challenge

Challenge: How to guarantee consistent global communication?

$$\begin{array}{l}
\mathtt{\mathbf{let}}\,(c_0, c_1, c_2) = \mathtt{\mathbf{new\_chan}}(3)\;\mathtt{\mathbf{in}} \\
\mathtt{\mathbf{fork}}\ \{\mathtt{\mathbf{let}}\, x = c_1[0].\mathtt{\mathbf{recv}}()\;\mathtt{\mathbf{in}}\; c_1[2].\mathtt{\mathbf{send}}(x+1)\}\,; \\
\mathtt{\mathbf{fork}}\ \{\mathtt{\mathbf{let}}\, x = c_2[1].\mathtt{\mathbf{recv}}()\;\mathtt{\mathbf{in}}\; c_2[0].\mathtt{\mathbf{send}}(x+1)\}\,; \\
c_0[1].\mathtt{\mathbf{send}}(40);\,\mathtt{\mathbf{let}}\, x = c_0[2].\mathtt{\mathbf{recv}}()\;\mathtt{\mathbf{in}}\;\mathtt{\mathbf{assert}}(x = 42)
\end{array}$$

Prior work: Syntactic duality

$$\begin{array}{lcl}
c_0 & : & ![1]\mathbb{Z}.\,?[2]\mathbb{Z}.\,\mathbf{end} \\
c_1 & : & ?[0]\mathbb{Z}.\,![2]\mathbb{Z}.\,\mathbf{end} \\
c_2 & : & ?[1]\mathbb{Z}.\,![0]\mathbb{Z}.\,\mathbf{end}
\end{array}$$

This work:

$$\begin{array}{l}
c_0 \rightarrowtail\, !\,[1]\,(x:\mathbb{Z})\,\langle x\rangle.\,?[2]\,\langle x+2\rangle.\,\mathbf{end} \\
c_1 \rightarrowtail\, ?[0]\,(x:\mathbb{Z})\,\langle x\rangle.\,!\,[2]\,\langle x+1\rangle.\,\mathbf{end}
\end{array}$$

Challenge

Challenge: How to guarantee consistent global communication?

$$\begin{array}{l}
\mathtt{let}\,(c_0, c_1, c_2) = \mathtt{new\_chan}(3)\ \mathtt{in} \\
\mathtt{fork}\ \{\mathtt{let}\,x = c_1[0].\mathtt{recv}()\ \mathtt{in}\ c_1[2].\mathtt{send}(x+1)\}\,; \\
\mathtt{fork}\ \{\mathtt{let}\,x = c_2[1].\mathtt{recv}()\ \mathtt{in}\ c_2[0].\mathtt{send}(x+1)\}\,; \\
c_0[1].\mathtt{send}(40);\,\mathtt{let}\,x = c_0[2].\mathtt{recv}()\ \mathtt{in}\ \mathtt{assert}(x = 42)
\end{array}$$

Prior work: Syntactic duality

$$\begin{array}{lcl}
c_0 & : & ![1]\mathbb{Z}.\,?[2]\mathbb{Z}.\,\mathbf{end} \\
c_1 & : & ?[0]\mathbb{Z}.\,![2]\mathbb{Z}.\,\mathbf{end} \\
c_2 & : & ?[1]\mathbb{Z}.\,![0]\mathbb{Z}.\,\mathbf{end}
\end{array}$$

This work:

$$\begin{array}{l}
c_0 \rightarrowtail\, ![1]\,(x:\mathbb{Z})\,\langle x\rangle.\,?[2]\,\langle x+2\rangle.\,\mathbf{end} \\
c_1 \rightarrowtail\, ?[0]\,(x:\mathbb{Z})\,\langle x\rangle.\,!\,[2]\,\langle x+1\rangle.\,\mathbf{end} \\
c_2 \rightarrowtail\, ?[1]\,(x:\mathbb{Z})\,\langle x\rangle.\,!\,[0]\,\langle x+1\rangle.\,\mathbf{end}
\end{array}$$

Challenge

Challenge: How to guarantee consistent global communication?

$$\begin{array}{l}
\texttt{let}\,(c_0, c_1, c_2) = \texttt{new\_chan}(3)\ \texttt{in} \\
\texttt{fork}\ \{\texttt{let}\, x = c_1[0].\texttt{recv}()\ \texttt{in}\ c_1[2].\texttt{send}(x+1)\}\,; \\
\texttt{fork}\ \{\texttt{let}\, x = c_2[1].\texttt{recv}()\ \texttt{in}\ c_2[0].\texttt{send}(x+1)\}\,; \\
c_0[1].\texttt{send}(40);\, \texttt{let}\, x = c_0[2].\texttt{recv}()\ \texttt{in}\ \texttt{assert}(x = 42)
\end{array}$$

Prior work: Syntactic duality

$$\begin{array}{lcl}
c_0 & : & \mathbf{!}[1]\mathbb{Z}.\,\mathbf{?}[2]\mathbb{Z}.\,\mathbf{end} \\
c_1 & : & \mathbf{?}[0]\mathbb{Z}.\,\mathbf{!}[2]\mathbb{Z}.\,\mathbf{end} \\
c_2 & : & \mathbf{?}[1]\mathbb{Z}.\,\mathbf{!}[0]\mathbb{Z}.\,\mathbf{end}
\end{array}$$

This work: Semantic duality

$$\begin{array}{l}
c_0 \rightarrowtail \mathbf{!}\,[1]\,(x:\mathbb{Z})\,\langle x\rangle.\,\mathbf{?}[2]\,\langle x+2\rangle.\,\mathbf{end} \\
c_1 \rightarrowtail \mathbf{?}[0]\,(x:\mathbb{Z})\,\langle x\rangle.\,\mathbf{!}\,[2]\,\langle x+1\rangle.\,\mathbf{end} \\
c_2 \rightarrowtail \mathbf{?}[1]\,(x:\mathbb{Z})\,\langle x\rangle.\,\mathbf{!}\,[0]\,\langle x+1\rangle.\,\mathbf{end}
\end{array}$$

Challenge

Challenge: How to guarantee consistent global communication?

$$\begin{array}{l}
\mathtt{let}\,(c_0, c_1, c_2) = \mathtt{new\_chan}(3)\,\mathtt{in} \\
\mathtt{fork}\ \{\mathtt{let}\,x = c_1[0].\mathtt{recv}()\,\mathtt{in}\,c_1[2].\mathtt{send}(x+1)\}\,; \\
\mathtt{fork}\ \{\mathtt{let}\,x = c_2[1].\mathtt{recv}()\,\mathtt{in}\,c_2[0].\mathtt{send}(x+1)\}\,; \\
c_0[1].\mathtt{send}(40);\,\mathtt{let}\,x = c_0[2].\mathtt{recv}()\,\mathtt{in}\,\mathtt{assert}(x = 42)
\end{array}$$

Prior work: Syntactic duality

$$\begin{array}{lcl}
c_0 & : & ![1]\mathbb{Z}.\,?[2]\mathbb{Z}.\,\mathbf{end} \\
c_1 & : & ?[0]\mathbb{Z}.\,![2]\mathbb{Z}.\,\mathbf{end} \\
c_2 & : & ?[1]\mathbb{Z}.\,![0]\mathbb{Z}.\,\mathbf{end}
\end{array}$$

This work: Semantic duality

$$\begin{array}{l}
c_0 \rightarrowtail\, ![1]\,(x : \mathbb{Z})\,\langle x\rangle.\,?[2]\,\langle x+2\rangle.\,\mathbf{end} \\
c_1 \rightarrowtail\, ?[0]\,(x : \mathbb{Z})\,\langle x\rangle.\,!\,[2]\,\langle x+1\rangle.\,\mathbf{end} \\
c_2 \rightarrowtail\, ?[1]\,(x : \mathbb{Z})\,\langle x\rangle.\,!\,[0]\,\langle x+1\rangle.\,\mathbf{end}
\end{array}$$

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:

$$\begin{array}{l}
\mathbf{let}\,(c_0, c_1, c_2) = \mathbf{new\_chan}(3)\;\mathbf{in} \\
\mathbf{fork}\;\{\mathbf{let}\,x = c_1[0].\mathbf{recv}()\;\mathbf{in}\;c_1[2].\mathbf{send}(x+1)\}\,; \\
\mathbf{fork}\;\{\mathbf{let}\,x = c_2[1].\mathbf{recv}()\;\mathbf{in}\;c_2[0].\mathbf{send}(x+1)\}\,; \\
c_0[1].\mathbf{send}(40);\,\mathbf{let}\,x = c_0[2].\mathbf{recv}()\;\mathbf{in}\;\mathbf{assert}(x = 42)
\end{array}$$

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

Multiparty Actris

Channel endpoint ownership: $c \rightarrowtail p$

Multiparty Actris

Channel endpoint ownership: $c \rightarrowtail p$

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

Multiparty Actris

Channel endpoint ownership: $c \rightarrowtail p$

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

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

Multiparty Actris

Channel endpoint ownership: $c \rightarrowtail p$

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

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

Rules:

Ht-send

$$\{c \rightarrowtail \mathbf{!}\,[i]\,(\vec{x}:\vec{\tau})\,\langle v\rangle .\, p\}\; c[i].\mathbf{send}(v[\vec{t}/\vec{x}])\; \{c \rightarrowtail p[\vec{t}/\vec{x}]\}$$

Multiparty Actris

Channel endpoint ownership: $c \rightarrowtail p$

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

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

Rules:

Ht-send

$$\{c \rightarrowtail \mathbf{!}\,[i]\,(\vec{x}:\vec{\tau})\,\langle v\rangle.\,p\}\; c[i].\mathbf{send}(v[\vec{t}/\vec{x}])\;\{c \rightarrowtail p[\vec{t}/\vec{x}]\}$$

Ht-recv

$$\{c \rightarrowtail \mathbf{?}[i]\,(\vec{x}:\vec{\tau})\,\langle v\rangle.\,p\}\; c[i].\mathbf{recv}()\;\{w.\,\exists\vec{t}.\, w = v[\vec{t}/\vec{x}] * c \rightarrowtail p[\vec{t}/\vec{x}]\}$$

Multiparty Actris

Channel endpoint ownership: $c \rightarrowtail p$

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

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

Rules:

Ht-send

$$\{c \rightarrowtail \mathbf{!}\,[i]\,(\vec{x}:\vec{\tau})\,\langle v\rangle.\,p\}\; c[i].\mathtt{send}(v[\vec{t}/\vec{x}])\; \{c \rightarrowtail p[\vec{t}/\vec{x}]\}$$

Ht-recv

$$\{c \rightarrowtail \mathbf{?}[i]\,(\vec{x}:\vec{\tau})\,\langle v\rangle.\,p\}\; c[i].\mathtt{recv}()\; \{w.\,\exists \vec{t}.\, w = v[\vec{t}/\vec{x}] * c \rightarrowtail p[\vec{t}/\vec{x}]\}$$

Ht-new

$$\{\textsc{consistent}\ \vec{p} * |\vec{p}| = n+1\}\; \mathtt{new\_chan}(|\vec{p}|)\; \{(c_0,\ldots,c_n).\, c_0 \rightarrowtail \vec{p}_0 * \ldots * c_n \rightarrowtail \vec{p}_n\}$$

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.

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.

$$\dfrac{(\forall i, j.\ \texttt{semantic\_dual}\ \vec{p}\ i\ j)}{\textsc{consistent}\ \vec{p}}\ast$$

$$\dfrac{\begin{array}{c}\vec{p}_i = \mathbf{!}\,[j]\,(\vec{x_1} : \vec{\tau_1})\,\langle v_1\rangle.\,p_1 \mathrel{-\!\!\ast} \vec{p}_j = \mathbf{?}[i]\,(\vec{x_2} : \vec{\tau_2})\,\langle v_2\rangle.\,p_2 \mathrel{-\!\!\ast} \\ \forall \vec{x_1} : \vec{\tau_1}.\ \exists \vec{x_2} : \vec{\tau_2}.\ v_1 = v_2 \ast \triangleright(\textsc{consistent}\ (\vec{p}[i := p_1][j := p_2]))\end{array}}{\texttt{semantic\_dual}\ \vec{p}\ i\ j}\ast$$

Protocol Consistency - Example

Protocol consistency example:

$$\begin{aligned}\vec{p}_0 &:= \mathbf{!}\,[1]\,(x:\mathbb{Z})\,\langle x\rangle.\,\mathbf{?}[2]\,\langle x+2\rangle.\,\mathbf{end}\\ \vec{p}_1 &:= \mathbf{?}[0]\,(x:\mathbb{Z})\,\langle x\rangle.\,\mathbf{!}\,[2]\,\langle x+1\rangle.\,\mathbf{end}\\ \vec{p}_2 &:= \mathbf{?}[1]\,(x:\mathbb{Z})\,\langle x\rangle.\,\mathbf{!}\,[0]\,\langle x+1\rangle.\,\mathbf{end}\end{aligned}$$

Protocol consistency:

$$\frac{(\forall i, j.\ \texttt{semantic\_dual}\ \vec{p}\ i\ j)}{\textsc{consistent}\ \vec{p}}\,*$$

$$\frac{\begin{array}{c}\vec{p}_i = \mathbf{!}\,[j]\,(\vec{x_1}:\vec{\tau_1})\,\langle v_1\rangle.\,p_1 \mathrel{-\!\!*} \vec{p}_j = \mathbf{?}[i]\,(\vec{x_2}:\vec{\tau_2})\,\langle v_2\rangle.\,p_2 \mathrel{-\!\!*}\\ \forall \vec{x_1}:\vec{\tau_1}.\,\exists \vec{x_2}:\vec{\tau_2}.\ v_1 = v_2 * \triangleright(\textsc{consistent}\ (\vec{p}[i := p_1][j := p_2]))\end{array}}{\texttt{semantic\_dual}\ \vec{p}\ i\ j}\,*$$

Roundtrip Example - Verified

Roundtrip program:

$$\begin{array}{l}
\mathbf{let}\,(c_0, c_1, c_2) = \mathbf{new\_chan}(3)\,\mathbf{in} \\
\mathbf{fork}\ \{\mathbf{let}\,x = c_1[0].\mathbf{recv}()\,\mathbf{in}\,c_1[2].\mathbf{send}(x+1)\}\,; \\
\mathbf{fork}\ \{\mathbf{let}\,x = c_2[1].\mathbf{recv}()\,\mathbf{in}\,c_2[0].\mathbf{send}(x+1)\}\,; \\
c_0[1].\mathbf{send}(40);\mathbf{let}\,x = c_0[2].\mathbf{recv}()\,\mathbf{in}\ \mathbf{assert}(x = 42)
\end{array}$$

Protocols:

$$\begin{array}{l}
c_0 \rightarrowtail \mathbf{!}\,[1]\,(x:\mathbb{Z})\,\langle x\rangle.\,\mathbf{?}[2]\,\langle x+2\rangle.\,\mathbf{end} \\
c_1 \rightarrowtail \mathbf{?}[0]\,(x:\mathbb{Z})\,\langle x\rangle.\,\mathbf{!}\,[2]\,\langle x+1\rangle.\,\mathbf{end} \\
c_2 \rightarrowtail \mathbf{?}[1]\,(x:\mathbb{Z})\,\langle x\rangle.\,\mathbf{!}\,[0]\,\langle x+1\rangle.\,\mathbf{end}
\end{array}$$

Roundtrip Example - Verified

Roundtrip program:

$$
\begin{array}{l}
\mathbf{let}\,(c_0, c_1, c_2) = \mathbf{new\_chan}(3)\,\mathbf{in} \\
\mathbf{fork}\ \{\mathbf{let}\,x = c_1[0].\mathbf{recv}()\,\mathbf{in}\,c_1[2].\mathbf{send}(x+1)\}\,; \\
\mathbf{fork}\ \{\mathbf{let}\,x = c_2[1].\mathbf{recv}()\,\mathbf{in}\,c_2[0].\mathbf{send}(x+1)\}\,; \\
c_0[1].\mathbf{send}(40);\,\mathbf{let}\,x = c_0[2].\mathbf{recv}()\,\mathbf{in}\ \mathbf{assert}(x = 42)
\end{array}
$$

Protocols:

$$
\begin{array}{l}
c_0 \rightarrowtail\, ![1]\,(x:\mathbb{Z})\,\langle x\rangle.\,?[2]\,\langle x+2\rangle.\,\mathbf{end} \\
c_1 \rightarrowtail\, ?[0]\,(x:\mathbb{Z})\,\langle x\rangle.\,![2]\,\langle x+1\rangle.\,\mathbf{end} \\
c_2 \rightarrowtail\, ?[1]\,(x:\mathbb{Z})\,\langle x\rangle.\,![0]\,\langle x+1\rangle.\,\mathbf{end}
\end{array}
$$

Verified Safety!

Roundtrip Reference Example

Roundtrip reference program:

$$
\begin{array}{l}
\mathbf{let}\,(c_0, c_1, c_2) = \mathbf{new\_chan}(3)\,\mathbf{in} \\
\mathbf{fork}\ \{\mathbf{let}\,\ell = c_1[0].\mathbf{recv}()\,\mathbf{in}\,\ell \leftarrow (!\,\ell + 1);\, c_1[2].\mathbf{send}(\ell)\}\,; \\
\mathbf{fork}\ \{\mathbf{let}\,\ell = c_2[1].\mathbf{recv}()\,\mathbf{in}\,\ell \leftarrow (!\,\ell + 1);\, c_2[0].\mathbf{send}()\}\,; \\
\mathbf{let}\,\ell = \mathbf{ref}\,40\,\mathbf{in}\,c_0[1].\mathbf{send}(\ell);\, c_0[2].\mathbf{recv}();\,\mathbf{let}\,x = !\,\ell\,\mathbf{in}\,\mathbf{assert}(x = 42)
\end{array}
$$

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

Multiparty Actris with Resources

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

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

Rules:

Ht-send

$$\{c \rightarrowtail \mathbf{!}\,[i]\,(\vec{x}:\vec{\tau})\langle v\rangle\{P\}.\,p * P[\vec{t}/\vec{x}]\}\; c[i].\mathbf{send}(v[\vec{t}/\vec{x}])\;\{c \rightarrowtail p[\vec{t}/\vec{x}]\}$$

Ht-recv

$$\{c \rightarrowtail \mathbf{?}\,[i]\,(\vec{x}:\vec{\tau})\langle v\rangle\{P\}.\,p\}\; c[i].\mathbf{recv}()\;\{w.\,\exists\vec{t}.\,w = v[\vec{t}/\vec{x}] * c \rightarrowtail p[\vec{t}/\vec{x}] * P[\vec{t}/\vec{x}]\}$$

Ht-new

$$\{\textsc{consistent}\;\vec{p} * |\vec{p}| = n+1\}\;\mathbf{new\_chan}(|\vec{p}|)\;\{(c_0,\ldots,c_n).\,c_0 \rightarrowtail \vec{p}_0 * \ldots * c_n \rightarrowtail \vec{p}_n\}$$

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.

$$\frac{(\forall i, j.\ \texttt{semantic\_dual}\ \vec{p}\ i\ j)}{\textsc{consistent}\ \vec{p}}\ast$$

$$\frac{\begin{array}{c}\vec{p}_i = \mathbf{!}\,[j]\,(\vec{x_1} : \vec{\tau_1})\langle v_1 \rangle \{P_1\}.\, p_1 \mathrel{-\!\!*} \vec{p}_j = \mathbf{?}\,[i]\,(\vec{x_2} : \vec{\tau_2})\langle v_2 \rangle \{P_2\}.\, p \mathrel{-\!\!*} \\ \forall \vec{x_1} : \vec{\tau_1}.\, P_1 \mathrel{-\!\!*} \exists \vec{x_2} : \vec{\tau_2}.\, v_1 = v_2 \ast P_2 \ast \triangleright(\textsc{consistent}\ (\vec{p}[i := p_1][j := p_2]))\end{array}}{\texttt{semantic\_dual}\ \vec{p}\ i\ j}\ast$$

Protocol Consistency with Resources - Example

Protocol consistency example:

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

Protocol consistency:

$$
\frac{(\forall i, j.\ \texttt{semantic\_dual}\ \vec{p}\ i\ j)}{\textsc{consistent}\ \vec{p}}\ast
$$

$$
\frac{\begin{array}{c}\vec{p}_i = \mathbf{!}\,[j]\,(\vec{x_1} : \vec{\tau_1})\,\langle v_1 \rangle \{P_1\}.\,p_1 \mathrel{-\!\!*} \vec{p}_j = \mathbf{?}[i]\,(\vec{x_2} : \vec{\tau_2})\,\langle v_2 \rangle \{P_2\}.\,p_2 \mathrel{-\!\!*}\\ \forall \vec{x_1} : \vec{\tau_1}.\ P_1 \mathrel{-\!\!*} \exists \vec{x_2} : \vec{\tau_2}.\ v_1 = v_2 \ast P_2 \ast \triangleright(\textsc{consistent}\ (\vec{p}[i := p_1][j := p_2]))\end{array}}{\texttt{semantic\_dual}\ \vec{p}\ i\ j}\ast
$$

Roundtrip Reference Example - Verified

Roundtrip reference program:

$$\begin{array}{l}
\texttt{let}\,(c_0, c_1, c_2) = \texttt{new\_chan}(3)\,\texttt{in} \\
\texttt{fork}\;\{\texttt{let}\,\ell = c_1[0].\texttt{recv}()\,\texttt{in}\,\ell \leftarrow (!\,\ell + 1); c_1[2].\texttt{send}(\ell)\}\,; \\
\texttt{fork}\;\{\texttt{let}\,\ell = c_2[1].\texttt{recv}()\,\texttt{in}\,\ell \leftarrow (!\,\ell + 1); c_2[0].\texttt{send}()\}\,; \\
\texttt{let}\,\ell = \texttt{ref}\,40\,\texttt{in}\,c_0[1].\texttt{send}(\ell); c_0[2].\texttt{recv}(); \texttt{let}\,x = !\,\ell\,\texttt{in}\,\texttt{assert}(x = 42)
\end{array}$$

Protocols:

$$\begin{array}{l}
c_0 \rightarrowtail \mathbf{!}\,[1]\,(\ell : \mathsf{Loc}, x : \mathbb{Z})\,\langle \ell \rangle \{\ell \mapsto x\}.\,\mathbf{?}[2]\,\langle () \rangle \{\ell \mapsto (x + 2)\}.\,\mathbf{end} \\
c_1 \rightarrowtail \mathbf{?}[0]\,(\ell : \mathsf{Loc}, x : \mathbb{Z})\,\langle \ell \rangle \{\ell \mapsto x\}.\,\mathbf{!}\,[2]\,\langle \ell \rangle \{\ell \mapsto (x + 1)\}.\,\mathbf{end} \\
c_2 \rightarrowtail \mathbf{?}[1]\,(\ell : \mathsf{Loc}, x : \mathbb{Z})\,\langle \ell \rangle \{\ell \mapsto x\}.\,\mathbf{!}\,[0]\,\langle () \rangle \{\ell \mapsto (x + 1)\}.\,\mathbf{end}
\end{array}$$

Protocol Consistency - Recursion

Protocols are contractive in the tail:

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

Protocol Consistency - Recursion

Protocols are contractive in the tail:

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

Protocols:

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

Recursion via Löb induction ($\triangleright$):

$$\dfrac{\begin{array}{c}\vec{p}_i = !\,[j]\,(\vec{x_1} : \vec{\tau_1})\,\langle v_1 \rangle \{P_1\}.\, p_1 \mathrel{-\!\!*} \vec{p}_j = ?[i]\,(\vec{x_2} : \vec{\tau_2})\,\langle v_2 \rangle \{P_2\}.\, p_2 \mathrel{-\!\!*} \\ \forall \vec{x_1} : \vec{\tau_1}.\, P_1 \mathrel{-\!\!*} \exists \vec{x_2} : \vec{\tau_2}.\, v_1 = v_2 * P_2 * \triangleright(\textsc{consistent}\ (\vec{p}[i := p_1][j := p_2]))\end{array}}{\texttt{semantic\_dual}\ \vec{p}\ i\ j} *$$

Protocol Consistency - Framing

Consider the replacement of process 1 with a forwarder:

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

Protocol Consistency - Framing

Consider the replacement of process 1 with a forwarder:

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

Protocols:

$$\begin{aligned}
\vec{p}_0 &= \mu rec.\, \mathbf{!}\,[1]\,(\ell : \mathsf{Loc}, x : \mathbb{Z})\,\langle \ell \rangle \{\ell \mapsto x\}.\, \mathbf{?}[2]\,\langle () \rangle \{\ell \mapsto (x+1)\}.\, rec \\
\vec{p}_1 &= \mu rec.\, \mathbf{?}[0]\,(v : \mathsf{Val})\,\langle v \rangle.\, \mathbf{!}\,[2]\,\langle v \rangle.\, rec \\
\vec{p}_2 &= \mu rec.\, \mathbf{?}[1]\,(\ell : \mathsf{Loc}, x : \mathbb{Z})\,\langle \ell \rangle \{\ell \mapsto x\}.\, \mathbf{!}\,[0]\,\langle () \rangle \{\ell \mapsto (x+1)\}.\, rec
\end{aligned}$$

Protocol consistency owns resources while in transit:

$$\frac{\begin{array}{c}\vec{p}_i = \mathbf{!}\,[j]\,(\vec{x_1} : \vec{\tau_1})\,\langle v_1 \rangle \{P_1\}.\, p_1 \mathrel{-\!\!*} \vec{p}_j = \mathbf{?}[i]\,(\vec{x_2} : \vec{\tau_2})\,\langle v_2 \rangle \{P_2\}.\, p_2 \mathrel{-\!\!*} \\ \forall \vec{x_1} : \vec{\tau_1}.\, P_1 \mathrel{-\!\!*} \exists \vec{x_2} : \vec{\tau_2}.\, v_1 = v_2 * P_2 * \triangleright(\textsc{consistent}\,(\vec{p}[i := p_1][j := p_2]))\end{array}}{\texttt{semantic\_dual}\, \vec{p}\; i\; j}\,*$$

Protocol Consistency - Branching

Consider the extension of process 1 with a rerouter:

$$\texttt{let}\,(v, b) = c_1[0].\texttt{recv}()\,\texttt{in}\,c_1[\texttt{if}\,b\,\texttt{then}\,2\,\texttt{else}\,3].\texttt{send}(v)$$

Protocol Consistency - Branching

Consider the extension of process 1 with a rerouter:

$$\texttt{let}\ (v, b) = c_1[0].\texttt{recv}()\ \texttt{in}\ c_1[\texttt{if}\ b\ \texttt{then}\ 2\ \texttt{else}\ 3].\texttt{send}(v)$$

Protocols:

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

We can do case analysis on the binders:

$$\frac{\begin{array}{c}\vec{p}_i = \mathbf{!}\,[j]\,(\vec{x_1} : \vec{\tau_1})\,\langle v_1\rangle\{P_1\}.\,p_1 \mathrel{-\!\!*} \vec{p}_j = \mathbf{?}[i]\,(\vec{x_2} : \vec{\tau_2})\,\langle v_2\rangle\{P_2\}.\,p_2 \mathrel{-\!\!*} \\ \forall \vec{x_1} : \vec{\tau_1}.\ P_1 \mathrel{-\!\!*} \exists \vec{x_2} : \vec{\tau_2}.\ v_1 = v_2 * P_2 * \triangleright(\textsc{consistent}\ (\vec{p}[i := p_1][j := p_2]))\end{array}}{\texttt{semantic\_dual}\ \vec{p}\ i\ j} *$$

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

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*

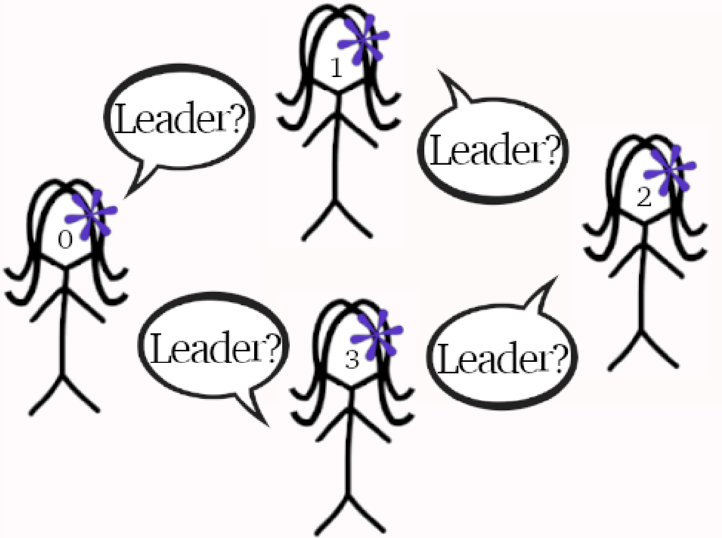
Goal: Prove **uniqueness** and **agreement**

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: $0 \rightarrow 2, 2 \rightarrow 1, 1 \rightarrow 0$

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

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)**

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)**

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)**

Chang and Roberts Ring Leader Election - Implementation

We encode election(i) as **inl** i and elected(i) as **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:

```
$\text{process}\ c\ i \triangleq \mathbf{rec}\ \mathit{rec}\ \mathit{isp} =$
  $\mathbf{match}\ c[i_r].\mathbf{recv}()\ \mathbf{with}$
  $\mid \mathbf{inl}\ i' \Rightarrow \mathbf{if}\ i < i'\ \mathbf{then}\ c[i_l].\mathbf{send}(\mathbf{inl}\ i');\ \mathit{rec}\ \mathbf{true}$          (1.1)
              $\mathbf{else\ if}\ i = i'\ \mathbf{then}\ c[i_l].\mathbf{send}(\mathbf{inr}\ i);\ \mathit{rec}\ \mathbf{false}$   (1.2)
              $\mathbf{else\ if}\ \mathit{isp}\ \mathbf{then}\ \mathit{rec}\ \mathbf{true}$                          (1.3)
              $\mathbf{else}\ c[i_l].\mathbf{send}(\mathbf{inl}\ i);\ \mathit{rec}\ \mathbf{true}$                   (1.4)
  $\mid \mathbf{inr}\ i' \Rightarrow \mathbf{if}\ i = i'\ \mathbf{then}\ i'$                                   (2.1)
              $\mathbf{else}\ c[i_l].\mathbf{send}(\mathbf{inr}\ i');\ i'$                           (2.2)
  $\mathbf{end}$
```

Chang and Roberts Ring Leader Election - Validation

Procedure for starting the election:

$$\text{init}\ c\ i \triangleq c[i_l].\mathtt{\textbf{send}}(\mathtt{\textbf{inl}}\ i);\ \text{process}\ c\ i\ \mathtt{\textbf{true}}$$

Chang and Roberts Ring Leader Election - Validation

Procedure for starting the election:

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

Closed program example of election:

$$\begin{array}{l}
\text{ring\_ref\_prog}\ n \triangleq \\
\quad \mathtt{let}\ \ell = \mathtt{ref}\ 42\ \mathtt{in} \\
\quad \mathtt{let}\ (c_0, \ldots, c_{n-1}) = \mathtt{new\_chan}(n)\ \mathtt{in} \\
\quad \mathtt{for}(i = 1 \ldots (n-1)) \left\{ \mathtt{fork} \left\{ \begin{array}{l} \mathtt{let}\ i' = \text{process}\ c_i\ i\ \mathtt{false}\ \mathtt{in} \\ \mathtt{if}\ i' = i\ \mathtt{then}\ \mathtt{free}\ \ell\ \mathtt{else}\ () \end{array} \right\} \right\}; \\
\quad \mathtt{let}\ i' = \text{init}\ c_0\ 0\ \mathtt{in}\ \mathtt{if}\ i' = 0\ \mathtt{then}\ \mathtt{free}\ \ell\ \mathtt{else}\ ()
\end{array}$$

Chang and Roberts Ring Leader Election - Validation

Procedure for starting the election:

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

Closed program example of election:

$$\begin{array}{l}
\text{ring\_ref\_prog}\ n \triangleq \\
\quad \mathbf{let}\ \ell = \mathbf{ref}\ 42\ \mathbf{in} \\
\quad \mathbf{let}\ (c_0, \ldots, c_{n-1}) = \mathbf{new\_chan}(n)\ \mathbf{in} \\
\quad \mathbf{for}(i = 1 \ldots (n-1)) \left\{ \mathbf{fork} \left\{ \begin{array}{l} \mathbf{let}\ i' = \text{process}\ c_i\ i\ \mathbf{false}\ \mathbf{in} \\ \mathbf{if}\ i' = i\ \mathbf{then}\ \mathbf{free}\ \ell\ \mathbf{else}\ () \end{array} \right\} \right\}; \\
\quad \mathbf{let}\ i' = \text{init}\ c_0\ 0\ \mathbf{in}\ \mathbf{if}\ i' = 0\ \mathbf{then}\ \mathbf{free}\ \ell\ \mathbf{else}\ ()
\end{array}$$

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:

$$
\begin{array}{l}
\mathsf{ring\_prot}(i:\mathbb{N})(P:\mathsf{iProp})(p:\mathbb{N}\to\mathsf{iProto}):\mathbb{B}\to\mathsf{iProto}\triangleq\mu rec.\,\lambda(isp:\mathbb{B}).\\
\quad \&[i_r]\left\{\begin{array}{lll}
\mathbf{inl}(i':\mathbb{N})\langle i'\rangle & \Rightarrow\mathbf{if}\ i<i'\ \mathbf{then}\ !\,[i_l]\,\langle\mathbf{inl}\ i'\rangle.\ rec\ \mathbf{true} & (\mathbf{1.1})\\
 & \quad\mathbf{else}\ \mathbf{if}\ i=i'\ \mathbf{then}\ !\,[i_l]\,\langle\mathbf{inr}\ i\rangle.\ rec\ \mathbf{false} & (\mathbf{1.2})\\
 & \quad\mathbf{else}\ \mathbf{if}\ isp\ \mathbf{then}\ rec\ \mathbf{true} & (\mathbf{1.3})\\
 & \quad\mathbf{else}\ !\,[i_l]\,\langle\mathbf{inl}\ i\rangle.\ rec\ \mathbf{true} & (\mathbf{1.4})\\
\mathbf{inr}(i':\mathbb{N})\langle i'\rangle\{i=i'\Rightarrow P\} & \Rightarrow\mathbf{if}\ i=i'\ \mathbf{then}\ p\ i' & (\mathbf{2.1})\\
 & \quad\mathbf{else}\ !\,[i_l]\,\langle\mathbf{inr}\ i'\rangle.\ p\ i' & (\mathbf{2.2})
\end{array}\right\}
\end{array}
$$

Chang and Roberts Ring Leader Election - Protocol

We can define the ring leader election protocol as:

$$\mathsf{ring\_prot}(i:\mathbb{N})(P:\mathsf{iProp})(p:\mathbb{N}\to\mathsf{iProto}):\mathbb{B}\to\mathsf{iProto}\triangleq \mu rec.\,\lambda(isp:\mathbb{B}).$$

$$\&[i_r]\left\{\begin{array}{lll}
\mathtt{inl}(i':\mathbb{N})\langle i'\rangle & \Rightarrow \mathtt{if}\ i<i'\ \mathtt{then}\ \mathtt{!}\,[i_l]\ \langle \mathtt{inl}\ i'\rangle.\ rec\ \mathtt{true} & \mathbf{(1.1)}\\
 & \quad \mathtt{else\ if}\ i=i'\ \mathtt{then}\ \mathtt{!}\,[i_l]\ \langle\mathtt{inr}\ i\rangle.\ rec\ \mathtt{false} & \mathbf{(1.2)}\\
 & \quad \mathtt{else\ if}\ isp\ \mathtt{then}\ rec\ \mathtt{true} & \mathbf{(1.3)}\\
 & \quad \mathtt{else}\ \mathtt{!}\,[i_l]\ \langle\mathtt{inl}\ i\rangle.\ rec\ \mathtt{true} & \mathbf{(1.4)}\\
\mathtt{inr}(i':\mathbb{N})\langle i'\rangle\{i=i'\Rightarrow P\} & \Rightarrow \mathtt{if}\ i=i'\ \mathtt{then}\ p\ i' & \mathbf{(2.1)}\\
 & \quad \mathtt{else}\ \mathtt{!}\,[i_l]\ \langle\mathtt{inr}\ i'\rangle.\ p\ i' & \mathbf{(2.2)}
\end{array}\right\}$$

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

$$\{c \rightarrowtail \mathsf{ring\_prot}\ i\ P\ p\ isp\}\ \mathsf{process}\ c\ i\ isp\ \{i'.\ c\rightarrowtail (p\ i') * (i=i'\Rightarrow P)\}$$

Chang and Roberts Ring Leader Election - Init

The protocol for starting an election is an extension of the ring protocol:

$$\begin{aligned}&\mathsf{init\_prot}(i : \mathbb{N})(P : \mathsf{iProp})(p : \mathbb{N} \to \mathsf{iProto}) : \mathsf{iProto} \triangleq \\ &\quad \mathbf{!}\,[i_l]\,\langle \mathtt{inl}\; i\rangle\{P\}.\, \mathsf{ring\_prot}\; i\; P\; p\; \mathtt{true}\end{aligned}$$

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:

$$\{c \rightarrowtail (\mathsf{init\_prot}\; i\; P\; p) * P\}\; \mathsf{init}\; c\; i\; \{i'.\, c \rightarrowtail (p\; i') * (i = i' \Rightarrow P)\}$$

Chang and Roberts Ring Leader Election - Leader Uniqueness

$$
\begin{array}{l}
\mathsf{ring\_ref\_prog}\ n \triangleq \\
\quad \mathbf{let}\ \ell = \mathbf{ref}\ 42\ \mathbf{in} \\
\quad \mathbf{let}\ (c_0, \ldots, c_{n-1}) = \mathbf{new\_chan}(n)\ \mathbf{in} \\
\quad \mathbf{for}(i = 1 \ldots (n-1)) \left\{ \mathbf{fork} \left\{ \begin{array}{l} \mathbf{let}\ i' = \mathsf{process}\ c_i\ i\ \mathbf{false}\ \mathbf{in} \\ \mathbf{if}\ i' = i\ \mathbf{then}\ \mathbf{free}\ \ell\ \mathbf{else}\ () \end{array} \right\} \right\}; \\
\quad \mathbf{let}\ i' = \mathsf{init}\ c_0\ 0\ \mathbf{in}\ \mathbf{if}\ i' = 0\ \mathbf{then}\ \mathbf{free}\ \ell\ \mathbf{else}\ ()
\end{array}
$$

We verify the program for 3 participants with the following protocols:

$$
\begin{array}{l}
c_0 \rightarrowtail \mathsf{init\_prot}\ 0\ (\ell \mapsto 42)\ (\lambda i'.\, \mathbf{end}) \\
c_1 \rightarrowtail \mathsf{ring\_prot}\ 1\ (\ell \mapsto 42)\ (\lambda i'.\, \mathbf{end})\ \mathbf{false} \\
c_2 \rightarrowtail \mathsf{ring\_prot}\ 2\ (\ell \mapsto 42)\ (\lambda i'.\, \mathbf{end})\ \mathbf{false}
\end{array}
$$

Chang and Roberts Ring Leader Election - Leader Uniqueness

$$
\begin{array}{l}
\mathsf{ring\_ref\_prog}\ n \triangleq \\
\quad \mathbf{let}\ \ell = \mathbf{ref}\ 42\ \mathbf{in} \\
\quad \mathbf{let}\ (c_0, \ldots, c_{n-1}) = \mathbf{new\_chan}(n)\ \mathbf{in} \\
\quad \mathbf{for}(i = 1 \ldots (n-1)) \left\{ \mathbf{fork} \left\{ \begin{array}{l} \mathbf{let}\ i' = \mathsf{process}\ c_i\ i\ \mathbf{false}\ \mathbf{in} \\ \mathbf{if}\ i' = i\ \mathbf{then}\ \mathbf{free}\ \ell\ \mathbf{else}\ () \end{array} \right\} \right\}; \\
\quad \mathbf{let}\ i' = \mathsf{init}\ c_0\ 0\ \mathbf{in}\ \mathbf{if}\ i' = 0\ \mathbf{then}\ \mathbf{free}\ \ell\ \mathbf{else}\ ()
\end{array}
$$

We verify the program for 3 participants with the following protocols:

$$
\begin{array}{l}
c_0 \rightarrowtail \mathsf{init\_prot}\ 0\ (\ell \mapsto 42)\ (\lambda i'.\ \mathbf{end}) \\
c_1 \rightarrowtail \mathsf{ring\_prot}\ 1\ (\ell \mapsto 42)\ (\lambda i'.\ \mathbf{end})\ \mathbf{false} \\
c_2 \rightarrowtail \mathsf{ring\_prot}\ 2\ (\ell \mapsto 42)\ (\lambda i'.\ \mathbf{end})\ \mathbf{false}
\end{array}
$$

We can thus verify: $\{\mathsf{True}\}\ \mathsf{ring\_ref\_prog}\ 3\ \{\mathsf{True}\}$

Chang and Roberts Ring Leader Election - Leader Uniqueness

$$
\begin{array}{l}
\mathsf{ring\_ref\_prog}\ n \triangleq \\
\quad \texttt{let}\ \ell = \texttt{ref}\ 42\ \texttt{in} \\
\quad \texttt{let}\ (c_0, \ldots, c_{n-1}) = \texttt{new\_chan}(n)\ \texttt{in} \\
\quad \texttt{for}(i = 1 \ldots (n-1)) \left\{ \texttt{fork} \left\{ \begin{array}{l} \texttt{let}\ i' = \mathsf{process}\ c_i\ i\ \texttt{false}\ \texttt{in} \\ \texttt{if}\ i' = i\ \texttt{then}\ \texttt{free}\ \ell\ \texttt{else}\ () \end{array} \right\} \right\}; \\
\quad \texttt{let}\ i' = \mathsf{init}\ c_0\ 0\ \texttt{in}\ \texttt{if}\ i' = 0\ \texttt{then}\ \texttt{free}\ \ell\ \texttt{else}\ ()
\end{array}
$$

We verify the program for 3 participants with the following protocols:

$$
\begin{array}{l}
c_0 \rightarrowtail \mathbf{end} \\
c_1 \rightarrowtail \mathbf{end} \\
c_2 \rightarrowtail \mathbf{end}
\end{array}
$$

We can thus verify: $\{\mathsf{True}\}\ \mathsf{ring\_ref\_prog}\ 3\ \{\mathsf{True}\}$

Chang and Roberts Ring Leader Election - Leader Agreement

```
ring_del_prog $n \triangleq$
  let $(c_0, \ldots, c_n)$ = new_chan$(n + 1)$ in
  fork {let $i'$ = $c_n[0]$.recv() in for$(i = 1 \ldots (n - 1))$ {assert$(c_n[i]$.recv() $= i')$}} ;
  for$(i = 1 \ldots (n - 1))$ {fork {let $i'$ = process $c_i$ $i$ false in $c_i[n]$.send$(i')$}} ;
  let $i'$ = init $c_0$ 0 in $c_0[n]$.send$(i')$
```

Chang and Roberts Ring Leader Election - Leader Agreement

$$
\begin{array}{l}
\mathsf{ring\_del\_prog}\ n \triangleq \\
\quad \mathtt{let}\ (c_0, \ldots, c_n) = \mathtt{new\_chan}(n+1)\ \mathtt{in} \\
\quad \mathtt{fork}\ \left\{ \mathtt{let}\ i' = c_n[0].\mathtt{recv}()\ \mathtt{in}\ \mathtt{for}(i = 1 \ldots (n-1))\ \{\mathtt{assert}(c_n[i].\mathtt{recv}() = i')\} \right\}; \\
\quad \mathtt{for}(i = 1 \ldots (n-1))\ \left\{ \mathtt{fork}\ \left\{ \mathtt{let}\ i' = \mathsf{process}\ c_i\ i\ \mathtt{false}\ \mathtt{in}\ c_i[n].\mathtt{send}(i') \right\} \right\}; \\
\quad \mathtt{let}\ i' = \mathsf{init}\ c_0\ 0\ \mathtt{in}\ c_0[n].\mathtt{send}(i')
\end{array}
$$

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

$$
\begin{array}{l}
c_0 \rightarrowtail \mathsf{init\_prot}\ 0\ \mathsf{True}\ (\lambda i'.\, \mathbf{!}\,[3]\, \langle i' \rangle.\, \mathbf{end}) \\
c_1 \rightarrowtail \mathsf{ring\_prot}\ 1\ \mathsf{True}\ (\lambda i'.\, \mathbf{!}\,[3]\, \langle i' \rangle.\, \mathbf{end})\ \mathtt{false} \\
c_2 \rightarrowtail \mathsf{ring\_prot}\ 2\ \mathsf{True}\ (\lambda i'.\, \mathbf{!}\,[3]\, \langle i' \rangle.\, \mathbf{end})\ \mathtt{false} \\
c_3 \rightarrowtail \mathbf{?}[0]\,(i' : \mathbb{N})\, \langle i' \rangle.\, \mathbf{?}[1]\, \langle i' \rangle.\, \mathbf{?}[2]\, \langle i' \rangle.\, \mathbf{end}
\end{array}
$$

Chang and Roberts Ring Leader Election - Leader Agreement

$$
\begin{array}{l}
\mathsf{ring\_del\_prog}\ n \triangleq \\
\quad \mathbf{let}\ (c_0, \ldots, c_n) = \mathbf{new\_chan}(n+1)\ \mathbf{in} \\
\quad \mathbf{fork}\ \left\{\mathbf{let}\ i' = c_n[0].\mathbf{recv}()\ \mathbf{in}\ \mathbf{for}(i = 1 \ldots (n-1))\ \{\mathbf{assert}(c_n[i].\mathbf{recv}() = i')\}\right\}; \\
\quad \mathbf{for}(i = 1 \ldots (n-1))\ \left\{\mathbf{fork}\ \left\{\mathbf{let}\ i' = \mathsf{process}\ c_i\ i\ \mathbf{false}\ \mathbf{in}\ c_i[n].\mathbf{send}(i')\right\}\right\}; \\
\quad \mathbf{let}\ i' = \mathsf{init}\ c_0\ 0\ \mathbf{in}\ c_0[n].\mathbf{send}(i')
\end{array}
$$

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

$$
\begin{array}{l}
c_0 \rightarrowtail\ ![3]\ \langle 2 \rangle.\ \mathbf{end} \\
c_1 \rightarrowtail\ ![3]\ \langle 2 \rangle.\ \mathbf{end} \\
c_2 \rightarrowtail\ ![3]\ \langle 2 \rangle.\ \mathbf{end} \\
c_3 \rightarrowtail\ ?[0]\ (i' : \mathbb{N})\ \langle i' \rangle.\ ?[1]\ \langle i' \rangle.\ ?[2]\ \langle i' \rangle.\ \mathbf{end}
\end{array}
$$

Chang and Roberts Ring Leader Election - Leader Agreement

$$
\begin{array}{l}
\text{ring\_del\_prog}\ n \triangleq \\
\quad \mathbf{let}\,(c_0,\ldots,c_n) = \mathbf{new\_chan}(n+1)\ \mathbf{in} \\
\quad \mathbf{fork}\ \left\{\mathbf{let}\ i' = c_n[0].\mathbf{recv}()\ \mathbf{in}\ \mathbf{for}(i = 1\ldots(n-1))\,\{\mathbf{assert}(c_n[i].\mathbf{recv}() = i')\}\right\}; \\
\quad \mathbf{for}(i = 1\ldots(n-1))\,\left\{\mathbf{fork}\ \left\{\mathbf{let}\ i' = \text{process}\ c_i\ i\ \mathbf{false}\ \mathbf{in}\ c_i[n].\mathbf{send}(i')\right\}\right\}; \\
\quad \mathbf{let}\ i' = \text{init}\ c_0\ 0\ \mathbf{in}\ c_0[n].\mathbf{send}(i')
\end{array}
$$

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

$$
\begin{array}{l}
c_0 \rightarrowtail \mathbf{end} \\
c_1 \rightarrowtail \mathbf{end} \\
c_2 \rightarrowtail \mathbf{end} \\
c_3 \rightarrowtail \mathbf{end}
\end{array}
$$

Chang and Roberts Ring Leader Election - Leader Agreement

$$
\begin{array}{l}
\mathsf{ring\_del\_prog}\; n \triangleq \\
\quad \mathtt{let}\;(c_0,\ldots,c_n) = \mathtt{new\_chan}(n+1)\;\mathtt{in} \\
\quad \mathtt{fork}\;\{\mathtt{let}\; i' = c_n[0].\mathtt{recv}()\;\mathtt{in}\;\mathtt{for}(i = 1\ldots(n-1))\,\{\mathtt{assert}(c_n[i].\mathtt{recv}() = i')\}\}\,; \\
\quad \mathtt{for}(i = 1\ldots(n-1))\,\{\mathtt{fork}\;\{\mathtt{let}\; i' = \mathsf{process}\; c_i\; i\;\mathtt{false}\;\mathtt{in}\; c_i[n].\mathtt{send}(i')\}\}\,; \\
\quad \mathtt{let}\; i' = \mathsf{init}\; c_0\; 0\;\mathtt{in}\; c_0[n].\mathtt{send}(i')
\end{array}
$$

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

$$c_0 \rightarrowtail \mathbf{end}$$

$$c_1 \rightarrowtail \mathbf{end}$$

$$c_2 \rightarrowtail \mathbf{end}$$

$$c_3 \rightarrowtail \mathbf{end}$$

We can thus verify: $\{\mathsf{True}\}\;\mathsf{ring\_del\_prog}\;3\;\{\mathsf{True}\}$

Language Parametricity of Multiparty Actris

Multiparty Actris Ghost Theory

We prove language-generic ghost theory rules:

$$\textsc{proto-alloc}\qquad \frac{\textsc{consistent}\ \vec{p}}{\mathrel{|}\!\!\Rrightarrow \exists \chi.\ \mathsf{prot\_ctx}\ \chi\ |\vec{p}| * \mathop{\Huge *}_{i \mapsto p \in \vec{p}} \mathsf{prot\_own}\ \chi\ i\ p}$$

$$\textsc{proto-valid}\qquad \frac{\mathsf{prot\_ctx}\ \chi\ n \qquad \mathsf{prot\_own}\ \chi\ i\ p}{i < n}$$

$$\textsc{proto-step}\qquad \frac{\begin{array}{c}\mathsf{prot\_ctx}\ \chi\ n \qquad P_1[\vec{t_1}/\vec{x_1}] \\ \mathsf{prot\_own}\ \chi\ i\ (\mathbf{!}\,[j]\,(\vec{x_1}:\vec{\tau_1})\,\langle v_1\rangle\{P_1\}.\,p_1) \qquad \mathsf{prot\_own}\ \chi\ j\ (\mathbf{?}[i]\,(\vec{x_2}:\vec{\tau_2})\,\langle v_2\rangle\{P_2\}.\,p_2)\end{array}}{\begin{array}{c}\mathrel{|}\!\!\Rrightarrow \triangleright \exists(\vec{t_2} : \vec{\tau_2}).\ \mathsf{prot\_ctx}\ \chi\ * \mathsf{prot\_own}\ \chi\ i\ (p_1[\vec{t_1}/\vec{x_1}]) * \mathsf{prot\_own}\ \chi\ j\ (p_2[\vec{t_2}/\vec{x_2}]) * \\ (v_1[\vec{t_1}/\vec{x_1}]) = (v_2[\vec{t_2}/\vec{x_2}]) * P_2[\vec{t_2}/\vec{x_2}]\end{array}}$$

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

- Such as Ht-send, Ht-recv, and Ht-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

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, ...

$$
\begin{aligned}
&\mathbf{!}\,[1]\,\langle\text{“Thank you”}\rangle\{\texttt{MultrisOverview}\}.\\
&\mu \mathit{rec}.\,\mathbf{?}[1]\,(q : \texttt{Question}\ \texttt{i})\,\langle q\rangle\{\texttt{AboutMultris}\ q\}.\\
&\qquad\quad \mathbf{!}\,[i]\,(a : \texttt{Answer})\,\langle a\rangle\{\texttt{Insightful}\ q\ a\}.\,\mathit{rec}
\end{aligned}
$$