Mixtris: Mechanised Higher-Order Separation Logic for Mixed Choice Multiparty Message Passing

JONAS KASTBERG HINRICHSEN, Aalborg University, Denmark
IWAN QUÉMERAIS, ENS-Lyon, France
LARS BIRKEDAL, Aarhus University, Denmark

Mixed choice multiparty message passing is an expressive concurrency programming paradigm where components use non-determinism to choose between concurrent options for sending and receiving messages. This flexibility makes it possible to program advanced algorithms, such as leader election protocols, succinctly. We present Mixtris, a mechanised higher-order separation logic for reasoning about functional correctness of higher-order imperative programs with mixed choice multiparty message passing. Mixtris builds upon recent work on separation logic for (non-mixed choice) multiparty message-passing programs, by drawing inspiration from session type systems for mixed choice multiparty message-passing programs. Mixtris is the first program logic for mixed choice multiparty message passing. We prove soundness of Mixtris using a novel model of our mixed choice multiparty protocols. We demonstrate how Mixtris can be used to formally reason about challenging examples, including some leader election protocols such as Chang and Roberts' ring leader election protocol. All the results in the paper (both meta-theory and examples) have been formalised in the Rocq proof assistant on top of the Iris program logic framework.

Additional Key Words and Phrases: Mixed choice, multiparty, message passing, session types, separation logic

1 Introduction

Mixed choice [22] is a variant of message passing that allows non-deterministic branching between a set of choices containing both inputs and outputs, guaranteeing that only one such choice happens, and that both parties agree on it. Mixed choice is central to succinctly modeling key components of multiparty concurrent systems such as consensus algorithms like leader election [23]. There currently exists no techniques for verifying functional correctness—a crucial property for verifying leader uniqueness and agreement [12]—of systems using mixed choice multiparty message passing. The state-of-the-art of verifying mixed choice systems is the session type system by Peters and Yoshida [26], that enables decidable verification of crash- and deadlock-freedom of concurrent systems expressed in a synchronous π-calculus setting. π-calculus famously specialise in expressing interactions between processes, while abstracting over their implementation-level details. In contrast, Hinrichsen et al. [12] developed a separation logic for interactive verification of partial functional correctness for implementation-level non-mixed choice multiparty message passing in the functional setting; where channel endpoints are first-class terms alongside other programming paradigms, such as shared memory and higher-order functions. Their semantics are akin to those explored in work on GV-style session type systems [8, 14, 36]. At present, there is limited understanding of how mixed choice can be implemented in the functional setting [27, 30, 31]. Closest is Reppy et al. [27], who implemented a mixed choice-like construct in Concurrent ML, and developed their own model checker, alongside test cases exploring upwards of one million execution traces, to garner faith in their result. Finally, no prior results regarding mixed choice message passing have been foundationally verified [1] (also known as mechanised); having a formal soundness theorem, that is proven sound w.r.t. the operational semantics of the system in a proof assistant. Given that unsound results have previously been found in the literature on message passing verification [28], the value of mechanised results is evident.

Authors' Contact Information: Jonas Kastberg Hinrichsen, Aalborg University, Denmark, jkas@itu.dk; Iwan Quémerais, ENS-Lyon, France, iwan.quemerais@ens-lyon.fr; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.com.

To address these gaps, this paper introduces **Mixtris**, a *foundationally verified* higher-order concurrent separation logic for interactive verification of partial *functional correctness* of mixed choice multiparty message-passing, based on a novel *implementation* of mixed choice multiparty message passing in the functional setting.

Mixed choice semantics. Defining a semantics for mixed choice in the multi-threaded functional setting pose interesting challenges, as a consequence of the inherent asynchrony. To understand this challenge, let us first consider the synchronous π-calculus semantics of mixed choice. In synchronous π-calculus, mixed choice is expressed as a range of concurrent choices, where two matching parties are non-deterministically chosen to synchronise in one step, e.g. consider:

$$\begin{aligned} e_A &:= ![B]\langle v_1\rangle.e_{A1} + ?[C](x_3).e_{A2} \\ e_B &:= ![C]\langle v_2\rangle.e_{B1} + ?[A](x_1).e_{B2} \\ e_C &:= ![A]\langle v_3\rangle.e_{C1} + ?[B](x_2).e_{C2} \end{aligned}$$

Where the three processes are executed in parallel ($e_A \mid\mid e_B \mid\mid e_C$), and each process tries to either (+) send (**!**) a value to the right (*e.g.,* A sending v_1 to B, continuing as A_1) or receive (**?**) a value from the left (*e.g.,* A receiving v_3 from C, binding it to x_3, continuing as $e_{A2}[v_3/x_3]$). The system will non-deterministically reduce to one of the following in one step:

$$e_{A1} \mid\mid e_{B2}[v_1/x_1] \mid\mid e_C \qquad e_A \mid\mid e_{B1} \mid\mid e_{C2}[v_2/x_2] \qquad e_{A2}[v_3/x_3] \mid\mid e_B \mid\mid e_{C1}$$

Conventionally for multi-threaded functional languages, we have per-thread semantics, where operations on each thread are interleaved by a scheduler. As a result, we face a challenge when modelling synchronous exchange; any exchange will have at least three individual steps: (1) a handshake is extended, (2) the handshake is accepted by the message destination, (3) the handshake is observed by the message origin. Consequently, if a participant has other outstanding handshakes between (2) and (3), they may be accepted by other parties, resulting in a race condition that violate the safety semantics of mixed choice, which require that only one choice is made. We navigate this challenge with a first approximation of mixed choice semantics in the functional setting, by enforcing mutual exclusion on attempted handshakes for each participant, and subsequently encode mixed choice by alternating between the individual choices until one succeeds.

A limitation of our approach is that it does not deterministically make progress: parties may repeatedly miss each others handshake attempts. However, given parties that repeatedly attempt the individual handshakes of a mixed choice, along with uniform scheduling, every attempt has a non-zero chance to succeed—that the corresponding party is scheduled to accept the corresponding handshake. Given enough time, we informally conclude that the handshake, and thereby the mixed choice, enjoys almost-sure termination under uniform scheduling and repeated handshake attempts.

In summary, our semantics can emulate the above synchronous π-calculus configuration, where the reduction happens over some bounded number of steps, assuming uniform scheduling and repeated handshake attempts. This is evidenced by our protocol language, that more closely resemble the structure of mixed choice multiparty session types [26] where synchronisation happens in one step (during step (2) presented above), as illustrated below. Our solution and liveness argument carry semblance to prior work on encoding synchronous mixed choice in asynchronous settings, often referred to as the "binary decision problem" [7, 24, 35], which we further discuss in §7.

Overview of Mixtris. Our implementation of mixed choice is facilitated by using *uncommitted* message-passing primitives; $c[i]$.**try_send**(v) and $c[i]$.**try_recv**$()$. Here, c is the channel endpoint we operate on, i is the id of the participant we attempt to interact with, and v is the sent value. The implementation uses a novel concept of synchronisation cells, that are carefully designed w.r.t. the above three linearisation points. A multiparty channel of n participants is achieved using an

$n \times n$ matrix of synchronisation cells, where each entry (i, j) is used as the synchronisation cell for sending values from i to j. The uncommitted primitives differ from the mixed choice primitives of prior work on multiparty communication, which uses a *ranged choice* that concurrently attempts an arbitrary amount of choices. Using uncommitted send and receives we emulate the prior primitives for mixed choice by alternating between trying to send and trying to receive until one of them succeed. For binary choice, this would be:

```
send_recv c i j v ≜ if c[i].try_send(v) then none
                    else match c[j].try_recv() with
                         | some x ⇒ some x
                         | none   ⇒ send_recv c i j v
                         end
```

Here, send_recv first tries to send v to i, and returns **none** in case of success. In case of failure, it tries to receive from j, and does a case analysis on the result using **match**. In the case of success, it binds the result to x and returns **some** x. In case of failure the program loops. With this, we can effectively emulate the π-calculus mixed choice example above, as shown below. To verify functional correctness of mixed choice programs, such as the above, we draw inspiration from the verification foundation of Multris [12], the aforementioned separation logic for functional correctness of non-mixed choice multiparty message passing. We extend the verification interface of Multris, adding a mixed choice construct to their protocol language of multiparty dependent separation protocols, to be used alongside their notion of channel endpoint ownership:

$$c \rightarrowtail (\mathbf{!}\,[i]\,(\vec{x}:\vec{\tau})\,\langle v\rangle\{P\}.\,p) + (\mathbf{?}[j]\,(\vec{y}:\vec{\sigma})\,\langle w\rangle\{Q\}.\,q)$$

Here, $c \rightarrowtail \ldots$ asserts exclusive permission to use channel endpoint c in accordance with the protocol. In the protocol, $\mathbf{!}/\mathbf{?}$ specifies sending/receiving, i specifies the party we communicate with, and $\vec{x}:\vec{\tau}$ specify newly introduced information, in the form of binders that bind into the remaining protocol. In the remaining protocol, v specifies the exchanged value, P specifies exchanged separation logic resources, and p specifies the protocol continuation. The mixed choice operator + is novel, and specifies that a choice can be made between the left and the right protocol, even when one is sending and the other is receiving.

The Mixtris logic provides rules for the uncommitted send and receive primitives, which require that the operation under consideration corresponds to one of the choices of the protocol. In the case of success, the protocol reduces w.r.t. the choice, while in the case of failure, the protocol state remains unchanged, thus preserving all possible choices. With these rules, we can verify functional correctness of the program w.r.t. the protocol above. Note that the protocol intentionally does not describe the implementation-level looping behaviour of the program, but is strictly concerned with the case of successful exchanges, similar to mixed choice multiparty session types [26].

To verify complete programs Mixtris has to guarantee *protocol consistency*; that each local protocol is sound w.r.t. the protocols it interacts with. Drawing inspiration from Multris, we carry out this proof as a simulation of all possible paths that can be taken by a pool of protocols. For each interaction, we must prove that the binders, value, and resources required by the receiver can be satisfied by that of the sender. In addition, we can use the current environment; any previously known binders, and—by virtue of working in separation logic—available resources.

This is a key feature, as it allows delegating resources that never entered the system during the protocol. In the case of leader election, the elected leader is often given elevated privileges. Such resources do not necessarily enter the system at any point of the protocol. Rather, they exist beforehand and are simply

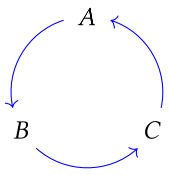

Fig. 1. Election

delegated upon election completion. To demonstrate this idea, consider the simple leader election in Fig. 1 with 3 participants: A, B, and C, that each use mixed choice to race for the leadership. Only one of the 3 possible communications may happen, thus the elected leader will be the one who receives a message, and will be given elevated privileges in the form of separation logic resources; *e.g.,* if B sends to C, then C will be elected leader. Now consider a situation where we have a pre-allocated reference ℓ, which the leader should deallocate. We can implement each participant, using the send_recv construct as follows:

Party A:

```
match send_recv c_A B C () with
| none   ⇒ ()
| some _ ⇒ free ℓ
end
```

Party B:

```
match send_recv c_B C A () with
| none   ⇒ ()
| some _ ⇒ free ℓ
end
```

Party C:

```
match send_recv c_C A B () with
| none   ⇒ ()
| some _ ⇒ free ℓ
end
```

The **free** ℓ operation is only safe to execute the first time it is performed (as per no use-after-free). To guarantee no use-after-free, conventional separation logic captures the permission to exclusively operate on a reference via the resource $\ell \mapsto -$, which is consumed by **free** ℓ. We can employ this approach in Mixtris via protocols like the following instance of the above protocol for send_recv:

$$\mathbf{!}\,[i]\,.\,\mathbf{end} + \mathbf{?}[j]\,\{\ell \mapsto -\}.\,\mathbf{end}$$

Note that we omit $\langle v \rangle$, whenever $v := ()$, and $\{P\}$, whenever $P :=$ True. The protocol specifies that the resource $\ell \mapsto -$ is received alongside the received message, which allows the elected leader—and only the elected leader—to free the reference. The proof of protocol consistency is straightforward, as only one of the three possible exchanges happen, which we can satisfy using the pre-existing resource $\ell \mapsto -$.

We thus conclude the formal proof of partial functional correctness; if any exchange happens, at most one happens, as we would otherwise violate use-after-free, which is ruled out by the logic. We recall that the logic does *not* guarantee that an exchange eventually happens, but informally conclude this under uniform scheduling, as the program satisfies the aforementioned assumption which require that handshakes are repeatedly attempted.

To demonstrate the expressive power of Mixtris, we verify the ring leader election algorithm by Chang and Roberts [6]. A simplified version of the algorithm, where only one fixed participant can start an election, was verified in Multris [12]. We also verify a leader election algorithm considered by the state-of-the-art work on mixed choice multiparty session types [26]. The algorithm decides a leader using races over mixed choice, similar to the above, but with 5 participants over 2 rounds.

We remark that the Mixtris logic is *backwards compatible* w.r.t. Multris; all of the constructions and rules of Multris are available in Mixtris. However, to support mixed choice we had to change the foundational definition of their multiparty dependent separation protocols, and consequently *re-prove* the soundness of all the pre-existing rules. As such, the similarity is an earned benefit, that allowed us to reuse the surface-level verification approach of Multris, and directly inherit all existing verified examples in Multris. To make the distinction between borrowed (but re-proven) and novel clear, we colour-code novel concepts in the context of Multris.

The Mixtris logic is foundationally verified on top of the Iris framework [17, 19, 20]. This is achieved by first verifying higher-order atomic specifications for the novel synchronisation cell construction, to properly leverage its linearisation points. We then construct the mixed-choice multiparty dependent separation protocols using Iris's support for solving guarded recursive domain equations. We prove a language-agnostic reasoning principle for mixed choice multiparty message passing in separation logic. We finally define the channel endpoint ownership, in terms of the synchronisation cell abstraction and the ghost theory, and use it to prove the Mixtris rules.

Contributions. The contributions of the paper can be summarised as follows:

- We give an implementation of a novel synchronisation cell construct, and build multiparty channels with uncommitted message-passing primitives on top of them (§ 2).
- We present Mixtris, a higher-order concurrent separation logic for verifying partial functional correctness of mixed choice message-passing programs (§ 3).
- We demonstrate the expressive power of Mixtris by implementing and verifying extended versions of two leader election algorithms considered by the state-of-the-art (§ 4).
- We give a foundational soundness proof of Mixtris and using Iris (§ 5)
- We give the first mechanised result regarding the verification of mixed choice message passing; all our results are mechanised in the Rocq Prover, on top of the Iris framework (§ 6).

All of our results are mechanised in the Rocq prover [33], constituting the first mechanised result on mixed choice verification, and can be found in our accompanying artifact [2].

2 Semantics and Implementation of Mixed Choice Multiparty Channels

In this section we give our implementation for mixed choice multiparty channels in the functional setting. We first give an overview of the semantics of the shared memory language that we are working with § 2.1. We then give the novel implementation of the *synchronisation cells*, which is the foundation of our channels (§ 2.2). We then give the implementation for the channels (§ 2.3). Finally, we give the complete overview of the example presented in § 1 (§ 2.4).

2.1 Shared Memory Semantics

The Mixtris channels are built on top of shared memory references in an untyped functional language with higher-order functions, higher-order mutable references, fork-based concurrency, and atomic instructions for thread-safe concurrent memory manipulation. The language and its proof rules are inherited from the Iris framework, and is known as HeapLang [32].

$$\begin{array}{lll}
e := & n \mid \mathtt{true} \mid \mathtt{false} \mid e + e \mid e - e \mid e < e \mid \ \ldots & \text{(Constants and operators)} \\
& \mathtt{assert}(e) \mid \mathtt{let}\ x = e\ \mathtt{in}\ e \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e & \text{(Control flow)} \\
& (e, e) \mid \ \mathtt{fst}\ e \mid \ \mathtt{snd}\ e \mid \lambda x.e \mid e\ e \mid \mathtt{rec}\ \mathrm{f}\ x.\ e \mid & \text{(Pairs and functions)} \\
& \mathtt{inl}\ e \mid \ \mathtt{inr}\ e \mid \mathtt{match}\ e\ \mathtt{with}\ \mathtt{inl}\ e \Rightarrow e;\ \mathtt{inr}\ e \Rightarrow e\ \mathtt{end} \mid & \text{(Tagged unions)} \\
& \mathtt{ref}\ e \mid \ \mathtt{free}\ e \mid !e \mid e \leftarrow e \mid \ \mathtt{Xchg}\ e\ e \mid \ldots & \text{(References and atomics)}
\end{array}$$

The language has a strict right-to-left evaluation order. The concurrent semantics use a configuration over a thread pool $\vec{e}$ and a shared heap h. Every step of the configuration steps over an arbitrarily chosen $e \in \vec{e}$, which may add new threads to the pool. The semantics defines crashing behaviour by some reductions being invalid, *e.g.*, running $\mathtt{free}\ \ell$ where ℓ is not allocated, or $\mathtt{assert}(b)$ where b evaluates to **false**. The most notable instruction is $\mathtt{Xchg}\ \ell\ v$, which atomically (1) returns the current value of ℓ, and (2) updates the reference to contain v. We use **Xchg** as the foundation for our synchronisation cell implementation. The remaining language instructions are quite standard, and so we elide detailed exposition. Note that channel operations are not primitive to the language but implemented.

2.2 Implementation of Synchronisation Cells

The goal of the synchronisation cells is to have a generic primitive for synchronous directed binary communication between two threads on top of which our channels can be defined. By synchronous we mean that the putter and getter agree on the result of a transaction. We define the synchronisation cells in terms of the `new_sync`, `sync_put`, `sync_get`, `sync_try_put` and `sync_try_get` primitives, shown in Fig. 2. It is worth nothing that the cells are not specific to Mixtris, and can be used in other settings using communications between threads.

```
new_sync () := ref none


wait c := match !c with
          | none   ⇒ ()
          | some _ ⇒ wait c
          end
```

```
sync_put c v := c ← some v;
                wait c.


sync_try_put c v :=
  c ← some v;
  match Xchg c none with
  | none   ⇒ true
  | some _ ⇒ false
  end
```

```
sync_get c :=
  match Xchg c none with
  | none   ⇒ sync_get c
  | some v ⇒ v
  end

sync_try_get c := Xchg c none
```

Fig. 2. Implementation of synchronisation cells

new_sync creates an initially empty reference (**ref none**) that will be used as a synchronisation cell. sync_put puts a value in the synchronisation cell ($c \leftarrow$ **some** v) and calls wait which loops until the cell is empty again. sync_get empties the reference with the atomic **Xchg** c **none** operation and: if it was already empty (**none**) then nothing was gotten and we loop to try again, otherwise (**some** v) the stored value v is returned. sync_try_put puts a value in the synchronisation cell, and then atomically checks if the value was gotten.[1] If the value was gotten, then the put was a success and we return **true**, otherwise the value is taken back and we return **false**. sync_try_get tries to take a sent value out of the cell. If the cell was empty, the get failed and we return **none**, otherwise the received value is taken out and returned.

The implementation enjoys the three linearisation points presented in §1. (1) happens during the $c \leftarrow$ **some** v instruction in sync_try_put. (2) happens during the **Xchg** c **none** instruction in sync_try_get. (3) happens during the **Xchg** c **none** instruction in sync_try_put. In summary, a successful exchange occurs when the three points happen in order.

It is worth noting that the committed and uncommitted instructions are compatible. If a putter calls the committed sync_put instruction, after which the getter calls sync_try_get, they will successfully exchange the value. The dual case, with sync_get and sync_try_put, is also valid.

2.3 Implementation of Mixed Choice Multiparty Channels

With the synchronisation cells, we can now implement the channels of Mixtris. The implementation uses a matrix library, whose implementation we elide for brevity sake. The matrix library has two instructions, new_matrix, and $m_{i,j}$. The instruction new_matrix $n\,m\,f$, which creates and $n \times m$ matrix, and populate each entry (i, j) using the function argument $f\ i\ j$. The instruction $m_{i,j}$, which returns the value at entry (i, j) of the matrix.

The implementation of the channels can be found in Fig. 3. new_chan creates a matrix of $n \times n$ synchronisation cells, where each participant i, for each corresponding participant j, uses the synchronisation cell stored in (i, j) and (j, i) for sending and receiving, respectively. The channel endpoint for each participant i is a tuple (m,i), returned by new_chan. **send** uses the synchronisation cell $m_{i,j}$ to send the value v to the participant j. **recv** uses the synchronisation cell $m_{j,i}$ to receive from the participant j. **try_send** and **try_recv** are similar to **send** and **recv** respectively, using the uncommitted counterparts of the synchronisation cell primitives.

We remark that the implementation does not break abstraction w.r.t the underlying synchronisation channels. As a result, a user can give their own implementation, provided that they preserve

[1] A more live implementation can be achieved by letting the putter sleep for a bit after putting the value in the cell. However, we elide such detail, as the verification effort remains unchanged.

$$\texttt{new\_chan}\, n := \ \textbf{let}\ m = \texttt{new\_matrix}\ n\ n\ (\lambda\_,\_.\, \texttt{new\_sync}())\ \textbf{in}\ ((m,0),\ldots,(m,n-1))$$

$$c[j].\textbf{send}(v) := \textbf{let}\ (m,i) = c\ \textbf{in} \quad \texttt{sync\_put}\ m_{i,j}\ v \qquad c[j].\textbf{recv}() := \textbf{let}\ (m,i) = c\ \textbf{in} \quad \texttt{sync\_get}\ m_{j,i}$$

$$c[j].\textbf{try\_send}(v) := \textbf{let}\ (m,i) = c\ \textbf{in} \quad \texttt{sync\_try\_put}\ m_{i,j}\ v \qquad c[j].\textbf{try\_recv}() := \textbf{let}\ (m,i) = c\ \textbf{in} \quad \texttt{sync\_try\_get}\ m_{j,i}$$

Fig. 3. Implementation of communication channels

the necessary synchronisation requirements. In §5.1 we give a formal account for the necessary requirements, in the form of specifications for the synchronisation cells, which we subsequently verify our channel specifications on top of.

With the uncommitted primitives we can emulate the ranged mixed choice (over an arbitrary amount of choices), by attempting each in sequence, and loop in the case that none succeed, similar such as the binary case where we used the send_recv construction. As discussed in §1 this use of the uncommitted primitives is live, under uniform scheduling.

2.4 Example Mixed-Choice Program: Three-way Leader Election

We can now give a precise definition of the threeway election example presented in §1:

$$\begin{array}{l}
\texttt{threeway\_election\_example} \triangleq \\
\quad \textbf{let}\ \ell = \textbf{ref}\, 42\ \textbf{in} \\
\quad \textbf{let}\ (c_A, c_B, c_C) = \textbf{new\_chan}(3)\ \textbf{in} \\
\quad \textbf{fork}\, \{\textbf{match}\ \texttt{send\_recv}\ c_B\ C\ A\ ()\ \textbf{with none} \Rightarrow ();\ \textbf{some}\ \_ \Rightarrow \textbf{free}\ \ell\ \textbf{end}\}\,; \\
\quad \textbf{fork}\, \{\textbf{match}\ \texttt{send\_recv}\ c_C\ A\ B\ ()\ \textbf{with none} \Rightarrow ();\ \textbf{some}\ \_ \Rightarrow \textbf{free}\ \ell\ \textbf{end}\}\,; \\
\quad \textbf{match}\ \texttt{send\_recv}\ c_A\ B\ C\ ()\ \textbf{with none} \Rightarrow ();\ \textbf{some}\ \_ \Rightarrow \textbf{free}\ \ell\ \textbf{end}
\end{array}$$

3 The Mixtris Logic

In this section we present the Mixtris logic for mixed choice multiparty message passing. We first describe the Mixtris separation logic foundation, along with its soundness theorem (§3.1). We then introduce our mixed choice multiparty protocol language (§3.2). We then describe the Mixtris mixed choice multiparty message passing rules, based on the Mixtris protocols (§3.3). We finally show how to prove the notion of *protocol consistency*; a property that ensures that all expectations of a receiver can be satisfied by the sender, for any interleaving of the non-deterministic communication (§3.4).

3.1 The Mixtris Separation Logic Foundation and Adequacy Theorem

Mixtris is an Iris-based higher-order concurrent separation logic with standard rules for heap manipulation and fork-based concurrency. A proof in Mixtris guarantees crash-freedom (and consequently memory safety). Reasoning is based on weakest preconditions $\text{wp}\ e\ \{\Phi\}$, which states that (1) the expression e is safe to execute, and (2) if the expression terminates with some value v, the postcondition $\Phi\, v$ holds. Weakest preconditions are sufficient for defining the more standard Hoare triples as $\{P\}\ e\ \{\Phi\} \triangleq P \vdash \text{wp}\ e\ \{\Phi\}$. The Mixtris adequacy theorem is as follows:

Theorem 3.1 (Mixtris Adequacy). *A proof of* $\text{wp}\ e\ \{\Phi\}$ *guarantees that* e *is* ***safe****, i.e., if* $([e], \emptyset) \to^* ([e_0 \ldots e_n], h)$*, then for each* $i \leq n$ *either* e_i *is a value or* (e_i, h) *can do a step. Furthermore, any returned value* v *of* e *satisfies* $\Phi(v)$*.*

Separation logic propositions:

$$\begin{array}{lll}
P, Q \in \texttt{iProp} & ::= \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q & (\text{ Propositional logic })\\
& \mid \forall x.\, P \mid \exists x.\, P \mid x \;=\; y & (\text{ Higher-order logic with equality })\\
& \mid P * Q \mid P \mathrel{-\!\!*} Q & (\text{ Separation logic })\\
& \mid \,\triangleright P \mid \mathsf{wp}\ e\ \{\Phi\} & (\text{ Step indexing and weakest preconditions })\\
& \mid \ell \mapsto x & (\text{ Heap cell ownership })\\
& \mid c \rightarrowtail p \mid p \sqsubseteq q & (\text{ Channel ownership and subprotocol relation })
\end{array}$$

Basic weakest precondition rules (excerpt):

$$\begin{array}{cccc}
\textsc{Wp-alloc} & \textsc{Wp-load} & \textsc{Wp-store} & \textsc{Wp-free}\\
\mathsf{wp}\ \texttt{ref}\,v\ \{\ell.\, \ell \mapsto v\} & \dfrac{\ell \mapsto v}{\mathsf{wp}\ !\ell\ \{w.\, w = v\ *\ \ell \mapsto v\}}* & \dfrac{\ell \mapsto v}{\mathsf{wp}\ \ell \leftarrow w\ \{\ell \mapsto w\}}* & \dfrac{\ell \mapsto v}{\mathsf{wp}\ \texttt{free}\,\ell\ \{\mathsf{True}\}}*
\end{array}$$

$$\begin{array}{ccc}
\textsc{Wp-xchg} & \textsc{Wp-fork} & \textsc{Wp-bind}\\
\dfrac{\ell \mapsto v}{\mathsf{wp}\ \texttt{Xchg}\ \ell\, v'\ \{w.\, w = v * \ell \mapsto v'\}}* & \dfrac{\mathsf{wp}\ e\ \{\mathsf{True}\}}{\mathsf{wp}\ \texttt{fork}\,\{e\}\ \{\mathsf{True}\}}* & \dfrac{\mathsf{wp}\ e\ \big\{v.\, \mathsf{wp}\ K[v]\ \{\Phi\}\big\}}{\mathsf{wp}\ K[e]\ \{\Phi\}}*
\end{array}$$

Fig. 4. The basic rules of separation logic

The goal in Mixtris is then to prove the weakest precondition for the program under consideration, such as the example presented in § 2.4. An excerpt of the standard weakest precondition rules for higher-order concurrent separation logic can be found in Fig. 4, where $\frac{P\ \ Q}{R}*$ is defined as $\vdash (P * Q \mathrel{-\!\!*} R)$ and K is an evaluation context that dictates the right-to-left evaluation order. We elide further details, as the separation logic is fairly common, and refer the interested reader to [18]. The rules are sufficient for verifying a program like the following:

$$\texttt{let}\ \ell = \texttt{ref}\ 42\ \texttt{in}\ \texttt{free}\ \ell$$

The proof follows by symbolic execution via Wp-alloc followed by Wp-free. However, a program like the following is not safe, and consequently cannot be verified:

$$\texttt{let}\ \ell = \texttt{ref}\ 42\ \texttt{in}\ \texttt{fork}\,\{\texttt{free}\ \ell\}\ ;\texttt{free}\ \ell$$

The reason for this is that both threads call **free** ℓ, which violates no use-after-free.

The crux of verifying the example presented in § 2.4 is then evident; we must determine that only the elected leader will ever call **free** ℓ, via the protocols and the weakest precondition rules for mixed-choice communication, presented in the following section.

3.2 The Mixtris Mixed Choice Multiparty Protocols

The Mixtris protocol language allows specifying the message-passing behavior of a channel. A protocol is a tree of messages that may be sent and received on a channel; at each step, there may be a choice of messages to send or receive and the channel may follow a different protocol depending on which action happened. The full grammar of the protocol language is as follows:

$$p, q \in \mathsf{iProto} ::= \mathbf{!}\,[i]\ (\vec{x} : \vec{\tau})\ \langle v \rangle \{P\}.\, p \mid \mathbf{?}[i]\ (\vec{x} : \vec{\tau})\ \langle v \rangle \{P\}.\, p \mid \mathbf{end} \mid p + q \mid \mu x.p$$

The meaning of each of these protocol constructs is:

- $\mathbf{!}\,[i]\ (\vec{x} : \vec{\tau})\ \langle v \rangle \{P\}.\, p$: Providing an instantiation of the binders $\vec{x} : \vec{\tau}$, which v, P and p can depend on, send to i value v, resources P, and then continue with protocol p.

- $\mathbf{?}[i]\ (\vec{x}:\vec{\tau})\ \langle v\rangle\{P\}.\,p$: Provided an instantiation of the binders $\vec{x}:\vec{\tau}$, which v, P and p can depend on, receive from i value v, resources P, and then continue with protocol p.
- **end**: Termination of protocol; the channel endpoint can no longer be used.
- $p + q$: The actions allowed by both protocols are possible, and may be chosen by the channel endpoint user. As such, the choice between these actions is non-deterministic. The operator can be nested, is associative, commutative, and has **end** as its left and right identity, *i.e.*, $p_1 + p_2 + p_3 \equiv (p_1 + p_2) + p_3 \equiv p_3 + (p_1 + p_2) \equiv p_3 + (p_1 + p_2) + \mathbf{end}$. There are no restrictions on the operator, and thus protocols with mutually exclusive choices like $\mathbf{?}[i]\ \langle\texttt{true}\rangle.\,\mathbf{end} + \mathbf{?}[i]\ \langle\texttt{false}\rangle.\,\mathbf{end}$ are valid, but will be excluded by protocol consistency.
- $\mu x.p$: A recursive protocol. The protocol can refer to itself using the name x. The logic also supports recursive protocols with parameters (*i.e.* fixpoints over $A \rightarrow \texttt{iProto}$).

The Mixtris protocol language internalises value-based branching, using the dependent binders, similar to previous work on dependent separation protocols:

$$\&[i]\begin{Bmatrix}\texttt{inl}(\vec{x_1}:\vec{\tau_1})\langle v_1\rangle\{P_1\} \Rightarrow p_1\\ \texttt{inr}(\vec{x_2}:\vec{\tau_2})\langle v_2\rangle\{P_2\} \Rightarrow p_2\end{Bmatrix} \triangleq \begin{array}{l} \mathbf{?}[i](\vec{x}:\vec{\tau_1}+\vec{\tau_2})\\ \quad\langle\texttt{match}\,\vec{x}\,\texttt{with}\ \texttt{inl}\,\vec{x_1} \Rightarrow \texttt{inl}\,v_1;\ \texttt{inr}\,\vec{x_2} \Rightarrow \texttt{inr}\,v_2\ \texttt{end}\rangle\\ \quad\{\texttt{match}\,\vec{x}\,\texttt{with}\ \texttt{inl}\,\vec{x_1} \Rightarrow P_1;\ \texttt{inr}\,\vec{x_2} \Rightarrow P_2\ \texttt{end}\}.\\ \quad\texttt{match}\,\vec{x}\,\texttt{with}\ \texttt{inl}\,\vec{x_1} \Rightarrow p_1;\ \texttt{inr}\,\vec{x_2} \Rightarrow p_2\ \texttt{end}\end{array}$$

We omit $\langle v\rangle$, whenever $v := ()$, and $\{P\}$, whenever $P :=$ True. While we do not use the dependent binders for the protocol examples in this section, they are imperative to the verification of the Chang and Roberts ring leader election, presented in §4.1.

The protocols for the example program in §2.4 can be defined as follows:

$$\begin{aligned} p_A &:= (\mathbf{!}\,[B]\,.\,\mathbf{end}) + (\mathbf{?}[C]\ \{P\}.\,\mathbf{end})\\ p_B &:= (\mathbf{!}\,[C]\,.\,\mathbf{end}) + (\mathbf{?}[A]\ \{P\}.\,\mathbf{end})\\ p_C &:= (\mathbf{!}\,[A]\,.\,\mathbf{end}) + (\mathbf{?}[B]\ \{P\}.\,\mathbf{end})\end{aligned}$$

Notably, each participant non-deterministically sends to the right, or receives from the left. Upon receiving a message, a protocol is given the resources P, here picked as $\ell \mapsto -$.

3.3 The Mixtris Channel Rules

We now cover how we can use the Mixtris protocols by giving weakest precondition rules for the synchronous channels presented in §2.3. The rules are displayed in Fig. 5.

The rule for channel creation Wp-new is identical to Multris, barring the new protocol consistency definition. When we create a new multiparty channel with $n > 0$, we gain ownership of each channel endpoint, captured by the exclusively owned channel endpoint ownership $c_i \rightarrowtail p_i$, for each participant i in a pool of protocols $(p_0, \ldots, p_{n-1})$ that is consistent. To create a channel, we must prove consistency of its protocol pool, as covered in §3.4.

The most interesting part of the rules is how we leverage the *subprotocol relation* $\sqsubseteq$ to (1) express the option-preserving nature of the rules for the uncommitted primitives Wp-try-send and Wp-try-recv, and (2) preserve the original Multris rules for the committed primitives. Notably, the rules sub-choice-l and sub-choice-r let us limit choices, *e.g.*, $p_1 + p_2 \sqsubseteq p_1$. The Wp-try-send and Wp-try-recv rules use this property to state that the current protocol p_1 must have the *choice* of sending or receiving, respectively. Notably, we preserve all the original choices of p_1 in the case of failure, reflecting the uncommitted nature of the primitives. The committed primitives are stated in terms of the fixed send and receive protocols, identically to the corresponding Multris rules. This is made possible by the new subprotocol relation, alongside the rule chan-sub, that lets us weaken

Channel rules:

Wp-new

$$\frac{\text{CONSISTENT}\ (p_0,\ldots,p_{n-1}) \qquad n>0}{\mathsf{wp}\ \textbf{new\_chan}(n)\ \{(c_0,\ldots,c_{n-1}).\ c_0 \rightarrowtail p_0 * \cdots * c_{n-1} \rightarrowtail p_{n-1}\}}{*}$$

CHAN-SUB

$$\frac{c \rightarrowtail p_1 \qquad p_1 \sqsubseteq p_2}{c \rightarrowtail p_2}{*}$$

Wp-send

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

Wp-recv

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

Wp-try-send

$$\frac{c \rightarrowtail q \qquad q \sqsubseteq \mathbf{!}\,[i]\ (\vec{x}:\vec{\tau})\ \langle v\rangle\{P\}.\,p \qquad P[\vec{t}/\vec{x}]}{\mathsf{wp}\ c[i].\textbf{try\_send}(v[\vec{t}/\vec{x}])\ \{b.\ \textbf{if}\ b\ \textbf{then}\ c \rightarrowtail p[\vec{t}/\vec{x}]\ \textbf{else}\ c \rightarrowtail q * P[\vec{t}/\vec{x}]\}}{*}$$

Wp-try-recv

$$\frac{c \rightarrowtail q \qquad q \sqsubseteq \mathbf{?}[i]\ (\vec{x}:\vec{\tau})\ \langle v\rangle\{P\}.\,p}{\mathsf{wp}\ c[i].\textbf{try\_recv}()\ \left\{ov.\ \begin{array}{l} \textbf{match}\ ov\ \textbf{with} \\ \mid \textbf{some}\ w \Rightarrow \exists\vec{t}.\ w = v[\vec{t}/\vec{x}] * c \rightarrowtail p[\vec{t}/\vec{x}] * P[\vec{t}/\vec{x}] \\ \mid \textbf{none} \quad\ \Rightarrow c \rightarrowtail q \\ \textbf{end}. \end{array}\right\}}{*}$$

Subprotocol rules:

SUB-SEND

$$\frac{\forall\vec{x_2}:\vec{\tau_2}, \Phi_2 \mathrel{-\!\!*} \exists\vec{x_1}:\vec{\tau_1}, (v_1 = v_2) * \Phi_1 * \triangleright p_1 \sqsubseteq p_2}{\mathbf{!}\,[i]\ (\vec{x_1}:\vec{\tau_1})\ \langle v_1\rangle\{\Phi_1\}.\,p_1 \sqsubseteq \mathbf{!}\,[i]\ (\vec{x_2}:\vec{\tau_2})\ \langle v_2\rangle\{\Phi_2\}.\,p_2}{*}$$

SUB-CHOICE-L

$$p_1 + p_2 \sqsubseteq p_1$$

SUB-CHOICE-R

$$p_1 + p_2 \sqsubseteq p_2$$

SUB-RECV

$$\frac{\forall\vec{x_1}:\vec{\tau_1}, \Phi_1 \mathrel{-\!\!*} \exists\vec{x_2}:\vec{\tau_2}, (v_1 = v_2) * \Phi_2 * \triangleright p_1 \sqsubseteq p_2}{\mathbf{?}[i]\ (\vec{x_1}:\vec{\tau_1})\ \langle v_1\rangle\{\Phi_1\}.\,p_1 \sqsubseteq \mathbf{?}[i]\ (\vec{x_2}:\vec{\tau_2})\ \langle v_2\rangle\{\Phi_2\}.\,p_2}{*}$$

SUB-CHOICE-MONO

$$\frac{p_1 \sqsubseteq p_1' \qquad p_2 \sqsubseteq p_2'}{p_1 + p_2 \sqsubseteq p_1' + p_2'}{*}$$

Fig. 5. The Mixtris rules for multiparty message passing concurrency

the protocol *before* applying the corresponding rules. They are oblivious to the possibility that the protocol may have originally been a mixed choice protocol.

Similar to Multris, in both of the rules for sending, we provide an instantiation $\vec{t}$ of the protocol binders, and the resources P given that instantiation. If the send succeeds (as always is the case for the committed primitive), the channel endpoint ownership is updated to the tail of the protocol for the given binders. If the send fails, we preserve the original protocol along with the resources. Conversely, in the rules for receiving, we instead obtain an instantiation of the binders along with the resources.

With these rules we can prove the weakest preconditions for the program shown in §2.4, given the protocols shown in §3.2. Apart from the proof of protocol consistency which will be addressed in the following section, the proof follows as a symbolic execution of the program. In particular,

$$\frac{\textsc{present}\ \vec{p} \qquad \textsc{dual}\ \vec{p}}{\textsc{consistent}\ \vec{p}}* \qquad\qquad \frac{\forall i, j, (a[j]\ (\vec{x}:\vec{\tau})\ \langle v\rangle\{P\}.\, p') \in \vec{p}_i.\ j \in \vec{p}}{\textsc{present}\ \vec{p}}*$$

$$\frac{\begin{array}{c}\forall i, j, (!\,[j]\ (\vec{x}_1 : \vec{\tau}_1)\ \langle v_1\rangle\{P_1\}.\, p_1) \in \vec{p}_i, (?[i]\ (\vec{x}_2 : \vec{\tau}_2)\ \langle v_2\rangle\{P_2\}.\, p_2) \in \vec{p}_j. \\ \forall(\vec{x}_1 : \vec{\tau}_1).\, P_1 \mathrel{-\!\!*} (\exists(\vec{x}_2 : \vec{\tau}_2).\, v_1 = v_2 \ * \ P_2 \ * \ \triangleright \textsc{consistent}\ (\vec{p}[p_1/i][p_2/j]))\end{array}}{\textsc{dual}\ \vec{p}}*$$

Fig. 6. Protocol consistency rules

we first prove the following Hoare triple rule for send_recv:

$$\begin{array}{l}\{c \rightarrowtail (!\,[i]\ (\vec{x}_1 : \vec{\tau}_1)\ \langle v_1\rangle\{P_1\}.\, p_1) + (?[j]\ (\vec{x}_2 : \vec{\tau}_2)\ \langle v_2\rangle\{P_2\}.\, p_2) * P_1[\vec{t}_1/\vec{x}_1]\} \\ \quad \texttt{send\_recv}\ i\ j\ (v[\vec{t}_1/\vec{x}_1]) \\ \left\{ w.\ \begin{array}{l} \textbf{match}\ w\ \textbf{with} \\ \mid \textsf{none} \quad \Rightarrow c \rightarrowtail p_1[\vec{t}_1/\vec{x}_1] \\ \mid \textsf{some}\ x \Rightarrow \exists \vec{t}_2.\, c \rightarrowtail p_2[\vec{t}_2/\vec{x}_2] * P_1[\vec{t}_1/\vec{x}_1] * P_2[\vec{t}_2/\vec{x}_2] \\ \textbf{end} \end{array} \right\}\end{array}$$

Notably, we use Wp-try-send and Wp-try-recv during every iteration of the loop. The postcondition follows directly from the succeeding cases of either. In the case of a loop, we can use Löb induction, as we end up with the initial unchanged channel state, and the initial resources $P_1[\vec{t}_1/\vec{x}_1]$. Given this rule, the remaining proof of threeway_election_example is trivial; the leader is given $\ell \mapsto -$, as per the protocol, and can thus resolve **free** ℓ. All that remains is then to conclude the proof of protocol consistency.

3.4 Protocol Consistency

When creating a new multiparty channel of size n, we must prove that the pool of protocols $(p_0, \ldots, p_{n-1})$ is consistent. Drawing inspiration from Multris, we have to prove a notion of *semantic duality*, where for any possible exchange, the expectations of the receiver must be met by the requirements on the sender. For example, the two protocols:

$$p_i := !\,[j]\ (\ell : \text{Loc})\ \langle \ell\rangle\{\ell \mapsto -\}.\, p \qquad\qquad p_j := ?[i]\ (\ell' : \text{Loc})\ \langle \ell'\rangle\{\ell' \mapsto -\}.\, p$$

We would have to prove: $\forall(\ell : \text{Loc}).\, \ell \mapsto - \mathrel{-\!\!*} \exists(\ell' : \text{Loc}).\, \ell = \ell' * \ell' \mapsto -$. Note that this means the protocols does not have to be syntactically dual. Additionally, we can use existing separation logic resources. For example, given resources $\ell'' \mapsto -$, we can prove consistency of:

$$p_i := !\,[j]\ \langle \ell''\rangle.\, p \qquad\qquad p_j := ?[i]\ (\ell' : \text{Loc})\ \langle \ell'\rangle\{\ell' \mapsto -\}.\, p$$

Which yields the proof obligation: $\ell'' \mapsto - \mathrel{-\!\!*} \forall.\, \text{True} \mathrel{-\!\!*} \exists(\ell' : \text{Loc}).\, \ell' \mapsto -$. For instructive purposes, we include $\forall.\, \text{True} \mathrel{-\!\!*} P \equiv P$ for the quantification over the empty range of binders and the trivial resources of p_i.

Protocol consistency is checked by simulating all possible communication interleavings on the protocol level. The stark contrast is that in the presence of mixed choice, protocols may proceed non-deterministically. We capture this non-determinism in a similar way to the existing non-deterministic interleavings of protocol interactions in Multris. As a result, the change to the protocol consistency rules are relatively minimal; instead of considering all possible (previously single-choice) protocols, we now consider all possible choices of all possible protocols.

$$\begin{array}{c}
\text{Given } P\text{, show consistency of}\\
\begin{array}{rcl}
p_A & := & (\,!\,[B]\,.\,\mathbf{end}) + (?[C]\,\{P\}.\,\mathbf{end})\\
p_B & := & (\,!\,[C]\,.\,\mathbf{end}) + (?[A]\,\{P\}.\,\mathbf{end})\\
p_C & := & (\,!\,[A]\,.\,\mathbf{end}) + (?[B]\,\{P\}.\,\mathbf{end})
\end{array}\\
\begin{array}{c|c|c}
\xrightarrow{p_A \text{ sends to } p_B} & \xrightarrow{p_B \text{ sends to } p_C} & \xrightarrow{p_C \text{ sends to } c_A}\\
\begin{array}{rcl} p_A & := & \mathbf{end}\\ p_B & := & \mathbf{end}\\ p_C & := & -||- \end{array} &
\begin{array}{rcl} p_A & := & -||-\\ p_B & := & \mathbf{end}\\ p_C & := & \mathbf{end} \end{array} &
\begin{array}{rcl} p_A & := & \mathbf{end}\\ p_B & := & -||-\\ p_C & := & \mathbf{end} \end{array}
\end{array}
\end{array}$$

Fig. 7. An example of protocol consistency simulation. −||− means same as initial protocol.

The protocol consistency of protocols is proven using the rules in Fig. 6. The key is in the premise for proving DUAL $\vec{p}$. The first line checks for the possible communications from protocol $\vec{p}_i$ to $\vec{p}_j$, including all of their potential choices, using a simple concept of *protocol inclusion* $p_1 \in p_2$, where:

$$\frac{\text{PROTO-IN-CHOICE}}{p \in (p_1 + p_2) \dashv\vdash p \in p_1 \lor p \in p_2} \qquad \frac{\text{PROTO-IN-REFL}}{p \in p}$$

We use the notation $\forall(!\,[j]\;(\vec{x} : \vec{\tau})\;\langle v\rangle\{P\}.\,p) \in \vec{p}_i.\,P$ to implicitly quantify over all the free variables of the protocol ($\vec{\tau}$, v, P, p), and assume protocol inclusion.

The second line checks consistency of the given exchange, and is directly inherited from Multris. Specifically, it checks that all instantiations of the received variables can be given based on the available variables—the ones given by the sending protocol, including any previously known variables. Additionally, we obtain ownership of any resource given by the sender P_1, and must use them (and any previously owned resource) to satisfy the resource obligation of the receiver P_2. Finally, we must show that the subsequent pool of protocols is consistent. The later modality $\triangleright$ is used to allow Löb induction to be used in proofs of consistency, for recursive protocols. The PRESENT $\vec{p}$ obligation states that all possible communication attempts must be with a participant that is part of the pool of protocols.

We can now consider protocol consistency of `threeway_election_example`, as shown in Fig. 7. For the initial protocols, there are three possible communications, either p_A sends to p_B, p_B sends to p_A, or p_C send to p_A. The simulation has to consider all cases, verify that the interactions of the communications are sound and that the resulting protocols are consistent. Each exchange is identical, and there are no further exchanges, thus we must prove $P \mathrel{-\!\!*} \forall.\,\mathsf{True} \mathrel{-\!\!*} \exists.\,() = () * P * \mathsf{True}$.

The fact that we can rely on existing resources is a property of separation logic. The protocol consistency obligation is a separation logic proposition like any other, and can thus be proven locally, using currently available resources. Intuitively, the protocol consistency then *owns* P until it is given to the receiver. This notion is directly connected to the *implicit resource transfer* idea from Multris, where resources given by a sender can reside in the protocol consistency, and given to a receiver later down the line. We elide further details on protocol consistency, and refer the interested reader to the paper on Multris [12].

4 Mixed Choice Multiparty Verification Benchmark: Leader Elections

We demonstrate the expressive power of Mixtris by verifying the Chang and Robberts leader election algorithms. A simplified version of this algorithm was verified in Multris, which was restricted to one concurrent election, started by a pre-determined process. We show the changes we had to make to their implementation and verification approach to verify a version where any participant may start an election, and multiple elections can happen concurrently.

We additionally verified the leader election protocol by Peters and Yoshida [26] that determines a leader exclusively using the non-determinism of mixed choice. This generally follows as an extension of the `threeway_election_example`, and thus we elide it for brevity.

4.1 Chang and Roberts's Ring Leader Election

The prior work on Multris verified leader agreement and uniqueness for a simplified version of Chang and Roberts's ring leader election algorithm. However, that version had a designated election initiator, and thus did not consider concurrent elections, which naturally occurs from multiple participants trying to start elections at the same time. To this end, we had to make key changes to the implementation and verification. However, given the kinship between Mixtris and the Multris logic, a lot of the verification technique remain unchanged. For the sake of transparency and comparison, we approach the description of the algorithm and proof similar to the Multris paper [12], while colour-coding key novelties.

Chang and Roberts's ring leader election algorithm assumes that n participants, with unique IDs $id_0 \ldots id_{(n-1)}$ are arranged in a ring. Every process i receives messages from counter-clockwise participant $(i-1)\%n$ and sends messages to clockwise participant $(i+1)\%n$. The algorithm deterministically elects the participant with the highest numerical ID as the leader. Every participant i is considered participating or not (denoted b_i), where participation happens once a message is exchanged by the participant. During an election, two types of messages will be exchanged: $\mathsf{election}(k)$ and $\mathsf{elected}(k)$. When a message $\mathsf{election}(k)$ is received by a participant i, it is compared with its ID id_i:

(1.1) If $k > id_i$, send $\mathsf{election}(k)$, else
(1.2) If $k = id_i$, we are elected, send $\mathsf{elected}(id_i)$, else
(1.3) If $k < id_i$ and $b_i := \texttt{false}$, we send $\mathsf{elected}(id_i)$, else
(1.4) If $k < id_i$ and $b_i := \texttt{true}$, we do nothing

Upon receiving $\mathsf{elected}(k)$ participant i compares it to its ID :

(2.1) If $k = id_i$, terminate by returning k, else
(2.2) If $k \neq id_i$, send $\mathsf{elected}(k)$ and terminate by returning k.

Any participant i such that $b_i = \texttt{false}$ can start an election by sending $\mathsf{election}(id_i)$.

The intuition of the algorithm is that every participant will eventually observe a message $\mathsf{election}(k)$, and either rejects (replacing) or accepts (forwarding) it. Thus, one participant will eventually receive their own id, meaning that all participants have accepted them as leader. They then notify all participants by passing around the $\mathsf{elected}(k)$ message. Concurrent elections are stopped whenever they reach a rejecting participant that is already participating, as they know a strictly better election is already in progress.

In the rest of this section, we present the verification of leader uniqueness and agreement of the algorithm. We do so with an implementation and specification of the algorithm with 3 participants.

Implementation. We encode $\mathsf{election}(i)$ and $\mathsf{elected}(i)$ as $\texttt{inl}\, i$ and $\texttt{inr}\, i$, respectively. We write i_l and i_r for the left and right neighbours of participant i. They are defined as $i_l := (i+1)\%n$ and $i_r := (i-1)\%n$, for the given ring size n. For each process, the algorithm is split into two phases, a pre-participation phase (initial), and a participation phase. The program is depicted in Fig. 8.

(1) In the pre-participation phase (implemented as `cre_init_process`), the process is allowed to start elections. This is implemented by alternating via uncommitted sends and receives. Once a message is sent or received, the processes enters the participation phase. Any received message is processed according to the algorithm outline above. We can rule out receiving `elected`(i) messages, as a leader cannot be elected without the involvement of all processes.

```
cre_init_process $c\ i \triangleq$
   match send_recv $c\ i_l\ i_r$ (inl $i$) with
   | none          ⇒ cre_process $c\ i$
   | some(inl $i'$) ⇒ if $i < i'$ then $c[i_l]$.send(inl $i'$); cre_process $c\ i$          (1.1)
                     else if $i = i'$ then $c[i_l]$.send(inr $i$); cre_process $c\ i$     (1.2)
                     else $c[i_l]$.send(inl $i$); cre_process $c\ i$                      (1.3)
   | some(inr $i'$) ⇒ assert(false)
   end
cre_process $c\ i \triangleq$
   match $c[i_r]$.recv() with
   | inl $i'$ ⇒ if $i < i'$ then $c[i_l]$.send(inl $i'$); cre_process $c\ i$         (1.1)
               else if $i = i'$ then $c[i_l]$.send(inr $i$); cre_process $c\ i$    (1.2)
               else cre_process $c\ i$                                             (1.4)
   | inr $i'$ ⇒ if $i = i'$ then $i'$                                               (2.1)
               else $c[i_l]$.send(inr $i'$); $i'$                                   (2.2)
   end
```

Fig. 8. cre_process implementation for Chang and Roberts's leader election

(2) In the participation phrase (implemented as `cre_process`), the process awaits messages from its right neighbour. Once received, it checks whether it is an election or elected message, and proceeds according to the algorithm outline above.

Compared to the simplified implementation verified in Multris, we add the initial mixed-choice process, that each participant enters, allowing them to start elections. Additionally, since all participants (besides the initiator) was inherently non-participating in Multris, the reused process implementation now properly reflect case (1.4), where processes do not forward messages.

Leader uniqueness program. We verify leader uniqueness by verifying the following program:

```
cre_leader_prog $n \triangleq$
   let $\ell$ = ref 42 in
   let $(c_0, \ldots, c_{n-1})$ = new_chan($n$) in
   For($i = 0 \ldots (n-1)$) { fork { let $i'$ = cre_init_process $c_i\ i$ in
                                       if $i' = i$ then free $\ell$ else () } }
```

We use the same idea as for the Peters and Yoshida's leader election, which is to allocate a reference ℓ and make the elected leader deallocate it.

Verification of leader uniqueness. We define the election protocol for each participant as:

```
cre_init_process_prot $(i : \mathbb{N})\ (P : \text{iProp})\ (p : \mathbb{N} \rightarrow \text{iProto}) : \text{iProto} \triangleq$
   ! $[i_l]$ ⟨inl $i$⟩. cre_process_prot $i\ P\ p$ +
   ? $[i_r]$ $(i' : \mathbb{N})$ ⟨inl $i'$⟩. { if $i < i'$ then ! $[i_l]$ ⟨inl $i'$⟩. cre_process_prot $i\ P\ p$        (1.1)
                                     else if $i = i'$ then ! $[i_l]$ ⟨inr $i$⟩. cre_process_prot $i\ P\ p$   (1.2)
                                     else ! $[i_l]$ ⟨inl $i$⟩. cre_process_prot $i\ P\ p$ }                 (1.3)
```

$$\texttt{cre\_process\_prot}\ (i : \mathbb{N})\ (P : \mathsf{iProp})\ (p : \mathbb{N} \to \mathsf{iProto}) : \mathsf{iProto} \triangleq \mu rec.$$
$$\&[i_r]\left\{\begin{array}{lll} \mathtt{inl}(i' : \mathbb{N})\langle i' \rangle & \Rightarrow \mathtt{if}\ i < i'\ \mathtt{then}\,!\,[i_l]\ \langle \mathtt{inl}\ i' \rangle .\ rec & (1.1) \\ & \ \mathtt{else\ if}\ i = i'\ \mathtt{then}\,!\,[i_l]\ \langle \mathtt{inr}\ i \rangle .\ rec & (1.2) \\ & \ \mathtt{else}\ rec & (1.4) \\ \mathtt{inr}(i' : \mathbb{N})\langle i' \rangle \{ i = i' \twoheadrightarrow P \} & \Rightarrow \mathtt{if}\ i = i'\ \mathtt{then}\ p\ i' & (2.1) \\ & \ \mathtt{else}\,!\,[i_l]\ \langle \mathtt{inr}\ i' \rangle .\ p\ i' & (2.2) \end{array}\right\}$$

The protocol (including novelties) corresponds directly to the implementation. Similar to prior examples, the resources P are given to the leader, as captured by $i = i' \Rightarrow P$ in `proc_ack_prot`. The binder $(i : \mathbb{N})$ is used to keep track of exchanged ids, which is crucial for verifying that the resources P are only given to the elected leader (by invalidating the condition $i = i'$, for all non-leaders). In the case of `cre_leader_prog` we use $P := \ell \mapsto -$. We also add a continuation p that depends on the elected leader, that we used for leader agreement. For this example, we set it to $(\lambda n.\mathbf{end})$.

To link the protocol with the program we show the following Hoare triple, which follows directly from the use of the Mixtris rules for committed and uncommitted primitives:

$$\{c \rightarrowtail \texttt{cre\_init\_process\_prot}\ i\ P\ p\}\ \texttt{cre\_init\_process}\ c\ i\ \{i'.\ c \rightarrowtail p\ i' * (i = i' \twoheadrightarrow P)\}$$

The final step of verifying `ring_ref_prog` is proving the consistency of the pool of protocols:

$$c_i \rightarrowtail \texttt{cre\_init\_process\_prot}\ i\ P\ (\lambda n.\mathbf{end})\quad \text{for each } i \in [0, \ldots, n]$$

The proof of consistency uses a brute-force simulation, executing all the possible communications paths and shows that ultimately the resource $\ell \mapsto 42$ is owned exclusively by the elected participant. The algorithm uses a lot of non-determinism with the choice to launch an election and the possibility to have multiple elections at the same time, which result in a lot of cases that should be considered in the proof. The brute-force approach makes this proof difficult, and so we limit ourselves to proving consistency of 3 participants. With the above protocol system we verify the top-level program for 3 participants:

$$\{\mathsf{True}\}\ \texttt{ring\_ref\_prog}\ 3\ \{\mathsf{True}\}$$

This Hoare triple guarantees that the program is safe to execute via our adequacy theorem Theorem 3.1, thus certifying that the algorithm implementation achieves leader uniqueness.

Verification of leader agreement. To verify leader agreement, we follow the same approach to extending the leader uniqueness as in Multris. We first allocate a separate binary channel to a central coordinator who will receive the ID of the leader from the leader itself, after which point every participant sends the ID of the participant they think was elected. To facilitate this, the channel endpoint to the central coordinator is passed around the ring, described by a protocol used for the protocol continuation p of `cre_process_prot`. The coordinator tests that all the IDs it receives are identical, and crashes in case of failure. The addition of multiple elections (facilitated via mixed choice) pose no novelty over the delta between the leader uniqueness and agreement proofs presented in the prior work on Multris, and so we elide further details for brevity sake. The full proof can be found in our accompanying artifact [2].

5 Model and Soundness

In this section we explain how the Mixtris adequacy and rules were proven sound. Firstly, the Mixtris adequacy theorem Theorem 3.1 is a direct instance of the Iris adequacy theorem, as our implementations are achieved as shallow embedding on top of the HeapLang language. We thus focus on how the implementations were verified w.r.t. their rules. We first present the verification of atomic specifications for the synchronisation cells (§5.1). We then present the so-called Mixtris Ghost Theory (§5.2), which is a language-agnostic reasoning mechanism for mixed choice message passing.

$$\textsc{Wp-new-sync}\qquad \texttt{wp new\_sync ()}\left\{c.\ \begin{array}{l}\texttt{is\_sync\_cell\_put}\ c\ \Phi\ P\ *\\ \texttt{is\_sync\_cell\_get}\ c\ \Phi\ P\end{array}\right\}$$

$$\textsc{Wp-sync-put}\qquad \frac{\texttt{is\_sync\_cell\_put}\ c\ \Phi\ P \qquad \Phi\ v}{\texttt{wp sync\_put}\ c\ v\ \{\texttt{is\_sync\_cell\_put}\ c\ \Phi\ P\ *\ P\}}{}^{*}$$

$$\textsc{Wp-sync-get}\qquad \frac{\texttt{is\_sync\_cell\_get}\ c\ \Phi\ P \qquad (\forall w.\ \triangleright\Phi\ w\ \mathrel{-\!\!*}\ \mathrel{|\!\!\Rrightarrow}\triangleright\mathrel{|\!\!\Rrightarrow}\triangleright P\ *\triangleright\triangleright\Phi_2\ w)}{\texttt{wp sync\_get}\ c\ \{w.\ \texttt{is\_sync\_cell\_get}\ c\ \Phi\ P\ *\ \Phi_2\ w\}}{}^{*}$$

$$\textsc{Wp-sync-try-put}\qquad \frac{\texttt{is\_sync\_cell\_put}\ c\ \Phi\ P \qquad \Phi\ v \qquad R' \qquad (\triangleright\Phi\ v * R' \mathrel{-\!\!*}\ \mathrel{|\!\!\Rrightarrow}\triangleright\mathrel{|\!\!\Rrightarrow}\triangleright R)}{\texttt{wp sync\_try\_put}\ c\ v\ \{b.\ \texttt{is\_sync\_cell\_put}\ c\ \Phi\ P * \textbf{if}\ b\ \textbf{then}\ P * R'\ \textbf{else}\ R\}}{}^{*}$$

$$\textsc{Wp-sync-try-get}\qquad \frac{\texttt{is\_sync\_cell\_get}\ c\ \Phi\ P \qquad R \qquad (\forall w.\ \triangleright\Phi\ w * R \mathrel{-\!\!*}\ \mathrel{|\!\!\Rrightarrow}\triangleright\mathrel{|\!\!\Rrightarrow}\triangleright(P\ *\triangleright\Phi_2))}{\texttt{wp sync\_try\_get}\ c\ ()\left\{ov.\ \begin{array}{l}\texttt{is\_sync\_cell\_get}\ c\ \Phi\ P\ *\\ \textbf{match}\ ov\ \textbf{with some}\ w \Rightarrow \Phi_2\ w;\ \textbf{none} \Rightarrow R\ \textbf{end}\end{array}\right\}}{}^{*}$$

Fig. 9. Specifications of synchronisation cells

We then present how the synchronisation cell specifications, together with the aforementioned ghost theory, are used to verify the channel rules of Mixtris (§ 5.3). Finally, we discuss how we defined the mixed choice protocols, consistency relation, subprotocol relation, and ghost theory tokens, and how we validated the Mixtris Ghost Theory on top of them (§ 5.4).

5.1 Synchronisation Cell Specification and Verification

The implementation of the binary synchronisation cells is given in § 2.2. In this section we show how we verified specifications for them. The synchronisation cell has two distinct endpoints; a getter and a putter. The point of the synchronisation cell specifications is to allow the transfer of resources between the putter and getter, in *both* directions, alongside a synchronous value transfer. Intuitively, the putter can transfer resources $\Phi\ v$ when putting in the value v. Conversely, the getter can obtain the resources, and in return transfer resources P back, alongside the acknowledgment. The crux is that P may be derived from $\Phi\ v$, alongside atomically available resources, during the atomic step where the getter takes the value out of the synchronisation cell. We capture this idea with the specifications for the synchronisation cell operations, as seen in Fig. 9.

The rules include details related to so-called atomic updates [29], that explicate how we may leverage invariable resources—*e.g.*, for deriving P from Φ—only available during atomic transitions, such as **Xchg**. By virtue of the higher-order nature of Iris, these resources are often guarded to preserve soundness. Formally, the atomic updates are defined in terms of ghost updates $\mathrel{|\!\!\Rrightarrow}$ and laters $\triangleright$. $\mathrel{|\!\!\Rrightarrow} P$ states that we can interact with ghost state, such as accessing atomically available resources, before proving P. $\triangleright P$ guards P, asserting that P is only available after one step of computation. For brevity sake, we elide further details about these, but comment on their necessity in §5.3. We further elide details regarding so-called "masks" that prevent the unsound repeated access to atomically available resources. The complete rules can be found in our accompanying artifact [2].

Wp-new-sync: The abstract predicates $\texttt{is\_sync\_cell\_put}\ c\ \Phi\ P$ and $\texttt{is\_sync\_cell\_get}\ c\ \Phi\ P$ assert exclusive permission to operate as the putter and getter, respectively, and are obtained when a new synchronisation cell is create via Wp-new-sync, where the transferred resources Φ and P can be picked freely.

Wp-sync-put: A committed put can be resolved by giving up the dictated resources Φ for the exchanged value v, and in exchange the acknowledgement resources P are obtained.

Wp-sync-get: A committed set can be resolved by proving that the put resources $\Phi\ v$, can be atomically updated into the acknowledgement resource P, and some leftover resources Φ_2, which are obtained by the getter upon completion.

Wp-sync-try-put: On top of the requirements for sending, we must provide additional resources R' and a rollback atomic update $(\triangleright \Phi\ v * R' \mathrel{-\!\!*} \Rrightarrow \triangleright \Rrightarrow \triangleright R)$, which lets us recover initial resources R, originally used to satisfy the sent resources $\Phi\ v$, in the case of failure. In the case of success, we get the acknowledgement resources P, and the unused additional resources R'.

Wp-sync-try-get: We similarly want to recover the original resources, in case of failure. We achieve this through additional resources R, which can be used, alongside the received resources $\Phi\ v$, when deriving P and Φ_2. If the receive succeeds the postconditions are the same as for Wp-sync-get, otherwise we get the original resources R back.

Verification of synchronisation cell specifications. To verify the synchronisation cell we must first define the abstract predicates $\texttt{is\_sync\_cell\_put}\ c\ \Phi\ P$ and $\texttt{is\_sync\_cell\_get}\ c\ \Phi\ P$. We start by considering the invariant describing the synchronisation cell:

$$\texttt{sync\_cell\_inv}\ \gamma_t\ \gamma_v\ c\ \Phi\ P \triangleq \begin{array}{lr} c \mapsto \textbf{None} * \boxed{\texttt{tok}}^{\gamma_t} & \vee \quad (1) \\ \exists v, c \mapsto \textbf{Some}\ v * (\Phi\ v) * \boxed{\bullet_E\ v}^{\gamma_v} & \vee \quad (2) \\ \exists v, c \mapsto \textbf{None} * P * \boxed{\bullet_E\ v}^{\gamma_v} & (3) \end{array}$$

A synchronisation cell has three possible states. State (1) is the idle state, where no exchange is currently happening, meaning that the underlying reference is empty, captured by $\ell \mapsto$ **none**. State (2) is the transmitted state, where a value v has been put in the reference, captured by $\ell \mapsto$ **some** v, alongside the resources associated with the value $\Phi\ v$. State (3) is the acknowledged state, where the value has been taken out, captured by $\ell \mapsto$ **none**, and the resources have been replaced by the returned resources P.

To track the state of the synchronisation cell, we use ghost tokens $\boxed{\texttt{tok}}^{\gamma_t}$ and $\boxed{\bullet_E\ v}^{\gamma_v}$. The first token $\boxed{\texttt{tok}}^{\gamma_t}$ is used to distinguish between state (1) and (3). It is an exclusive token, meaning that $\boxed{\texttt{tok}}^{\gamma_t} * \boxed{\texttt{tok}}^{\gamma_t} \mathrel{-\!\!*}$ False. As such, if we own the token, we can deduce that the invariant cannot be in state (1). The second token $\boxed{\bullet_E\ v}^{\gamma_v}$ is used to keep track of the value being transferred, after it is put under an existential quantifier. It is an agreement token, which in combination with its counterpart $\boxed{\circ_E\ w}^{\gamma_v}$ satisfy $\boxed{\bullet_E\ v}^{\gamma_v} * \boxed{\circ_E\ w}^{\gamma_v} \mathrel{-\!\!*} v = w$ and $\boxed{\bullet_E\ v}^{\gamma_v} * \boxed{\circ_E\ w}^{\gamma_v} \mathrel{-\!\!*} \Rrightarrow \boxed{\bullet_E\ v'}^{\gamma_v} * \boxed{\circ_E\ v'}^{\gamma_v}$ That is, if we own the token, we can deduce the exact value stored in the cell, and update the ghost tokens in between transactions. Given the invariant definition, we can define the abstract predicate for our synchronisation cell as follows:

$$\begin{array}{l} \texttt{is\_sync\_cell}\ b\ c\ (\Phi : \texttt{val} \to \texttt{iProp})\ (P : \texttt{iProp}) \triangleq \\ \quad \exists \gamma_t, \gamma_v.\ \boxed{\texttt{sync\_cell\_inv}\ \gamma_t\ \gamma_v\ c\ \Phi\ P} * \textbf{if}\ b\ \textbf{then}\ \exists v.\ \boxed{\bullet_E\ v}^{\gamma_v} * \boxed{\circ_E\ v}^{\gamma_v} \end{array}$$

The predicate distinguishes sending and receiving permissions by b; we write $\texttt{is\_sync\_cell\_put} \triangleq \texttt{is\_sync\_cell}\ \textbf{true}$ and $\texttt{is\_sync\_cell\_get} \triangleq \texttt{is\_sync\_cell}\ \textbf{false}$. We use $\boxed{P}$ to assert that the synchronisation cell invariant is truly an invariant; it must remain unchanged in between all program steps. By virtue of the disjunct states, the inner state of the synchronisation cell can still

Grammar:

$$t, u, P, Q, p ::= \ \ldots \mid \mathsf{prot\_ctx}\ \chi\ n \mid \mathsf{prot\_own}\ \chi\ i\ p \mid \ldots$$

Rules:

PROTO-ALLOC

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

PROTO-LE

$$\frac{\mathsf{prot\_own}\ \chi\ i\ p_1 \qquad p_1 \sqsubseteq p_2}{\mathsf{prot\_own}\ \chi\ i\ p_2}{*}$$

PROTO-STEP

$$\frac{\begin{array}{c} \mathsf{prot\_ctx}\ \chi\ n \qquad P_1[\vec{t}_1/\vec{x}_1] \\ \mathsf{prot\_own}\ \chi\ i\ (\mathbf{!}\,[j]\ (\vec{x}_1 : \vec{\tau}_1)\ \langle v_1 \rangle \{P_1\}.\ p_1) \qquad \mathsf{prot\_own}\ \chi\ j\ (\mathbf{?}[i]\ (\vec{x}_2 : \vec{\tau}_2)\ \langle v_2 \rangle \{P_2\}.\ p_2) \end{array}}{\begin{array}{c} \Rrightarrow \triangleright \exists (\vec{t}_2 : \vec{\tau}_2).\ \mathsf{prot\_ctx}\ \chi\ n * \mathsf{prot\_own}\ \chi\ i\ (p_1[\vec{t}_1/\vec{x}_1]) * \mathsf{prot\_own}\ \chi\ j\ (p_2[\vec{t}_2/\vec{x}_2]) * \\ (v_1[\vec{t}_1/\vec{x}_1]) = (v_2[\vec{t}_2/\vec{x}_2]) * P_2[\vec{t}_2/\vec{x}_2] \end{array}}{*}$$

PROTO-VALID

$$\frac{\mathsf{prot\_ctx}\ \chi\ n \qquad \mathsf{prot\_own}\ \chi\ i\ p}{i < n}{*}$$

PROTO-VALID-PRESENT

$$\frac{\mathsf{prot\_ctx}\ \chi\ n \qquad \mathsf{prot\_own}\ \chi\ i\ (a[j]\ (\vec{x} : \vec{\tau})\ \langle v \rangle \{P\}.\ p)}{\triangleright j < n}{*}$$

Fig. 10. The Mixtris Ghost Theory

change, and we can track it via the ghost tokens. The predicate governs the ghost names γ_t and γ_v, for the ghost state used by the invariant. For the sender (where $b = \mathtt{true}$), the predicate governs the ghost tokens $\boxed{\bullet_E\ v}^{\gamma_v} * \boxed{\circ_E\ v}^{\gamma_v}$ used for tracking the state of the currently exchanged value.

With the abstract predicates in hand, the verification of the rules is relatively straightforward Iris reasoning. Most importantly is it to observe how the preconditions of the rules permit the necessary transitions between the invariant states. A putter first goes from (1) to (2), by using the resources given by the rule $\Phi\ v$, while keeping track of the value v. It then goes from either (2) or (3) to (1), depending on whether the getter has taken the value out. In case of (2), the original resources are still available, and we thus use them to either reattempt or abort safely, in the case of committed and uncommitted put, respectively. A getter either observes (1) and does nothing, or goes from (2) to (3). In the latter case it takes out the resources $\Phi\ v$, updates them to P and $\Phi_2\ v$ using the given transformation, puts back P, and returns with $\Phi_2\ v$.

5.2 Mixtris Ghost Theory

We now give an overview of the Mixtris Ghost Theory, shown in Fig. 10. A ghost theory is effectively a state transition system, reflected into separation logic. In this case, we capture the allowed transitions in our mixed choice multiparty protocols. The rule PROTO-ALLOC states that we can allocate a new ghost theory, consisting of the prot_ctx $\chi\ n$ and prot_own $\chi\ i\ p$, associated by the identifier χ. prot_ctx $\chi\ n$ act as an authority, asserting that the protocol pool governed by χ is consistent. The most important rule is PROTO-STEP, that captures the essence of the synchronous mixed choice transitions. Specifically, it captures that we must update the sender and receiver *synchronously*; we must have the tokens for both the sender and receiver.

It is worth noting that the ghost theory is *syntactically identical* to the one presented in Hinrichsen et al. [12]. Similar to the Multris rules, this is an earned effort, that is achieved through the subprotocol relation, that allow us to turn mixed choice protocols into the synchronising choice.

Instructively, the following rule is a direct consequence of PROTO-STEP and PROTO-LE:

$$\frac{\begin{array}{c}\text{PROTO-STEP-ALT}\\ \texttt{prot\_ctx}\ \chi\ n \qquad P_1[\vec{t_1}/\vec{x_1}] \qquad \texttt{prot\_own}\ \chi\ i\ q_1 \qquad \texttt{prot\_own}\ \chi\ j\ q_2 \\ q_1 \sqsubseteq (\mathbf{!}\,[j]\ (\vec{x_1}:\vec{\tau_1})\ \langle v_1\rangle\{P_1\}.\ p_1) \qquad q_2 \sqsubseteq (\mathbf{?}[i]\ (\vec{x_2}:\vec{\tau_2})\ \langle v_2\rangle\{P_2\}.\ p_2)\end{array}}{\begin{array}{c}\Rrightarrow \triangleright \exists(\vec{t_2} : \vec{\tau_2}).\ \texttt{prot\_ctx}\ \chi\ n * \texttt{prot\_own}\ \chi\ i\ (p_1[\vec{t_1}/\vec{x_1}]) * \texttt{prot\_own}\ \chi\ j\ (p_2[\vec{t_2}/\vec{x_2}]) * \\ (v_1[\vec{t_1}/\vec{x_1}]) = (v_2[\vec{t_2}/\vec{x_2}]) * P_2[\vec{t_2}/\vec{x_2}]\end{array}}*$$

5.3 Verification of Mixtris Channel Specifications

We now describe how we verify the implementation of the Mixtris communication channels, whose implementation was given in §2.3.

The main verification effort is to define the propositions with which we instantiate the synchronisation cell. The intuition is that we will use $\Phi\ v$ to transfer the token of the sender to the receiver, who then updates it using PROTO-STEP, alongside their own token, and using P to transfer the updated token of the sender back to the sender. We start by defining the propositions `proto_pre` and `proto_post` which will take the roles of Φ and P respectively.

$$\begin{aligned}&\texttt{proto\_pre}\ \gamma\ \gamma_{E1}\ \gamma_{E2}\ \gamma_{E3}\ i\ j\ \triangleq\\ &\quad \lambda v.\ \exists q, q', p.\ \texttt{prot\_own}\ \gamma\ i\ q * q \sqsubseteq q' * q' \sqsubseteq \mathbf{!}\,[j]\ \langle v\rangle.\ p *\\ &\qquad\qquad \boxed{\bullet_E\ (\texttt{Next}\ q)}^{\gamma_{E1}} * \boxed{\bullet_E\ (\texttt{Next}\ q')}^{\gamma_{E2}} * \boxed{\bullet_E\ (\texttt{Next}\ p)}^{\gamma_{E3}}\end{aligned}$$

$$\begin{aligned}&\texttt{proto\_post}\ \gamma\ \gamma_{E1}\ \gamma_{E2}\ \gamma_{E3}\ i \triangleq\\ &\quad \exists q, q', p.\ \texttt{prot\_own}\ \gamma\ i\ p *\\ &\qquad\qquad \boxed{\bullet_E\ (\texttt{Next}\ q)}^{\gamma_{E1}} * \boxed{\bullet_E\ (\texttt{Next}\ q')}^{\gamma_{E2}} * \boxed{\bullet_E\ (\texttt{Next}\ p)}^{\gamma_{E3}}\end{aligned}$$

Here, q is the original protocols, and p is the protocol continuation. We use a Multris notion of "pre-satisfied protocols" to preemptively provide the binder instantiations and resources for p, to avoid including them in the definition. To this end, we use an intermediate q', alongside (1) $q \sqsubseteq q'$ and (2) $q' \sqsubseteq \mathbf{!}\,[j]\ \langle v\rangle.\ p$, where (1) preserves the original choices of q, and (2) preserves the resources in case we have to abort. The `prot_own` token states that the original protocol q is updated to p, whenever the exchange succeeds. We then define the channel endpoint ownership $c \rightarrowtail p$:

$$\begin{aligned}c \rightarrowtail p \triangleq \exists \gamma, \vec{\gamma_{E1}}, \vec{\gamma_{E2}}, \vec{\gamma_{E3}}, m, i, n, p'.\ \ulcorner c = (m, i) \urcorner\ *\ &\boxed{\texttt{prot\_ctx}\ \gamma\ n}\\ * \texttt{is\_matrix}\ m\ n\ n\ \{i\}\ \{0, \ldots, n\}&\left(\lambda\ i\ j\ v,\ \begin{array}{l}\texttt{is\_sync\_cell\_send}\ v\\ \quad(\texttt{proto\_pre}\ \gamma\ \vec{\gamma_{E1}}_i\ \vec{\gamma_{E2}}_i\ \vec{\gamma_{E3}}_i\ i\ j)\\ \quad(\texttt{proto\_post}\ \gamma\ \vec{\gamma_{E1}}_i\ \vec{\gamma_{E2}}_i\ \vec{\gamma_{E3}}_i\ i)\end{array}\right)\\ * \texttt{is\_matrix}\ m\ n\ n\ \{0, \ldots, n\}\ \{j\}&\left(\lambda\ i\ j\ v,\ \begin{array}{l}\texttt{is\_sync\_cell\_recv}\ v\\ \quad(\texttt{proto\_pre}\ \gamma\ \vec{\gamma_{E1}}_i\ \vec{\gamma_{E2}}_i\ \vec{\gamma_{E3}}_i\ i\ j)\\ \quad(\texttt{proto\_post}\ \gamma\ \vec{\gamma_{E1}}_i\ \vec{\gamma_{E2}}_i\ \vec{\gamma_{E3}}_i\ i)\end{array}\right)\\ * \triangleright (p' \sqsubseteq p) *\ \texttt{prot\_own}\ \gamma\ i\ p'&\\ * \boxed{\bullet_E\ (\texttt{Next}\ p')}^{\vec{\gamma_{E1}}_i} * \boxed{\bullet_E\ (\texttt{Next}\ p')}^{\vec{\gamma_{E2}}_i} * \boxed{\bullet_E\ (\texttt{Next}\ p')}^{\vec{\gamma_{E3}}_i}&\\ * \boxed{\circ_E\ (\texttt{Next}\ p')}^{\vec{\gamma_{E1}}_i} * \boxed{\circ_E\ (\texttt{Next}\ p')}^{\vec{\gamma_{E2}}_i} * \boxed{\circ_E\ (\texttt{Next}\ p')}^{\vec{\gamma_{E3}}_i}&\end{aligned}$$

Here, $c = (m, i)$ means that the channel endpoint c is the i-th participant of the matrix m. The `is_matrix` $m\ n\ n\ is\ js\ \Phi$ proposition asserts that m is a square matrix of size $n \times n$, where each cell (i, j) satisfy $\Phi\ i\ j$. The arguments is and js respectively describe the set of rows and columns for which we have ownership of the resources $\Phi\ i\ j$. We use the proposition to describe participant i's ownership of the sending and receiving permission of all synchronisation cells in row i and column j, respectively. The propositions $\boxed{\texttt{iProto\_ctx}\ \gamma\ n}$ and `iProto_own` $\gamma\ i\ p'$ state that there exists a consistent pool of protocols of size n such that the i-th protocol of the pool is p'. The proposition $\triangleright (p' \sqsubseteq p)$ internalise the subprotocol relation, so we can locally update the channel ownership. The remaining assertions is ghost state ownership used in `proto_pre` and `proto_post` and their fragmental counterparts, used during a send to keep track of the various parts of the protocol state.

Proving the channel specifications of Fig. 5 is relatively straightforward, through a combination of reusing the verification pattern used by Hinrichsen et al. [12], and the synchronisation cell specifications from Fig. 9. The most notable challenge is to properly resolve all of the laters $\triangleright$ incurred by the higher-order ghost state and invariants. The crux of solving this challenge is in the design of the synchronisation cell rules, to properly expose the atomic updates they permit.

5.4 Model of Mixed Choice Multiparty Dependent Separation Protocols

In this section we cover how we defined and validated the protocols, consistency relation, subprotocol relation, and ghost theory.

Mixed Choice Multiparty Dependent Separation Protocols. The mixed choice multiparty protocols are defined as a variation of the multiparty protocols of Hinrichsen et al. [12]. Notably, we use a similar continuation-passing style for the protocol tails, to leverage the guarded recursion of Iris propositions. The key change is to let protocols be lists of potential exchanges:

$$\begin{aligned}
\mathsf{action} &::= \mathbf{send} \mid \mathbf{recv} \\
\mathsf{iProto} &\cong \mathsf{List}\,(\mathsf{action} \times \mathbb{N} \times (\mathsf{Val} \to \blacktriangleright\mathsf{iProto} \to \mathsf{iProto})) \\
\mathbf{end} &\triangleq \epsilon \\
\mathbf{!}\,[i]\,(\vec{x}:\vec{\tau})\,\langle v\rangle\{P\}.\,p &\triangleq [(\mathbf{send}, i, (\lambda w, p'.\, \exists \vec{x}:\vec{\tau}.\, (v = w) * P * (p' = \mathsf{next}\ p)))] \\
\mathbf{?}[i]\,(\vec{x}:\vec{\tau})\,\langle v\rangle\{P\}.\,p &\triangleq [(\mathbf{recv}, i, (\lambda w, p'.\, \exists \vec{x}:\vec{\tau}.\, (v = w) * P * (p' = \mathsf{next}\ p)))] \\
p_1 \,+\, p_2 &\triangleq p_1 \cdot p_2
\end{aligned}$$

The sending and receiving protocols are defined as singleton lists, while the terminating protocol is defined as the empty list. We define protocol choice as list concatenation. Note that this means that $p + \mathbf{end} = p$, which is different from prior work [26], where **end** is a distinct case.

Protocol consistency relation. Protocol consistency arise as the lifting of the Hinrichsen et al. [12] consistency definition, to protocols as lists. In particular, we define the new inclusion relation $p \in \vec{p}$ (presented in §3.4), and use it to range over all sending (resp. receiving) protocol choices in the given participant protocols $[(\mathbf{send}, j, \Phi_1)] \in \vec{p}[i]$ (resp. $[(\mathbf{recv}, i, \Phi_2)] \in \vec{p}[j]$). The definitions are then given as follows:

$$\begin{aligned}
\textsc{consistent}\ \vec{p} &\triangleq (\textsc{present}\ \vec{p}) * (\textsc{dual}\ \vec{p}) \\
\textsc{present}\ \vec{p} &\triangleq \forall i, j, a, \Phi.\ [(a, j, \Phi)] \in \vec{p}[i] \mathrel{-\!\!*} j \in \vec{p} \\
\textsc{dual}\ \vec{p} &\triangleq \forall i, j, \Phi_1, \Phi_2.\ i \neq j \mathrel{-\!\!*} [(\mathbf{send}, j, \Phi_1)] \in \vec{p}[i] \mathrel{-\!\!*} [(\mathbf{recv}, i, \Phi_2)] \in \vec{p}[j] \mathrel{-\!\!*} \\
&\qquad \forall v_1, p_1'.\ \Phi_1\ v_1\ (\mathsf{next}\ p_1) \mathrel{-\!\!*} \\
&\qquad\qquad (\exists v_2, p_2'.\ \Phi_2\ v_2\ (\mathsf{next}\ p_2) * \triangleright\,\textsc{consistent}\ (\vec{p}[i := p_1][j := p_2])) \\
p \in ps &\triangleq \exists i.\ \begin{cases} p = \epsilon & \texttt{if}\ ps[i] = \mathbf{none} \\ p = [aj\Phi] & \texttt{if}\ ps[i] = \mathbf{some}(aj\Phi) \end{cases}
\end{aligned}$$

The definition validates the protocol consistency rules shown in Fig. 6 by construction.

Component	**Section(s)**	**LOC**
Synchronisation cell implementation and verification	§2.2, §5.1	248
Protocols and Mixtris Ghost Theory	§5.2, §5.4	1949
Channel implementation and verification	§2.3, §5.3	635
Threeway leader election	§1, §2, §3	108
Peters and Yoshida election	N/A	199
Chang and Roberts election	§4.1	633
Matrix library	N/A	419
Proofmode tactics	§6	710
Total		4901

Table 1. Overview of the Mixtris Rocq mechanisation.

Subprotocol relation. Similar to the consistency relation, the subprotocol relation is a list lifting of the relation from Hinrichsen et al. [12], using the $p_1 \in p_2$ relation:

$$\begin{aligned}
p_1 \sqsubseteq p_2 \triangleq \forall i, a, \Phi_2.\ & [(i, a, \Phi_2)] \in p_2 \mathrel{-\!\!*} \\
& \exists \Phi_1.\ [(i, a, \Phi_1)] \in p_1 \mathrel{-\!\!*}\ * \\
& \quad \textbf{match}\ a\ \textbf{with} \\
& \quad \mid \textbf{send} \Rightarrow \forall v, p_2'.\ \Phi_2\ v\ (\text{next}\ p_2') \mathrel{-\!\!*} \exists p_1'.\ \Phi_2\ v\ (\text{next}\ p_1') * \triangleright\, p_1' \sqsubseteq p_2' \\
& \quad \mid \textbf{recv}\ \Rightarrow \forall v, p_1'.\ \Phi_1\ v\ (\text{next}\ p_1') \mathrel{-\!\!*} \exists p_2'.\ \Phi_2\ v\ (\text{next}\ p_2') * \triangleright\, p_1' \sqsubseteq p_2' \\
& \quad \textbf{end}
\end{aligned}$$

For any singleton protocol of the target protocol p_2, there must exist a corresponding singleton subprotocol in the original protocol p_1. We borrow the remaining notion of the subprotocol relation from prior work, capturing that sending protocols may become stronger, while receiving protocols may become weaker. The subprotocol relation supports the spatial subprotocol framing concept as presented in Hinrichsen et al. [11], further facilitating compositional reasoning between parties.

Ghost theory tokens. With the consistency and subprotocol relation definitions, we can define the ghost theory tokens similarly to the approach taken by Hinrichsen et al. [12]. Notably, we define the tokens prot_ctx χ n and prot_own χ p. The prot_ctx χ n token governs the authoritative view of the consistent state of all protocols. The prot_own χ p token governs a single protocol fragment, explicitly closed under the subprotocol relation.

$$\text{prot\_ctx}\ \chi\ n \triangleq \exists \vec{p}.\ |\vec{p}| = n * \boxed{\bullet \vec{p}}^{\chi} * \triangleright\, \text{CONSISTENT}\ \vec{p}$$

$$\text{prot\_own}\ \chi\ i\ p \triangleq \exists p'.\ \boxed{\circ(i, p')}^{\chi} * \triangleright (p' \sqsubseteq p)$$

These definitions give rise to the ghost theory rules presented in Fig. 10, following trivial ghost state verification effort, as a result of the close relationship between the rules and the protocol consistency and subprotocol relation definitions.

6 Mechanisation

For the mechanisation effort we were able to reuse a lot of the foundation built by Multris. Even so, since we changed the fundamental protocol model, we had to make changes to every single part of the infrastructure (barring the Multris examples, which we inherited directly by re-obtaining the original Multris verification interface). An overview of the mechanisation is seen in Table 1.

The key mechanisation challenges was to design and develop the Iris Proof Mode tactics for (1) resolving the uncommitted message passing primitives in the style of symbolic execution, and (2) automating the majority of the protocol consistency proof.

For (1) we leveraged the design of the uncommitted message passing primitive rules, where the presence of a valid choice is captured in terms of a subprotocol relation. This allowed us to reuse existing Multris infrastructure that search the context for protocols that can be transformed into matching the send/receive via a subprotocol relation. We extended this infrastructure with support for mixed choice, which then searches the protocol left-to-right, to see if any of the choices match the current message passing instruction.

For (2) we leveraged the brute-force tactic of Multris, that finds all synchronising pairs of the protocol pool, and tries to resolve each synchronisation via Rocq's eager variable unification and Iris Proof Mode infrastructure for matching up resource obligation with the resources in the context. Their approach is based on rewriting rules, to avoid breaking abstraction of the protocols. With the addition of mixed choice, this approach was insufficient, and the tactic took too long to tractably verify even the examples presented in this paper. Instead, we now unfold the protocol abstractions, to expose the underlying Rocq lists, which can be simplified using Rocq's own infrastructure, yielding a much faster result. After the tactic has run, we (try to) repack the protocols to preserve the protocol abstraction before presenting remaining proof obligations to the user.

7 Related Work

The paper is primarily concerned with the semantics, implementation, and verification of mixed choice message passing, w.r.t. session protocols. Here, we discuss the related work regarding existing semantics (§7.1), implementations (§7.2), and verification (§7.3) of mixed choice.

7.1 Semantics of Mixed Choice

The concept of mixed choice message passing was first formally considered in the original inception of π-calculus [22], which allowed unrestricted non-determinism between processes $P + Q$.

A landmark result by Palamidessi [23] uncovered that the synchronous π-calculus with mixed choice is strictly more expressive than the asynchronous counterpart. In particular, asynchronous pi-calculus cannot express leader election algorithms. Even so, recovering synchronous mixed choice semantics in asynchronous settings have been studied as the "binary interaction problem" [24, 35]. The general problem relates to the three linearisation points that we observe in §1, where it is crucial that no party accept more than one handshake at a time. Similar to our approach, existing solutions for the binary interaction problem verify "partial synchrony", which is related to our notion of synchrony under assumptions on the scheduler (in their case network) [7, 24].

Finally, mixed choice is often expected to not impose an order between the given choices, to ensure uniform distribution. Similar to our argument about termination, we argue that our approach yields a non-zero non-guaranteed probability that each choice is made, which is in line with the semantics proposed by Palamidessi and Herescu [24].

7.2 Implementations of Mixed Choice

Beyond the work already mentioned in the introduction [27, 30, 31], Berry and Gonthier [4] and Thomsen et al. [34] designed programming languages with mixed choice as a primitive, that generate synchronous machine-level instructions for the mixed choice construction. It would be interesting to further understand the architecture that is necessary to support these instruction sets, and understand the trade-offs between using them versus implementing mixed choice based on more common atomic instructions, such as our use of **Xchg**.

7.3 Verification of Mixed Choice

Protocols for verifying mixed choice message passing primarily focus on synchronous mixed choice. The state-of-the-art result is by Peters and Yoshida [26], who developed a general typing system for a multiparty synchronous mixed choice calculi, and showed how the calculi subsumes the expressivity of different variations of process calculi. The structure of our dependent separation protocols are closely related to their approach, with the key difference that they are dependent. For one, our dependent binders yield an embedded value-dependent branching construct, while value-based branching is inherent to their mixed choice construction, which can range over different labels for the same inputs/outputs from the same correspondent. In addition to crash-freedom, their system allows proving deadlock-freedom. Finally, their type system permits decidable type checking whereas our logic is undecidable, requiring interactive verification effort.

Variants of mixed choice have been uncovered in the context of global session types [5, 16, 21], where a single almost-correct-by-construction global type is given, which is then soundly projected into local types. The global session type approach to multiparty message passing comes with restrictions, and is in the general case unable to model a significant portion of interesting mixed choice programs, and in the specialised case domain specific to certain problems.

Pears et al. [25] present a session type system for asynchronous binary mixed choice, based on timeouts. The asynchrony permits more application domains, such as distributed systems, where the impossibility of atomic synchronicity render synchronous mixed choice inapplicable.

Hamers and Jongmans [10] develops a runtime verification tool for their own domain-specific language Discourje, based on monitors inspired by multiparty session types. Discourje is based on Clojures channels, and thereby their mixed choice semantics. The tool does not have a foundationally verified soundness theorem.

Barbanera and Dezani-Ciancaglini [3] presents an approach to modular decomposition of multiparty session types, in which they separately define and verify the two election stages of the leader election example from Peters and Yoshida [26]. This is an interesting direction, as it may reduce the proof effort regarding protocol consistency, which pose a challenging mechanisation effort in the presence of mixed choice; even when the proof itself is trivial.

8 Future Work

More performant mixed choice semantics. While our mixed choice semantics are almost surely terminating (as discussed in § 1), the performance is not ideal. It would be interesting to pursue a more performant implementation of mixed choice. An immediate avenue for investigation is to try and make individual handshake attempts concurrent, committed, and race-free, by drawing inspiration from Reppy et al. [27].

Mechanisation of mixed choice session types. Hinrichsen et al. [13] mechanised an advanced type system for binary session types, by combining prior work on dependent separation protocols, and the idea of semantically interpreting types in logic. It would be interesting to see if a similar approach could be used to interpret a type system inspired by Peters and Yoshida [26] via Mixtris, to obtain a mechanised type system for mixed choice multiparty message passing.

Liveness and Deadlock freedom. We argue that our approach preserve liveness requirements of mixed choice, *when used correctly*; when participants loop over their prospective choices. However, our logic does not enforce this behaviour. Jacobs et al. [15] verified deadlock freedom alongside functional correctness for a binary message passing system, by adding linearity to their dependent separation protocol-based logic and restricting their semantics to only create channels during thread creation. It would be interesting to investigate if a similar approach could let us enforce the above reuqirement for liveness, and in turn verify deadlock freedom in Mixtris.

Distributed systems. The potential application of the asynchronous mixed choice protocol by Pears et al. [25] in distributed systems may open the doors for functional verification of mixed choice distributed systems. Seeing as non-mixed choice dependent separation protocols have been applied to distributed systems in the past [9], it would be interesting to understand if the asynchronous protocols could be adopted by the Mixtris approach, and in turn be applied to distributed systems, following their discoveries.

Modular proof of protocol consistency. As the idea of modular verification is intimately related to separation logic, it would be interesting to investigate if the work on modular mixed sessions by Barbanera and Dezani-Ciancaglini [3] applies to our mixed choice protocols, and if it could reduce the proof effort pertaining to protocol consistency as a result.

Similarly, it would be interesting to understand if the global session types of prior work [5, 16, 21] could apply in the context of multiparty dependent separation protocols, and if they could reduce the proof obligations regarding protocol consistency. As observed by Hinrichsen et al. [12], the dependent binders of the dependent separation protocols may pose interesting challenges, as soundly scoping the binders is non-trivial.

Data-Availability Statement

The Coq development for this paper can be found in [2].

References

[1] Andrew W. Appel. 2001. Foundational Proof-Carrying Code. In *LICS*. https://doi.org/10.1109/LICS.2001.932501

[2] Anonymous Authors. 2025. Supplementary Material: Rocq Mechanisation of "Mixtris: Mechanised Higher-Order Separation Logic for Mixed Choice Multiparty Message Passing".

[3] Franco Barbanera and Mariangiola Dezani-Ciancaglini. 2025. Modular Multiparty Sessions with Mixed Choice. In *DisCoTec*.

[4] Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.* (1992). https://doi.org/10.1016/0167-6423(92)90005-V

[5] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2012. On Global Types and Multi-Party Session. *LMCS* (2012). https://doi.org/10.2168/LMCS-8(1:24)2012

[6] Ernest Chang and Rosemary Roberts. 1979. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM* 22, 5 (May 1979), 281–283. https://doi.org/10.1145/359104.359108

[7] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* (1988). https://doi.org/10.1145/42282.42283

[8] Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear type theory for asynchronous session types. *JFP* (2010). https://doi.org/10.1017/S0956796809990268

[9] Leon Gondelman, Jonas Kastberg Hinrichsen, Marío Pereira, Amin Timany, and Lars Birkedal. 2023. Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols. ICFP (2023). https://doi.org/10.1145/3607859

[10] Ruben Hamers and Sung-Shik Jongmans. 2020. Discourje: Runtime Verification of Communication Protocols in Clojure. In *TACAS*. https://doi.org/10.1007/978-3-030-45190-5_15

[11] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2022. Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic. *LMCS* (2022). https://doi.org/10.46298/lmcs-18(2:16)2022

[12] Jonas Kastberg Hinrichsen, Jules Jacobs, and Robbert Krebbers. 2024. Multris: Functional Verification of Multiparty Message Passing in Separation Logic. OOPSLA (2024). https://doi.org/10.1145/3689762

[13] Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. 2021. Machine-checked semantic session typing. In *CPP*. https://doi.org/10.1145/3437992.3439914

[14] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022. Multiparty GV: Functional Multiparty Session Types with Certified Deadlock Freedom. ICFP (2022). https://doi.org/10.1145/3547638

[15] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2024. Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing. POPL (2024). https://doi.org/10.1145/3632889

[16] Sung-Shik Jongmans and Francisco Ferreira. 2023. Synthetic Behavioural Typing: Sound, Regular Multiparty Sessions via Implicit Local Types (Pearl/Brave New Idea). In *ECOOP*. https://doi.org/10.4230/LIPICS.ECOOP.2023.42

[17] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. https://doi.org/10.1145/2951913.2951943

[18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* (2018). https://doi.org/10.1017/S0956796818000151

[19] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. https://doi.org/10.1145/2676726.2676980

[20] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. ICFP (2018). https://doi.org/10.1145/3236772

[21] Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. 2021. Generalising Projection in Asynchronous Multiparty Session Types. In *CONCUR*. https://doi.org/10.4230/LIPICS.CONCUR.2021.35

[22] Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I and II. *Inf. Comput.* (1992). https://doi.org/10.1016/0890-5401(92)90008-4

[23] Catuscia Palamidessi. 2003. Comparing The Expressive Power Of The Synchronous And Asynchronous Pi-Calculi. *Math. Struct. Comput. Sci.* (2003). https://doi.org/10.1017/S0960129503004043

[24] Catuscia Palamidessi and Oltea Mihaela Herescu. 2005. A randomized encoding of the Pi-calculus with mixed choice. *Theor. Comput. Sci.* (2005). https://doi.org/10.1016/J.TCS.2004.11.020

[25] Jonah Pears, Laura Bocchi, and Andy King. 2023. Safe Asynchronous Mixed-Choice for Timed Interactions. In *COORDINATION*. https://doi.org/10.1007/978-3-031-35361-1_12

[26] Kirstin Peters and Nobuko Yoshida. 2024. Separation and Encodability in Mixed Choice Multiparty Sessions. In *LICS (LICS '24)*. https://doi.org/10.1145/3661814.3662085

[27] John H. Reppy, Claudio V. Russo, and Yingqi Xiao. 2009. Parallel concurrent ML. In *ICFP*. https://doi.org/10.1145/1596550.1596588

[28] Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *POPL* (2019). https://doi.org/10.1145/3290343

[29] Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Modular Reasoning about Separation of Concurrent Data Structures. In *ESOP*. https://doi.org/10.1007/978-3-642-37036-6_11

[30] The Clojure Team. 2025. https://clojure.org.

[31] The Go Team. 2025. https://go.dev/.

[32] The Iris Team. 2025. https://gitlab.mpi-sws.org/iris/iris/blob/master/docs/heap_lang.md.

[33] The Rocq Team. 2025. https://rocq-prover.org/.

[34] Bent Thomsen, Lone Leth Thomsen, and Tsung-Min Kuo. 1996. A Facile Tutorial. In *CONCUR*. https://doi.org/10.1007/3-540-61604-7_61

[35] Yih-Kuen Tsay and Rajive L. Bagrodia. 1994. Fault-Tolerant Algorithms for Fair Interprocess Synchronization. *IEEE Trans. Parallel Distributed Syst.* (1994). https://doi.org/10.1109/71.296319

[36] Philip Wadler. 2012. Propositions as sessions. In *ICFP*. https://doi.org/10.1145/2364527.2364568