Multris: Functional Verification of Multiparty Message Passing in Separation Logic

JONAS KASTBERG HINRICHSEN, Aarhus University, Denmark
JULES JACOBS, Cornell University, USA
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands

We introduce Multris, a separation logic for verifying the functional correctness of programs that combine multiparty message-passing communication with shared-memory concurrency. The foundation of our work is a novel notion of multiparty *protocol consistency*, which guarantees safe communication among a set of parties, provided each party adheres to its prescribed protocol. We develop tactics for automatically verifying protocol consistency and for reasoning about message-passing operations in Multris. We evaluate Multris on a range of examples, including the well-known two- and three-buyer protocols, as well as a new verification benchmark based on Chang and Roberts's ring leader election protocol. To ensure the reliability of our work, we prove soundness of Multris w.r.t. a low-level channel semantics using the Iris framework in Coq.

1 INTRODUCTION

Message passing is an attractive concurrency paradigm due to its simplicity and expressiveness. Verification of message-passing programs has thus received a lot of attention. A prominent approach is the discipline of session types [14, 15], which is built around protocols consisting of sequences of send ($!\tau$) and receive ($?\tau$) actions that specify what operations should be performed in what order on a channel. While session type systems can automatically guarantee desirable properties such as memory safety and deadlock freedom through type checking, they do not guarantee functional correctness, *i.e.,* they do not guarantee that **assert** statements never fail at run-time, nor that the output of the program satisfies a specification. As a result, several systems for functional verification based on session types have been developed [4, 8, 29, 7, 32, 38, 11, 12, 20]. We categorize these as:

(1) Whether the verification system supports message-passing communication between *two parties* (binary) only, or between *multiple parties* (multiparty) [16, 17].
(2) Whether the verification system supports programs whose correctness relies on the *interaction* of message passing with *shared memory* concurrency. This is important because empirical studies of large programs in Go and Scala [35, 37] show that programmers often use shared memory in practice.
(3) Whether the verification system is *higher-order*, *i.e.,* it supports the verification of programs that send functions and channels as messages. This is important to reason about programs that use delegation or are written in functional style [9].
(4) Whether the verification system comes with a *foundational proof* [1] of soundness that is machine-checked by a general-purpose proof assistant such as Agda, Isabelle, Coq or Lean. The formal soundness theorem states that if a program can be verified according to the rules of the verification system, then it is functionally correct w.r.t. the operational semantics of the language. This is valuable given that the literature is known to contain unsound results about type systems for multiparty session types [33].

To date, there are verification systems that cover (1), but only support (2–4) to a limited extent; and verification systems that cover (2–4), but not (1). The design-by-contract system by Bocchi et al. [4] covers (1), but does not support reasoning about shared memory and higher-order programs, and neither has a foundational soundness proof. Similarly, the refinement type system by Zhou et al. [38] covers (1), and while it is embedded in the F* proof assistant, this is primarily to make use of the infrastructure of F*—a machine-checked soundness proof is missing. On the other end of

the spectrum, the Actris separation logic [11, 12, 19] in the Iris framework in Coq [24, 22, 26, 23] covers (2–4), but not (1)—Actris only supports message passing between two parties.

This paper addresses this gap and presents **Multris**, which is (to our knowledge) the most powerful verification system for message passing to date, and support reasoning about *multiparty messaging* combined with shared memory and higher-order messaging. Multris features a foundational soundness proof in Coq using Iris. Note that like Iris (and other concurrent program logics that build on it) we verify *partial correctness*, which means that we do not prove termination. Termination is an open problem for higher-order logics such as Iris, even in the absence of message passing. Before describing our conceptual contributions, we give a brief introduction to Multris.

Example and Key Features of Multris. Let us consider the following small but illustrative multi-party program consisting of three threads that are executed in parallel:

Party A:	**Party B:**	**Party C:**
$\mathbf{let}\ l = \mathbf{ref}\ 40\ \mathbf{in}$	$\mathbf{let}\ x = c[A].\mathbf{recv}()\ \mathbf{in}$	$\mathbf{let}\ l = c[B].\mathbf{recv}()\ \mathbf{in}$
$c[B].\mathbf{send}(l);\ c[C].\mathbf{recv}();$	$c[C].\mathbf{send}(x)$	$l \leftarrow\ !\, l + 2;$
$\mathbf{assert}(!\, l == 42)$		$c[A].\mathbf{send}()$

Party A allocates a reference containing the value 40 and sends this to Party B. Party B simply forwards the reference to Party C. Party C increases the value of the reference and sends an acknowledgment (the unit value) to Party A. Party A asserts that the value of the reference is 42.

Multris uses protocols that are inspired by *local types* [16, 17] in multiparty session types and *dependent separation protocols* in Actris. The protocol $!\,[i]\ (\vec{x} : \vec{\tau})\ \langle v \rangle \{P\}.\ p$ expresses that a message must be sent to party i with value v satisfying P, and continue with protocol p. Dually, $?[i]\ (\vec{x} : \vec{\tau})\ \langle v \rangle \{P\}.\ p$ says a message can be received from party i with value v satisfying P. The binders $\vec{x} : \vec{\tau}$ are used to introduce logical variables. To verify the example program, we use:

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

$$p_B \triangleq\ ?[A]\ (v : \mathsf{Val})\ \langle v \rangle.\ !\,[C]\ \langle v \rangle.\ \mathbf{end}$$

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

An important ingredient of the Multris protocols is the assertion $\{P\}$. Unlike prior work on multiparty verification [4, 38], but like Actris, P is an arbitrary separation logic proposition instead of a pure proposition. This means we can use it to transfer ownership of a location $\ell \mapsto x$ from one party to another. Since Multris is based on Iris, the power of the binders $\vec{x} : \vec{\tau}$ and assertion P allow it to handle challenging features. Particularly, Hoare triples $\{P\}\ e\ \{\Phi\}$ and *channel ownership assertions* $c \rightarrowtail p$ are first-class Multris propositions and can be used in the protocol assertions, thereby supporting higher-order programs that send functions and channels as messages

Soundness of Multris guarantees that if we can prove the Hoare triple $\{c \rightarrowtail p_i\}\ \mathbf{Party}\ i\ \{\mathsf{True}\}$ for every party i, the program as a whole cannot lead to a failing **assert**. A crucial condition for soundness is *protocol consistency*, *i.e.,* that the behavior of all senders and receivers matches up.

Contribution #1: Protocol consistency in separation logic. There are two well-studied methods to ensure protocol consistency in multiparty session types: *top-down* [16, 17] and *bottom-up* [33]. Using the top-down method, one starts with a *global protocol*, from which the protocols for each party are *projected*. Using the bottom-up method, one starts from the protocols for each party, and verifies all interactions. In this paper we pursue the bottom-up method because it is very general and scales well to functional verification in separation logic.

To demonstrate the use for generality, let us take a look at the example. Party A sends a reference ℓ to the Party B. Since Party B just forwards the reference and does not access its stored value, the protocol p_A just says that Party B receives and forwards an arbitrary value $(v : \mathsf{Val})$. Correspondingly,

there is no protocol assertion in p_A that talks about the ownership of the reference $\ell \mapsto n$. Ownership of $\ell \mapsto n$ is implicitly transferred from Party A to Party C, without Party B's knowledge.

Our key idea to account for this generality is to have a central coordinator at the level of separation logic (our operational semantics is truly concurrent and has no physical central coordinator). During the verification of protocol consistency, this central coordinator takes ownership of resources that are implicitly transferred. Note that while this concrete example is somewhat artificial, we think it shows an important pattern—protocols should be as abstract as possible and only expose the aspects that are relevant for the correct functioning of the party in isolation. Our novel notion of protocol consistency allows for exactly that.

Contribution #2: Tactical support for protocol consistency. To establish protocol consistency we develop a Coq tactic that explores all possible ways of executing a collection of local protocols, and thereby reduces protocol consistency to reasoning about the protocol assertions $\{P\}$. Our Coq tactic is based on two fundamental ideas.

First, to make the automatic exploration feasible, we use a synchronous semantics. Scalas and Yoshida [33] show that protocol consistency is undecidable for an asynchronous semantics of simply-typed multiparty session types (with recursion). However, since the propositions P in the protocol assertions $\{P\}$ are arbitrary Iris propositions, protocol consistency in Multris is still undecidable. Even so, we observe that using the proof technique of *framing* in separation logic protocol consistency of some non-trivial protocols can be proved automatically.

Second, to support recursive protocols, we make use of the *later modality* and *Löb induction* [31, 2]. To verify consistency of a recursive protocol, Löb induction allows us to assume that consistency holds one step *later* without the need to find a generalized induction hypothesis.

We exploit the support for framing and Löb induction in the Iris Proof Mode in Coq [26, 25] to implement these ideas. Furthermore, using the Iris Proof Mode we provide a basic form of proof automation through tactics for symbolic execution of programs against their protocols.

Contribution #3: Foundational verification. Multris comes with a *foundational proof* [1] of soundness in the Coq proof assistant: whenever our tactics succeed, we obtain an (axiom-free) Coq proof that says the program is functionally correct w.r.t. the operational semantics of the language. We define the semantics of our synchronous multiparty operations through a shallow embedding on top of the Iris HeapLang language. We then use Iris to define the *channel ownership assertion* $c \rightarrowtail p$ and verify Multris's novel proof rules w.r.t. that definition. Finally, we define our tactics in such a way they produce closed proofs using Multris's proof rules.

Contribution #4: Multiparty verification benchmark. Most literature on multiparty session types have used the two- and three-buyer protocols [16, 3] as canonical benchmarks. While we do support these benchmarks (and have mechanized them, see our accompanying artifact [21]), we propose a new benchmark based on Chang and Roberts [6]'s ring leader election algorithm.

Ring leader election is interesting as the correctness of each party depends on the full network (the entire ring), while they are only locally aware of parts of the network (*i.e.,* their neighbors). We propose various dimensions to the benchmark—implementation, language, guarantees, scalability, proof automation, and features—and indicate for which of these dimensions we can solve the benchmark already and which provide useful directions for future research by the community.

Outline. §2 gives a tour of Multris highlighting its key features, proof rules, and novel notion of protocol consistency (Contribution #1). §3 presents our new multiparty verification benchmark based on ring leader election (Contribution #4). §4 describes the model of Multris and its foundational soundness proof (Contribution #3). §5 describes our mechanization in Coq and the tactic for proving

protocol consistency (Contribution #2). §6 concludes with related and future work. All of our results and examples have been mechanized in Coq [21].

2 TOUR OF MULTRIS

At a high level, Multris consists of the following components, which we briefly illustrate here and then describe in more detail in the corresponding sections:

§2.1 Introduces a programming language with constructs for multiparty message passing:

$$\begin{array}{l}\textbf{let}\ (c_0, c_1, c_2) = \textbf{new\_chan}(3)\ \textbf{in}\\ \textbf{fork}\{\ \ \textbf{let}\ x = c_1[0].\textbf{recv}()\ \textbf{in}\ c_1[2].\textbf{send}(x+20)\ \ \};\\ \textbf{fork}\{\ \ \textbf{let}\ x = c_2[1].\textbf{recv}()\ \textbf{in}\ c_2[0].\textbf{send}(x+30)\ \ \};\\ \underbrace{c_0[1].\textbf{send}(100)}_{\text{Send 100 to party 1}};\ \textbf{let}\ x = \underbrace{c_0[2].\textbf{recv}()}_{\text{Receive 150 from party 2}}\ \textbf{in}\ \textbf{assert}(x = 150)\end{array}$$

§2.2 States our goal: a separation logic for verifying multiparty message-passing programs such as the program above, with a soundness theorem that establishes that if $\{P\}\ e\ \{Q\}$ is derivable, then running program e in state P does not crash, and Q holds in the final state. For instance, if $\{\mathsf{True}\}\ e\ \{\mathsf{True}\}$ is proved for the program above, that should establish that $\textbf{assert}(x = 150)$ cannot fail.

Sections §2.3–§2.5 describe the components of the logic in more detail:

§2.3 Extends separation logic with channel ownership $c \rightarrowtail p$, which asserts that channel c may be used at protocol p. The protocol p specifies the message passing behavior of a channel:

$$\begin{array}{lll}\text{Protocol for } c_0\text{:} & p_0 \triangleq & \overbrace{!\,[1]\ (x:\mathbb{Z})\ \langle x\rangle.}^{\text{Send } x:\mathbb{Z} \text{ to } 1}\ \overbrace{?[2]\ \langle x+50\rangle.}^{\text{Receive } x+50 \text{ from } 2}\ \textbf{end}\\ \text{Protocol for } c_1\text{:} & p_1 \triangleq & ?[0]\ (x:\mathbb{Z})\ \langle x\rangle.\ !\,[2]\ \langle x+20\rangle.\ \textbf{end}\\ \text{Protocol for } c_2\text{:} & p_2 \triangleq & ?[1]\ (x:\mathbb{Z})\ \langle x\rangle.\ !\,[0]\ \langle x+30\rangle.\ \textbf{end}\end{array}$$

§2.4 Describes our program logic rules for updating the protocol state at each channel operation: :

$$\{c_0 \rightarrowtail p_0\}\ c_0[1].\textbf{send}(100)\ \ \{c_0 \rightarrowtail p_0'\} \quad \text{where } p_0' = ?[2]\ \langle 150\rangle.\ \textbf{end}$$

§2.5 Introduces a notion of *protocol consistency* that ensures that demands of a receiver are always met by the corresponding sender *at the protocol level*. Checking that the protocols (p_0, p_1, p_2) above are consistent involves the proof obligation $(x + 20) + 30 = x + 50$.

These sections go into each of these components in more detail, and provide examples to illustrate language and protocol features of Multris, such as mutable state, recursion, and higher-order messaging.

2.1 The Multris Programming Language

The Multris programming language supports message-passing concurrency with constructs for creating channels, sending and receiving messages, and forking threads. The language also includes constructs for constants, arithmetic operations, control flow, pairs, functions, tagged unions, references, and atomic operations. The grammar of the language is shown in Fig. 1.

Our multiparty channels are synchronous and implemented as an independent communication channel between each pair of parties, as follows:

$\textbf{new\_chan}(n)$ creates a multiparty channel with n parties, returning a tuple $(c_1, \ldots, c_n)$ of endpoints.
$c_i[j].\textbf{send}(v)$ sends a value v to party j via endpoint c_i

$$
\begin{array}{rll}
e ::= & \texttt{new\_chan}(n) \mid c[i].\texttt{send}(v) \mid c[i].\texttt{recv}() \mid \texttt{fork}\,\{e\} \mid & \text{(Message passing concurrency)} \\
& n \mid \texttt{true} \mid \texttt{false} \mid e + e \mid e - e \mid e < e \mid \ldots & \text{(Constants and operators)} \\
& \texttt{assert}(e) \mid \texttt{let}\, x = e\, \texttt{in}\, e \mid \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e \mid & \text{(Control flow)} \\
& (e, e) \mid \texttt{fst}\, e \mid \texttt{snd}\, e \mid \lambda x.\, e \mid e\ e \mid \texttt{rec}\ \mathrm{f}\, x.\, e & \text{(Pairs and functions)} \\
& \texttt{inl}\, e \mid \texttt{inr}\, e \mid \texttt{match}\, e\, \texttt{with}\, \texttt{inl}\, e \Rightarrow e;\, \texttt{inr}\, e \Rightarrow e\, \texttt{end} \mid & \text{(Tagged unions)} \\
& \texttt{ref}\, e \mid \texttt{free}\, e \mid !\, e \mid e \leftarrow e \mid \texttt{Xchg}\, e\ e & \text{(References and atomics)}
\end{array}
$$

Fig. 1. The Multris language constructs.

$c_i[j].\texttt{recv}()$ receives a value from party j via endpoint c_i

The communication structure is such that a $c_i[j].\texttt{send}(v)$ is matched by a $c_j[i].\texttt{recv}()$. These two parties then perform a synchronous exchange when communicating: both send and receive are blocking, and the sender blocks until the receiver has received the message. Synchronization is only between the parties involved: the rest of the parties can continue to execute in parallel while a subset of the parties are blocked.

The $\texttt{fork}\,\{e\}$ construct forks a new thread to execute e. The $\texttt{assert}(e)$ construct asserts that the expression e evaluates to **true**. In Coq, the language semantics is defined with a small-step operational semantics with a thread pool, and the $\texttt{assert}(e)$ construct is made to get stuck if e evaluates to **false**. This way, if we prove that a program does not get stuck, then we have also proved that all assertions in the program are true.

The construct $\texttt{rec}\ \mathrm{f}\, x.\, e$ is like $\lambda x.\, e$, but allows the body of the function e to refer to itself using the name f. The constructs $\texttt{ref}\, v$ and $\texttt{free}\, r$ allocate and deallocate a mutable reference cell, respectively. The construct $!\, r$ reads the value of a reference cell, and $r \leftarrow v$ writes a value to a reference cell. The construct $\texttt{Xchg}\, r\ v$ atomically sets the value of reference r to v and returns the original value of r. References can be freely mixed with message passing, as shown in the following variant of the preceding example:

$$
\begin{array}{l}
\texttt{let}\ (c_0, c_1, c_2) = \texttt{new\_chan}(3)\ \texttt{in} \\
\texttt{fork}\{\ \texttt{let}\ \ell = c_1[0].\texttt{recv}()\ \texttt{in}\ \ \ell \leftarrow !\,\ell + 20;\ c_1[2].\texttt{send}(\ell)\ \ \}; \\
\texttt{fork}\{\ \texttt{let}\ \ell = c_2[1].\texttt{recv}()\ \texttt{in}\ \ \ell \leftarrow !\,\ell + 30;\ c_2[0].\texttt{send}()\ \ \}; \\
\texttt{let}\ \ell = \texttt{ref}\ 100\ \texttt{in}\ c_0[1].\texttt{send}(\ell);\ c_0[2].\texttt{recv}();\ \texttt{assert}(!\,\ell = 150)
\end{array}
$$

This program creates a reference cell ℓ with value 100, and sends r to party 1. Party 1 increments the value of ℓ by 20 and forwards the reference ℓ to party 2. Party 2 increments ℓ by 30 and sends an empty message to party 0, as 0 already knows about ℓ. Finally, party 0 asserts that the value stored in ℓ is 150. Note that the assertion crucially relies on the synchronization of the message passing, as otherwise it would be possible for the increment of party 1 to get lost, if party 2 read the value of ℓ before party 1 incremented it.

2.2 The Multris Logic and Adequacy Theorem

Our goal is to develop a separation logic that allows us to prove correct message-passing programs such as the example in the previous section. Our program logic will be presented in weakest-precondition style, from which Hoare triples can be defined as $\{P\}\ e\ \{\Phi\} \triangleq P \vdash \mathsf{wp}\ e\ \{\Phi\}$. The Multris separation logic has an adequacy theorem that states that if we can derive a weakest precondition assertion $\mathsf{wp}\ e\ \{v.\, \phi(v)\}$, where $\phi(v)$ is a purely logical assertion, then running the

Separation logic propositions:

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

Basic weakest precondition rules:

$$
\textsc{Wp-pure-step}\ \frac{e_1 \rightsquigarrow_{\mathrm{pure}} e_2 \qquad \mathsf{wp}\ e_2\ \{\Phi\}}{\mathsf{wp}\ e_1\ \{\Phi\}}* \qquad
\textsc{Wp-val}\ \frac{\Phi\ v}{\mathsf{wp}\ v\ \{\Phi\}}* \qquad
\textsc{Wp-wand}\ \frac{\mathsf{wp}\ e\ \{\Phi\} \quad * \quad \forall v.\, \Phi\ v \mathrel{-\!\!*} \Psi\ v}{\mathsf{wp}\ e\ \{\Psi\}}*
$$

$$
\textsc{Wp-rec}\ \frac{\mathsf{wp}\ e[x := v][f := \mathbf{rec}\ \mathrm{f}\ x.\ e]\ \{\Phi\}}{\triangleright\, \mathsf{wp}\ (\mathbf{rec}\ \mathrm{f}\ x.\ e)\ v\ \{\Phi\}}* \qquad
\textsc{Löb}\ \frac{\triangleright P \mathrel{-\!\!*} P}{P}\Box \qquad
\textsc{Wp-bind}\ \frac{\mathsf{wp}\ e\ \{v.\, \mathsf{wp}\ K[v]\ \{\Phi\}\}}{\mathsf{wp}\ K[e]\ \{\Phi\}}*
$$

Heap manipulation rules:

$$
\textsc{Wp-alloc}\ \frac{}{\mathsf{wp}\ \mathbf{ref}\, v\ \{\ell.\, \ell \mapsto v\}}* \qquad
\textsc{Wp-load}\ \frac{\ell \mapsto v}{\mathsf{wp}\ !\ell\ \{w.\, w = v * \ell \mapsto v\}}* \qquad
\textsc{Wp-store}\ \frac{\ell \mapsto v}{\mathsf{wp}\ \ell \leftarrow w\ \{\ell \mapsto w\}}* \qquad
\textsc{Wp-free}\ \frac{\ell \mapsto v}{\mathsf{wp}\ \mathbf{free}\, \ell\ \{\mathsf{True}\}}*
$$

Fig. 2. The basic rules of separation logic.

program e will not crash, and the return value v will satisfy the predicate $\phi(v)$. This is formally stated as follows:

Theorem 2.1 (Adequacy). *A proof of* $\mathsf{wp}\ e\ \{v.\, \phi(v)\}$ *implies that* e *is* ***safe****, i.e., if* $([e], \emptyset) \rightarrow_{\mathrm{t}}^{*} ([e_1 \ldots e_n], h)$*, then for each* i *either* e_i *is a value or* (e_i, h) *can step. Furthermore, for* $i = 0$ *any returned value* v *satisfies* $\phi(v)$*.*

The notion of safety depends on the operational semantics $(\vec{e}, h) \rightarrow_{\mathrm{t}}^{*} (\vec{e}', h')$ of the language, which is a small step semantics over a thread pool $\vec{e}$ and a heap h. This semantics has been defined in such a way that invalid operations, such as sending a message to a party that is not on the channel, or running $\mathbf{assert}(b)$ where b is $\mathbf{false}$, will cause the program to get stuck. These conditions are therefore not considered safe, and the soundness theorem guarantees that they will not happen if the program is verified using the weakest precondition logic.

Ultimately, the adequacy theorem captures that the Multris weakest preconditions (and derived Hoare triples) are sound with respect to the operational semantics.

The Multris separation logic is built on Iris [24, 22, 23, 26, 27], and as such inherits many of its features. The basic rules of Iris are shown in Fig. 2, and include rules for reasoning about the mutable references of the heap via $\ell \mapsto v$, and the separating conjunction $P * Q$. For brevity sake, we omit further details about basic separation logic as the fragment we will use is standard.

The primary new proposition added to Iris by Multris is the *channel ownership assertion*:

$$c \rightarrowtail p$$

This assertion states that we own the channel c and that the channel is currently ready to be used according to the protocol p. We will describe these protocols next.

2.3 The Multris Protocol Language

The Multris protocol language allows specifying the expected message-passing behavior of a channel. A protocol is a sequence of messages that parties can send and receive on a channel. The protocols for the example program in §2.1 are shown below:

$$\text{Protocol for } c_0: \quad p_0 \triangleq \mathbf{!}\,[1]\ (\ell : \mathsf{Loc})(x : \mathbb{Z})\ \langle \ell \rangle \{\ell \mapsto x\}.\ \mathbf{?}[2]\ \langle () \rangle \{\ell \mapsto x + 50\}.\ \mathbf{end}$$

$$\text{Protocol for } c_1: \quad p_1 \triangleq \mathbf{?}[0]\ (\ell : \mathsf{Loc})(x : \mathbb{Z})\ \langle \ell \rangle \{\ell \mapsto x\}.\ \mathbf{!}\,[2]\ \langle \ell \rangle \{\ell \mapsto x + 20\}.\ \mathbf{end}$$

$$\text{Protocol for } c_2: \quad p_2 \triangleq \mathbf{?}[1]\ (\ell : \mathsf{Loc})(x : \mathbb{Z})\ \langle \ell \rangle \{\ell \mapsto x\}.\ \mathbf{!}\,[0]\ \langle () \rangle \{\ell \mapsto x + 30\}.\ \mathbf{end}$$

The protocol for c_0 sends to party 1 (indicated by $![1]$) a message containing ℓ (indicated by $\langle \ell \rangle$) and transfers ownership over ℓ as well (indicated by $\{\ell \mapsto x\}$). In the second step of the protocol, it expects to receive from party 2 (indicated by $?[2]$) an empty message (indicated by $\langle () \rangle$), along with the assertion that the reference ℓ that it initially sent to party 1 has been incremented by 50 (indicated by $\{\ell \mapsto x + 50\}$). The protocols for c_1 and c_2 are similar, but with different parties and different increments.

The full grammar of the protocol language is as follows.

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

The meaning of these protocols is:

- $\mathbf{!}\,[i]\ (\vec{x} : \vec{\tau})\ \langle v \rangle \{P\}.\ p$: We must send a message to party i with value v. The message must satisfy the precondition P. The message is followed by a continuation protocol p. The binders $\vec{x} : \vec{\tau}$ are used to introduce logical variables that can be used in v and P as well as in the continuation protocol p. When verifying a program, the proof of the sender may choose *any* values of these logical variables.
- $\mathbf{?}[i]\ (\vec{x} : \vec{\tau})\ \langle v \rangle \{P\}.\ p$: We can receive a message from party i, which will have value v satisfying condition P. The continuation protocol is p. The binders $\vec{x} : \vec{\tau}$ work similarly to the send case, except that the receiver must be verified to work for *all* values of these variables.
- **end**: The end of the protocol.
- $\mu x.\ p$: A recursive protocol. The protocol can refer to itself using the name x.

We now cover how the rules of the Multris logic guarantees that the channel endpoints comply with their given protocols.

2.4 The Multris Message-Passing Logic

The key rules of Multris that allow one to reason about message passing are displayed in Fig. 3:

Rule Wp-new**:** This rule states that if we create a new multiparty channel with $n + 1$ (non-zero) parties using $\mathbf{new\_chan}(n + 1)$, we can pick any vector of consistent protocols $\vec{p}$ and we get an n-ary separation conjunction of channel ownership assertions $c_0 \rightarrowtail p_0 * \cdots * c_n \rightarrowtail p_n$ for the new channel endpoints. We cover protocol consistency in the following section, and remark that all protocols shown thus far are consistent.

Rule Wp-send**:** This rule states that if we have a channel c with channel ownership assertion $c \rightarrowtail \mathbf{!}\,[i]\ (\vec{x} : \vec{\tau})\ \langle v \rangle \{P\}.\ p$, then we can choose an instantiation $\vec{x} := \vec{t}$ of the binders, and the value $v[\vec{x} := \vec{t}]$ must then match the value specified in the protocol. We must also provide a proof that the precondition $P[\vec{x} := \vec{t}]$ holds, and the channel ownership assertion will then be updated to $c \rightarrowtail p[\vec{x} := \vec{t}]$.

$$\textsc{Wp-new}\ \frac{\textsc{consistent}\ \vec{p} \qquad |\vec{p}| = n+1}{\mathsf{wp}\ \mathbf{new\_chan}(n+1)\ \{(c_0, \ldots, c_n).\ c_0 \rightarrowtail p_0 * \cdots * c_n \rightarrowtail p_n\}}* \qquad \textsc{Wp-fork}\ \frac{\mathsf{wp}\ e\ \{\mathsf{True}\}}{\mathsf{wp}\ \mathbf{fork}\ \{e\}\ \{\mathsf{True}\}}*$$

$$\textsc{Wp-send}\ \frac{c \rightarrowtail !\,[i]\ (\vec{x}:\vec{\tau})\ \langle v \rangle \{P\}.\ p \qquad P[\vec{x} := \vec{t}]}{\mathsf{wp}\ c[i].\mathbf{send}(v[\vec{x} := \vec{t}])\ \{c \rightarrowtail p[\vec{x} := \vec{t}]\}}* \qquad \textsc{chan-sub}\ \frac{c \rightarrowtail p_1 \qquad p_1 \sqsubseteq p_2}{c \rightarrowtail p_2}*$$

$$\textsc{Wp-recv}\ \frac{c \rightarrowtail ?[i]\ (\vec{x}:\vec{\tau})\ \langle v \rangle \{P\}.\ p}{\mathsf{wp}\ c[i].\mathbf{recv}()\ \{w.\ \exists \vec{t}.\ w = v[\vec{x} := \vec{t}] * c \rightarrowtail p[\vec{x} := \vec{t}] * P[\vec{x} := \vec{t}]\}}* \qquad \textsc{Sub-end}\ \mathbf{end} \sqsubseteq \mathbf{end}$$

$$\textsc{Sub-send}\ \frac{\forall \vec{x_2}:\vec{\tau_2}.\ P_2 \mathrel{-\!\!*} \exists \vec{x_1}:\vec{\tau_1}.\ (v_1 = v_2) * P_1 * \triangleright(p_1 \sqsubseteq p_2)}{!\,[i]\ (\vec{x_1}:\vec{\tau_1})\ \langle v_1 \rangle \{P_1\}.\ p_1 \sqsubseteq !\,[i]\ (\vec{x_2}:\vec{\tau_2})\ \langle v_2 \rangle \{P_2\}.\ p_2}*$$

$$\textsc{Sub-recv}\ \frac{\forall \vec{x_1}:\vec{\tau_1}.\ P_1 \mathrel{-\!\!*} \exists \vec{x_2}:\vec{\tau_2}.\ (v_1 = v_2) * P_2 * \triangleright(p_1 \sqsubseteq p_2)}{?[i]\ (\vec{x_1}:\vec{\tau_1})\ \langle v_1 \rangle \{P_1\}.\ p_1 \sqsubseteq ?[i]\ (\vec{x_2}:\vec{\tau_2})\ \langle v_2 \rangle \{P_2\}.\ p_2}*$$

Fig. 3. The Multris rules for message-passing concurrency.

Rule Wp-recv**:** This rule states that if we have a channel c with channel ownership assertion $c \rightarrowtail ?[i]\ (\vec{x}:\vec{\tau})\ \langle v \rangle \{P\}.\ p$, then we can receive a value w from the channel, and we learn that the value w is equal to the value $v[\vec{x} := \vec{t}]$ specified by the protocol, for some instantiation $\vec{x} := \vec{t}$ of the binders. We also obtain the condition $P[\vec{x} := \vec{t}]$, and the channel ownership assertion will be updated to $c \rightarrowtail p[\vec{x} := \vec{t}]$.

Rule Wp-fork**:** This rule states that if we fork a new thread to execute e, it is enough to verify that e does not crash.

Rule chan-sub**:** This rules capture that if we have a channel c with channel ownership assertion $c \rightarrowtail p_1$, and the protocol p_1 is a subprotocol of p_2, then we can use the channel with the protocol p_2.

Rules Sub-send**,** Sub-recv**,** Sub-end**:** These rules capture how to prove the subprotocol relation. Notably, we can strengthen sending protocols, *e.g.,* $!\,[i]\ (i : \mathbb{Z})\ \langle i \rangle \{i < 42\}.\ p \sqsubseteq !\,[i]\ (i : \mathbb{Z})\ \langle i \rangle \{i < 40\}.\ p$, and conversely weaken receiving protocols. We can also use this to prematurely "satisfy" sending protocols, *e.g.,* given $\ell \mapsto 42$, we can prove $!\,[i]\ (\ell : \mathsf{Loc}, x : \mathbb{Z})\ \langle \ell \rangle \{\ell \mapsto x\}.\ p \sqsubseteq !\,[i]\ \langle \ell \rangle.\ p$

With these rules we can prove the weakest precondition for the program shown in §2.2, given the protocols shown in §2.3. The proof follows almost entirely from symbolic execution, with the exception of the protocol consistency, which we will cover in the following section.

2.5 Multris Protocol Consistency

When a new multiparty channel of size $n+1$ is created, we choose protocols $(p_0, \ldots, p_n)$ for each party on the channel. The verifier must ensure that the protocols $(p_0, \ldots, p_n)$ are *consistent, i.e.,* that the behavior of the senders and corresponding receivers matches up. Our notion of protocol consistency is very general, so we will first consider an example.

$$
\begin{aligned}
p_0 &\triangleq \mathbf{!}\,[1]\,(\ell : \mathsf{Loc})(x : \mathbb{Z})\,\langle \ell \rangle\{\ell \mapsto x\}.\ \mathbf{?}[2]\,\langle () \rangle\{\ell \mapsto x + 50\}.\ \mathbf{end} \\
p_1 &\triangleq \mathbf{?}[0]\,(\ell : \mathsf{Loc})(x : \mathbb{Z})\,\langle \ell \rangle\{\ell \mapsto x\}.\ \mathbf{!}\,[2]\,\langle \ell \rangle\{\ell \mapsto x + 20\}.\ \mathbf{end} \\
p_2 &\triangleq \mathbf{?}[1]\,(\ell : \mathsf{Loc})(x : \mathbb{Z})\,\langle \ell \rangle\{\ell \mapsto x\}.\ \mathbf{!}\,[0]\,\langle () \rangle\{\ell \mapsto x + 30\}.\ \mathbf{end} \\
&\xrightarrow{c_0 \text{ sends to } c_1 \text{ (message: } \ell\text{, resources: } \ell \mapsto x)} \\
p_0' &\triangleq \mathbf{?}[2]\,\langle () \rangle\{\ell \mapsto x + 50\}.\ \mathbf{end} \\
p_1' &\triangleq \mathbf{!}\,[2]\,\langle \ell \rangle\{\ell \mapsto x + 20\}.\ \mathbf{end} \\
p_2 &\triangleq \mathbf{?}[1]\,(\ell : \mathsf{Loc})(x : \mathbb{Z})\,\langle \ell \rangle\{\ell \mapsto x\}.\ \mathbf{!}\,[0]\,\langle () \rangle\{\ell \mapsto x + 30\}.\ \mathbf{end} \\
&\xrightarrow{c_1 \text{ sends to } c_2 \text{ (message: } \ell\text{, resources: } \ell \mapsto x + 20)} \\
p_0' &\triangleq \mathbf{?}[2]\,\langle () \rangle\{\ell \mapsto x + 50\}.\ \mathbf{end} \\
p_1'' &\triangleq \mathbf{end} \\
p_2' &\triangleq \mathbf{!}\,[0]\,\langle () \rangle\{\ell \mapsto x + 50\}.\ \mathbf{end} \\
&\xrightarrow{c_2 \text{ sends to } c_0 \text{ (message: } ()\text{, resources: } \ell \mapsto x + 50)} \\
p_0'' &\triangleq \mathbf{end} \\
p_1'' &\triangleq \mathbf{end} \\
p_2'' &\triangleq \mathbf{end}
\end{aligned}
$$

Fig. 4. An example of the Multris protocol consistency simulation.

Protocol consistency is checked by simulating the possible interactions on the abstract protocol level. Consider the simulation for the preceding example protocols (p_0, p_1, p_2) shown in Fig. 4. For the initial protocols, there is only one possible interaction: c_0 sends to c_1. In this interaction, c_0 sends the value of ℓ to c_1, as well as the ownership of the reference cell ℓ. Once this step has been taken, the new protocol for c_0 is p_0', and the new protocol for c_1 is p_1', as shown above. The new protocol for c_2 remains the same, as it is not involved in this interaction.

For the new protocols, c_1 sends to c_2. In this interaction, c_1 sends an empty message to c_2. In order for the resource assertions to match, we must instantiate the logical variables in the receiver's protocol p_2 with ℓ and $x + 20$. In general, in each step we must show that for all possible choices of the logical variables in the sender's protocol, there exists an instantiation of the receivers protocol that makes the messages and resource assertions match up. In fact, the resources need not match up exactly, but the sender's resource assertion must imply the receiver's resource assertion.

In the last step, the protocols match trivially, and all parties have completed their interactions. In this particular example, there was always only one possible interaction at each step, but in general, there may be multiple possible interactions, and the verifier must show that all possible interactions are consistent (*e.g.,* if party 0 sends to 1, and party 2 sends to 3, then it is nondeterministic which one happens first).

It is important to note that protocol consistency is a purely protocol-level notion, and is independent of the actual program that is being verified.

The general case of protocol consistency. In the general case we have a set of protocols $(p_0, \ldots, p_i, \ldots, p_j, \ldots, p_n)$ where i and j have protocols that are ready to communicate:

$$p_i = \mathbf{!}\,[j]\;(\vec{x}_1 : \vec{\tau}_1)\;\langle v_1\rangle\{P_1\}.\,p_i' \qquad\qquad p_j = \mathbf{?}[i]\;(\vec{x}_2 : \vec{\tau}_2)\;\langle v_2\rangle\{P_2\}.\,p_j'$$

In this case a communication between i and j could happen. Therefore, we need to ensure that *for all* choices of $\vec{x}_1 : \vec{\tau}_1$, there exists a choice of $\vec{x}_2 : \vec{\tau}_2$ such that:

(1) The message v_1 that the sender i sends is consistent with the message v_2 that the receiver j expects.
(2) The condition P_1 that the sender guarantees implies the condition P_2 that the receiver expects.
(3) The continuation protocols p_i' and p_j' remain consistent with the new set of protocols $(p_0, \ldots, p_i', \ldots, p_j', \ldots, p_n)$ after the communication has happened.

This notion of consistency is flexible:

- We allow the number of binders and their types to differ between the sender and receiver. We merely insist that for every choice of $\vec{t}_1$ instantiating $\vec{x}_1 : \vec{\tau}_1$, there exists a choice of $\vec{t}_2$ instantiating $\vec{x}_2 : \vec{\tau}_2$ such that the messages match up, the conditions are implied, and the continuation protocols are consistent. This allows, for instance, to match a concrete sender with a parametric receiver:

  $$p_0 \triangleq \mathbf{!}\,[1]\;\langle 20\rangle.\,\mathbf{end} \qquad\qquad p_1 \triangleq \mathbf{?}[0]\;(x : \mathbb{Z})\;\langle x\rangle\{x > 0\}.\,\mathbf{end}$$

  Binders introduced by senders are available for use in instantiating the binders of receivers in the entirety of the continuation protocol, not just in the receiver that is the target of the send. This allows for *implicit transfer of information*, such as in the following example:

  $$p_0 \triangleq \mathbf{!}\,[1]\;(x : \mathbb{Z})\;\langle x\rangle.\,\mathbf{?}[2]\;\langle x + 2\rangle.\,\mathbf{end}$$
  $$p_1 \triangleq \mathbf{?}[0]\;(v : \mathsf{Val})\;\langle v\rangle.\,\mathbf{!}\,[2]\;\langle v\rangle.\,\mathbf{end}$$
  $$p_2 \triangleq \mathbf{?}[1]\;(x : \mathbb{Z})\;\langle x\rangle.\,\mathbf{!}\,[0]\;\langle x + 2\rangle.\,\mathbf{end}$$

  Here, 0 sends a number x to 1, although 1 receives it as a value. However, the protocol consistency retains knowledge of the true value of $v = x$, and so 2 can receive it as such.
- We allow accumulating resources from previous interactions. Once a communication between $\mathbf{!}\,[b]\;(\vec{x}_1 : \vec{\tau}_1)\;\langle v_1\rangle\{P_1\}.\,p_a'$ and $\mathbf{?}[a]\;(\vec{x}_2 : \vec{\tau}_2)\;\langle v_2\rangle\{P_2\}.\,p_b'$ has happened, we retain any leftover resources of P_1 that were not used to satisfy P_2. This allows for *implicit transfer of resources*, such as in the example from §1:

  $$p_0 \triangleq \mathbf{!}\,[1]\;(\ell : \mathsf{Loc})(x : \mathbb{Z})\;\langle \ell\rangle\{\ell \mapsto x\}.\,\mathbf{?}[2]\;\langle ()\rangle\{\ell \mapsto (x + 2)\}.\,\mathbf{end}$$
  $$p_1 \triangleq \mathbf{?}[0]\;(v : \mathsf{Val})\;\langle v\rangle.\,\mathbf{!}\,[2]\;\langle v\rangle.\,\mathbf{end}$$
  $$p_2 \triangleq \mathbf{?}[1]\;(\ell : \mathsf{Loc})(x : \mathbb{Z})\;\langle \ell\rangle\{\ell \mapsto x\}.\,\mathbf{!}\,[0]\;\langle ()\rangle\{\ell \mapsto (x + 2)\}.\,\mathbf{end}$$

  Here, the resources $\ell \mapsto x$ that the sender 0 yields is implicitly transferred to 2 through 1, even though 1's protocol does not mention neither ℓ, x, nor $\ell \mapsto x$. This is sound as the protocol consistency retains knowledge of the true value of $v = \ell$ along with the resources $\ell \mapsto x$ through the exchanges with 1.
- We allow for consistency proofs of recursive protocols using Löb induction. If during the course of the consistency proof, we loop back to a set of protocols that was seen earlier, Löb induction allows us to immediately finish the proof. This may strike the reader as wildly unsound: if our aim is to prove that an initial set of protocols is consistent, then how can we *assume* that they are consistent, simply by the fact that we revisit them? The key is that we are proving *partial correctness*: we are proving that *if* a communication happens, then the sender and receiver will behave correctly. We are not proving that a communication *will*

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

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

Fig. 5. The rules for protocol consistency.

happen, nor that the protocol will terminate. Even so, this is highly useful if we want to verify services that loop indefinitely. This is a common pattern in separation logic, where one proves that a program does not crash, and satisfies its postcondition *if* it terminates, but does not prove termination. The support for recursion let's us prove consistency of protocols, such as the following recursive variant of the above protocol:

$$p_0 \triangleq \mathbf{!}\,[1]\ (\ell : \mathrm{Loc})(x : \mathbb{Z})\ \langle \ell\rangle\{\ell \mapsto x\}.\ \mathbf{?}[2]\ \langle()\rangle\{\ell \mapsto (x+2)\}.\ p_0$$
$$p_1 \triangleq \mathbf{?}[0]\ (v : \mathrm{Val})\ \langle v\rangle.\ \mathbf{!}\,[2]\ \langle v\rangle.\ p_1$$
$$p_2 \triangleq \mathbf{?}[1]\ (\ell : \mathrm{Loc})(x : \mathbb{Z})\ \langle \ell\rangle\{\ell \mapsto x\}.\ \mathbf{!}\,[0]\ \langle()\rangle\{\ell \mapsto (x+2)\}.\ p_2$$

The consistency of protocols is formally defined by the rules in Fig. 5. The key is the second hypothesis of the rule for $\textsc{dual}\ \vec{p}\ i\ j$:

$$\forall \vec{x}_1:\vec{\tau}_1.\ P_1 \mathrel{-\!\!*} \exists \vec{x}_2:\vec{\tau}_2.\ v_1 = v_2 * P_2 * \triangleright(\textsc{consistent}\ (\vec{p}[i := p_1][j := p_2]))$$

This rule allows the binders to differ, because we must prove that for all choices of sender binders $\vec{x}_1:\vec{\tau}_1$, there exists a choice of receiver binders $\vec{x}_2:\vec{\tau}_2$. The sender's proposition P_1 can be assumed not only to prove the receiver's proposition P_2, but also to prove the continuation protocols consistent. The later modality $\triangleright$ is used for Löb induction to prove the consistency of recursive protocols.

The $\textsc{present}\ \vec{p}\ i$ obligation exist to prevent parties from trying to synchronize with non-existent parties, which evidently result in an illegal operation on trying to access a non-existent buffer.

Asserting the protocol consistency is undecidable, and may thus need manual proof effort, although we have constructed a brute-force procedure, as described in the following section. , that leverage some of the automation of the Iris Proof Mode, which has successfully automated the proof all protocol consistencies presented in the paper.

3 MULTIPARTY VERIFICATION BENCHMARK: RING LEADER ELECTION

We demonstrate the expressive power of Multris by verifying a version of Chang and Roberts [6]'s ring leader election algorithm (§3.1). We then discuss the limitations of our approach, and set a ring leader election benchmark challenge for the verification community (§3.2).

The key aspect of leader election is that only one participant should be elected as the leader. This property is challenging to verify, as it requires reasoning about the global state of the session, while each participant only interacts locally. Our consistency relation is well suited to the task. In the setting of separation logic, there is a particularly elegant way to encode the uniqueness of the leader: let the leader obtain a resource R that is up for election. If the resource is affine, then it can inherently only be obtained by one participant. Thus, if any participant can obtain the resource, then the leader is necessarily unique. By adding an exclusive resource to R, such as the exclusive permission to a reference $\ell \mapsto v$, or an exclusive token $\mathsf{tok}(\gamma)$, we can prove that $R * R \vdash \mathrm{False}$, which allows us to observe inside the logic that the leader is unique, by contradiction.

3.1 Verifying Chang and Roberts's Ring Leader Election Algorithm

Chang and Roberts [6]'s ring leader election algorithm assumes that n participants, with unique IDs $id_0 \ldots id_{(n-1)}$, are arranged in a ring, where every participant i receives messages from counter-clockwise participant $(i-1)\%n$, and sends messages to clockwise participant $(i+1)\%n$. All participants are initially marked as non-participating. There exists two types of message; election(i) messages and elected(i) messages. Received election(i') messages are compared to the receivers ID i and:

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

Received elected(i') messages are compared to the participants ID i and:

- If $i' = i$, terminate by returning i'. **(2.1)**
- If $i' \neq i$, send elected(i'), and terminate by returning i'. **(2.2)**

Any process may arbitrarily start an election (concurrently). To do so, it sends an election message with its ID clockwise. Every time a process sends or forwards an election message, the process also marks itself as a participant.

Election process. We encode election(i) as **inl** i and elected(i) as **inr** i. We write i_l and i_r for the left and right participants of participant i, respectively. They are constructed as $i_l \triangleq (i+1)\%n$ and $i_r \triangleq (i-1)\%n$, for the given ring of size n when the protocol network is defined. Note that this does not mean that i is aware of the size of the ring n. The leader election process can then be implemented as follows:

```
process c i ≜ rec rec isp.
  match c[i_r].recv() with
  | inl i' ⇒ if i < i' then c[i_l].send(inl i'); rec true          (1.1)
             else if i = i' then c[i_l].send(inr i); rec false     (1.2)
             else if isp then rec true                            (1.3)
             else c[i_l].send(inl i); rec true                    (1.4)
  | inr i' ⇒ if i = i' then i'                                    (2.1)
             else c[i_l].send(inr i'); i'                         (2.2)
  end
```

The implementation is in 1-to-1 correspondence with the version on paper, where each of the cases are accounted for. A participant with channel endpoint c can start an election by sending an initial election message clockwise with the participants own ID i:

```
init c i ≜ c[i_l].send(inl i); process c i true
```

Simple leader election example. To verify that the ring leader election appropriately elects a leader, and gives it the resources up for election, we use the following example:

```
ring_ref_prog n ≜
  let ℓ = ref 42 in
  let (c_0, ..., c_{n-1}) = new_chan(n) in
  for(i = 1 ... (n − 1)) {fork {let i' = process c_i i false in if i' = i then free ℓ else ()}}
  let i' = init c_0 0 in if i' = 0 then free ℓ else ()
```

We first allocate a reference ℓ, for which the exclusive permission is up for election. We then initialize the network, and spawn n participants. The main thread will start the election, which will

determine who gets to deallocate ℓ. Once the leader election is complete, all participants will try to deallocate the reference, if they deduce that they are the leader $i = i'$. The program is only safe (no double free) if there is (at most) one leader, which is the property we want to verify.

Verifying the simple leader election example. To verify the program, we first define its protocol. We first define branching protocols in terms of receiving protocols:

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

The binders use a sum-type, which dictates whether we receive **inl** or **inr**. Depending on the projection of the binders, we either receive P_1 or P_2 and continue as either p_1 or p_2. We now define the ring leader election protocol as:

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

The protocol is in 1-to-1 correspondence with the program, reflecting each its cases. This is *necessary*, as the protocol has to reflect the leader election algorithm in its entirety, as it would otherwise not be able to guarantee that a single leader is elected, and that all participants agree on the leader after the election. The protocol is parametric in resources P that are up for election, which are given to the elected leader in (2.1). The protocol is parameterized by a tail p, which in turn is parametric in the elected leader i'. With this protocol we verify the ring leader process:

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

The process does not locally know who the leader is (unless it is itself). However, since the protocol tail p is parameterized with the elected leader, it can leverage this fact.

The protocol for starting an election is a straightforward extension of the process protocol:

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

The initial message yields the P resource to the network, which the leader can then obtain upon election. With this protocol we can prove the following specification for the starting process:

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

We verify the program for 4 participants, by first proving the protocol consistency of the system:

$$\begin{array}{l}c_0 \rightarrowtail \mathsf{init\_prot}\ 0\ (\ell \mapsto 42)\ (\lambda i'.\,\mathbf{end})\\ c_1 \rightarrowtail \mathsf{ring\_prot}\ 1\ (\ell \mapsto 42)\ (\lambda i'.\,\mathbf{end})\ \mathbf{false}\\ c_2 \rightarrowtail \mathsf{ring\_prot}\ 2\ (\ell \mapsto 42)\ (\lambda i'.\,\mathbf{end})\ \mathbf{false}\\ c_3 \rightarrowtail \mathsf{ring\_prot}\ 3\ (\ell \mapsto 42)\ (\lambda i'.\,\mathbf{end})\ \mathbf{false}\end{array}$$

While it may not be immediately obvious, the proof follows straightforwardly by repeatedly aligning all synchronizations, and eventually observe that only one participant attempts to obtain the resource $\ell \mapsto 42$. The proof of consistency is entirely automated by our brute-force tactic.

With the above protocols we verify the program with 4 participants:

$$\{\mathsf{True}\}\ \mathsf{ring\_ref\_prog}\ 4\ \{\mathsf{True}\}$$

This Hoare triple guarantees that the program is safe to execute via our adequacy theorem.

Elected leader conformance. The above example verifies that at most one leader gets access to the drafted resources, however it does not leverage that all participants agree on the elected process after the election. To demonstrate this property we instead allocate an additional binary channel, which we use as a coordinator that receives each elected id, and asserts that they are all the same. To achieve this, we give the channel endpoint resource up for election, and let the elected leader relay their elected id, after which point it delegates the channel endpoint through the ring, so that each participant can relay their elected ID to the coordinator.

We first capture the relaying of the elected ID for each participant as follows:

$$\begin{array}{l}\mathsf{relay}\ c\ c'\ i\ i' \triangleq \\ \quad \mathbf{if}\ i = i'\ \mathbf{then}\ c'[0].\mathbf{send}(i'); c[i_l].\mathbf{send}(); c[i_r].\mathbf{recv}() \\ \quad \mathbf{else}\ c[i_r].\mathbf{recv}(); c'[0].\mathbf{send}(i'); c[i_l].\mathbf{send}()\end{array}$$

If the given participant is the leader ($i = i'$), it relays its ID over c', and then delegates the endpoint permission clockwise. Otherwise, the participant awaits the endpoint permission, after which point it relays its elected ID, and forwards the endpoint permission.

With the relay procedure in hand, we can define the relay example as follows:

$$\begin{array}{l}\mathsf{ring\_del\_prog}\ n \triangleq \\ \quad \mathbf{let}\ (c_r, c_s) = \mathbf{new\_chan}(2)\ \mathbf{in} \\ \quad \mathbf{fork}\left\{\mathbf{let}\ i' = c_r[1].\mathbf{recv}()\ \mathbf{in}\ \mathbf{while}(\mathbf{true})\{\mathbf{assert}(c_r[1].\mathbf{recv}() = i')\}\right\} \\ \quad \mathbf{let}\ (c_0, \ldots, c_{n-1}) = \mathbf{new\_chan}(n)\ \mathbf{in} \\ \quad \mathbf{for}(i = 1 \ldots (n-1))\left\{\mathbf{fork}\left\{\mathbf{let}\ i' = \mathsf{process}\ c_i\ i\ \mathbf{false}\ \mathbf{in}\ \mathsf{relay}\ c_i\ c_s\ i\ i'\right\}\right\} \\ \quad \mathbf{let}\ i' = \mathsf{init}\ c_0\ 0\ \mathbf{in}\ \mathsf{relay}\ c_0\ c_s\ 0\ i'\end{array}$$

We first allocate the binary channel coordinator and fork a thread which receives relayed IDs and assert that they are equal. Each participant carries out the leader election like before, and then carry about the relay procedure with the elected ID, after finishing the leader election. The program is only safe to execute if all elected IDs are identical, as the coordinator would otherwise not be able to assert their equality.

We start the verification effort, by verifying the relay procedure, based on the following protocols:

$$\begin{array}{rl}\mathsf{relay\_prot}'\ i' & \triangleq \mu rec.\, !\,[0]\,\langle i'\rangle.\, rec \\ \mathsf{relay\_prot} & \triangleq\ !\,[0]\,(i' : \mathbb{N})\,\langle i'\rangle.\,\mathsf{relay\_prot}'\ i' \\ \mathsf{del\_prot}\ i\ i'\ c' & \triangleq\ \mathbf{if}\ i = i'\ \mathbf{then}\ !\,[i_l]\,\langle()\rangle\{c' \rightarrowtail \mathsf{relay\_prot}'\ i'\}.\,?[i_r]\,\langle()\rangle\{c' \rightarrowtail \mathsf{relay\_prot}'\ i'\}.\,\mathbf{end} \\ & \quad\ \mathbf{else}\ ?[i_r]\,\langle()\rangle\{c' \rightarrowtail \mathsf{relay\_prot}'\ i'\}.\,!\,[i_l]\,\langle()\rangle\{c' \rightarrowtail \mathsf{relay\_prot}'\ i'\}.\,\mathbf{end}\end{array}$$

The relay_prot protocol captures that the leader first determines the elected ID i', after which point everyone must relay the same ID. The del_prot protocol captures that the leader must first delegate the channel endpoint permission of the coordinator, after which point it can await its return. Conversely, non-leaders await the channel endpoint permission, and then pass it on.

We can then prove the following specification of the relay procedure:

$$\left\{c \rightarrowtail (\mathsf{del\_prot}\ i\ i'\ c') * (i = i' \Rightarrow c' \rightarrowtail \mathsf{relay\_prot})\right\}\ \mathsf{relay}\ c\ c'\ i\ i'\ \{\mathsf{True}\}$$

In particular, if the participant is the leader, it owns the c' channel endpoint permission with the initial relay protocol (as a result of the election), over which it can first relay the elected ID, after which point it sends it clockwise around the ring.

With the relay protocol we can verify the example with 4 participants, by proving protocol consistency of the following two protocol systems:

$$c_r \rightarrowtail ?[0]\ (i' : \mathbb{N})\ \langle i' \rangle.\ \mu rec.\ ?[0]\ \langle i' \rangle.\ rec$$
$$c_s \rightarrowtail \mathsf{relay\_prot}$$

$$c_0 \rightarrowtail \mathsf{init\_prot}\ 0\ (c_s \rightarrowtail \mathsf{relay\_prot})\ (\lambda i'.\ \mathsf{del\_prot}\ 0\ i'\ (c_s \rightarrowtail \mathsf{relay\_prot'}\ i'))$$
$$c_1 \rightarrowtail \mathsf{ring\_prot}\ 1\ (c_s \rightarrowtail \mathsf{relay\_prot})\ (\lambda i'.\ \mathsf{del\_prot}\ 1\ i'\ (c_s \rightarrowtail \mathsf{relay\_prot'}\ i'))$$
$$c_2 \rightarrowtail \mathsf{ring\_prot}\ 2\ (c_s \rightarrowtail \mathsf{relay\_prot})\ (\lambda i'.\ \mathsf{del\_prot}\ 2\ i'\ (c_s \rightarrowtail \mathsf{relay\_prot'}\ i'))$$
$$c_3 \rightarrowtail \mathsf{ring\_prot}\ 3\ (c_s \rightarrowtail \mathsf{relay\_prot})\ (\lambda i'.\ \mathsf{del\_prot}\ 3\ i'\ (c_s \rightarrowtail \mathsf{relay\_prot'}\ i'))$$

The protocol consistency of the second system relies on the fact that there is complete consensus on the elected leader, as participants would otherwise disagree on the protocol of the delegated coordinator channel, which is parametric in the elected leader. Our brute-force tactic automate the consistency proof entirely.

With the above protocol system we verify the program for 4 participants:

$$\{\mathsf{True}\}\ \mathsf{ring\_del\_prog}\ 4\ \{\mathsf{True}\}$$

This Hoare triple again guarantees that the program is safe to execute via our adequacy theorem.

3.2 Ring Leader Election Benchmark

Verifying an implementation of the ring leader election algorithm is a challenging task, while we covered some aspects, more remain to be done. We propose the following benchmark to challenge the verification community to develop more expressive and scalable verification systems. We believe that this benchmark is an interesting and challenging problem in addition to the canonical two- and three-buyer [16, 3] benchmarks in the literature on multiparty session types.

The benchmark is to verify an implementation of ring leader election, satisfying variations of the following categories of properties:

- **Algorithm**, *e.g.,* unique IDs, anonymous, randomized.
- **Implementation**, *e.g.,* λ-calculus, shared memory, π-calculus, low level distributed system.
- **Guarantees**, *e.g.,* safety, functional correctness, deadlock freedom, termination.
- **Consistency**, *e.g.,* brute-force procedure, manual, model checking, by construction.
- **Scalability**, *e.g.,* fixed participants/elections, arbitrary participants/elections.
- **Features**, *e.g.,* delegation, non-deterministically chosen participant IDs.

We verify Chang and Roberts [6]'s **algorithm** that uses unique IDs for each participant. We believe that anonymous and randomized variants will be more difficult to verify. We **implement** the algorithm in a λ-calculus-like language with shared-memory. We believe that this makes the verification more challenging than π-calculus, which can abstract over low-level details of the individual nodes. Although the abstractions provided by the π-calculus may in turn allow for supporting some of the other properties. We **guarantee** safety and functional correctness. We believe the latter is crucial to properly show that the appropriate leader is elected. Even so, we do not guarantee deadlock-freedom or termination, which are equally interesting properties. Our proofs rely on the brute-force procedure for proving protocol **consistency**, which impedes **scalability**. As a result, we only allow a fixed number of participants and concurrent elections. As expressivity seem to be at odds with automation of the protocol consistency proof, we find that properly scaling the implementation alongside challenging properties (such as functional verification) is a high-ranking benchmark. In the same vein, we encourage the addition of more **features** supported by the system, such as channel delegation, as they may further complicate the protocol consistency proof.

```
new_chan(n) ≜
  let m = new_mat n n None in
  ((m, 0), ..., (m, n − 1))
```

```
c[j].recv() ≜
  let (m, i) = c in
  let x = Xchg m_{j,i} None in
  match x with
  | None   ⇒ c[j].recv()
  | Some v ⇒ v
  end
```

```
c[j].send(v) ≜
  let (m, i) = c in
  m_{i,j} ← Some v;
  (rec f _. match ! m_{i,j} with
            | None   ⇒ ()
            | Some _ ⇒ f ()
            end) ()
```

Fig. 6. The Multris channel implementation

4 MODEL AND SOUNDNESS

In this section we cover how we obtained the rules shown in Fig. 3 and our adequacy theorem Theorem 2.1. We first cover how the Multris adequacy theorem is a direct instance of the adequacy theorem of Iris, as a result of our channels being a shallow embedding on top of the Iris HeapLang language (§ 4.1). We then show the implementation of the multiparty channels on top of the HeapLang language (§4.2). We then present how we defined the channel endpoint ownership and verified the multiparty channel rules of Multris (§4.3). We finally remark on how our multiparty dependent separation protocols are defined, and how we constructed the protocol consistency as a result (§4.4).

4.1 Inheritance of the Iris Adequacy Theorem

The multiparty channel endpoints of Multris are implemented directly on top of the HeapLang instantiation of Iris, and the proofs of their specifications are derived through the corresponding program logic. As a result, our adequacy theorem Theorem 2.1 is directly inherited from the Iris adequacy theorem:

Theorem 4.1. *A proof of* $\{\mathsf{True}\}\ e\ \{\mathsf{True}\}$ *implies that* e *is* ***safe****, i.e., if* $([e], \emptyset) \to_{\mathrm{t}}^{*} ([e_1 \dots e_n], h)$*, then for each* i *either* e_i *is a value or* (e_i, h) *can step.*

4.2 Multiparty Channel Implementation

The implementation of our shared-memory multiparty channels can be seen in Fig. 6. The implementation uses an $n \times n$ matrix m of references, where each reference i, j acts as a one-sized buffer over which participant i can send a value to participant j. Channel endpoints are represented as tuple (m, i), where m is the matrix, and i is the participant ID associated with the channel endpoint.

The **new_chan**(n) operation allocates the $n \times n$ matrix, and returns n participant tuples, with IDs $0 \dots (n-1)$.

The $c[j]$.**send**(v) operation synchronously sends value v by storing **Some** v in the send buffer $m_{i,j}$, and then spins until the value has been taken out by the receiver.

The $c[j]$.**recv**() operation synchronously receives the next incoming value, by atomically taking any value out of its inbound buffer $m_{j,i}$, using **Xchg** $m_{j,i}$ **None**. If the value was is **Some** v, the value v is simply returned, setting the reference to **None**. Otherwise, the receive loops, having made no changes to the state, as **Xchg** $m_{j,i}$ **None** is effectively a no-op.

For brevity's sake we omit details about the matrix library, which is relatively standard.

4.3 Verification of the Multiparty Channel Specifications

The Multris channel specifications are defined in terms of the channel endpoint ownership $c \rightarrowtail p$. To verify the specifications, we must thus first provide a definition for the resource. To do so, we leverage the Iris methodology; constructing a *ghost theory* that effectively yields a separation logic

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

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

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

$$\frac{\mathsf{prot\_ctx}\ \chi\ n \qquad P_1[\vec{x_1} := \vec{t_1}] \qquad \mathsf{prot\_own}\ \chi\ i\ (!\,[j]\ (\vec{x_1} : \vec{\tau_1})\ \langle v_1 \rangle \{P_1\}.\ p_1) \qquad \mathsf{prot\_own}\ \chi\ j\ (?[i]\ (\vec{x_2} : \vec{\tau_2})\ \langle v_2 \rangle \{P_2\}.\ p_2)}{\Rrightarrow \triangleright \exists (\vec{t_2} : \vec{\tau_2}).\ \mathsf{prot\_ctx}\ \chi\ n * \mathsf{prot\_own}\ \chi\ i\ (p_1[\vec{x_1} := \vec{t_1}]) * \mathsf{prot\_own}\ \chi\ j\ (p_2[\vec{x_2} := \vec{t_2}]) * (v_1[\vec{x_1} := \vec{t_1}]) = (v_2[\vec{x_2} := \vec{t_2}]) * P_2[\vec{x_2} := \vec{t_2}]}$$

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. 7. The Multris ghost theory.

approach to a state transition system which models the domain problem, and then connect it to the implementation. We achieve this by constructing the *Multris ghost theory*.

The Multris ghost theory. The Multris ghost theory captures a language-agnostic semantics of synchronous multiparty communication; governing how we can allocate fresh systems, and make semantically sound transitions, with respect to the underlying logic. This is made precise by the resources and rules, explained in the following text, and shown in Fig. 7.

The Multris ghost theory has two resources: $\mathsf{prot\_ctx}\ \chi\ n$ and $\mathsf{prot\_own}\ \chi\ i\ p$. The resources are associated with each other using the logical identifier χ, called a ghost name. The $\mathsf{prot\_ctx}\ \chi\ n$ fragment acts as a central coordinator, ensuring that everyone transitions according to the global consistency. It also captures the number of participants in the network n. The $\mathsf{prot\_own}\ \chi\ i\ p$ records the current protocol p of participant i.

The rules of the ghost state capture the transitions of the modeled state transition system. The updates are reflected via *ghost updates* $\Rrightarrow$, that can be applied during program verification, as made precise by the associated rules:

BUPD-WP

$$\frac{P * R \mathrel{-\!\!*} \mathsf{wp}\ e\ \{v.\ Q\}}{(\Rrightarrow P) * R \mathrel{-\!\!*} \mathsf{wp}\ e\ \{v.\ Q\}}$$

The PROTO-ALLOC rule allocates resources for a consistent system of protocols $\vec{p}$, with a fresh identifier χ. It returns a single $\mathsf{prot\_ctx}\ \chi\ n$, and a $\mathsf{prot\_own}\ \chi\ i\ p$ for each $i \mapsto p \in \vec{p}$. The PROTO-STEP rule reflects a synchronous transition; it requires the presence of the central coordinator $\mathsf{prot\_ctx}\ \chi\ n$, a sending participant $\mathsf{prot\_own}\ \chi\ i\ (!\,[j]\ (\vec{x_1} : \vec{\tau_1})\ \langle v_1 \rangle \{P_1\}.\ p_1)$, a corresponding receiving participant $\mathsf{prot\_own}\ \chi\ j\ (?[i]\ (\vec{x_2} : \vec{\tau_2})\ \langle v_2 \rangle \{P_2\}.\ p_2)$, and the resources specified by the sending protocol, for some term instantiation $P_1[\vec{x} := \vec{t}]$.

The ghost update yields an instantiation of the binders of the receiving protocol $(\vec{y} : \vec{\tau_2})$, evidence that the protocol values are equal $v_1[\vec{x_1} := \vec{t}] = v_2[\vec{x_2} := \vec{y}]$, and the resources of the receiving

$$
\begin{array}{l}
c \rightarrowtail p \triangleq \\
\quad \exists \chi, \vec{\gamma E}, m, n, i, p'. \\
\qquad c = (m, i) * \boxed{\mathsf{prot\_ctx}\ \chi\ n} * \mathsf{is\_matrix}\ m\ n\ n\ (\lambda i, j, \ell.\ \exists \gamma t.\ \boxed{\mathsf{chan\_inv}\ \chi\ \vec{\gamma E_i}\ \gamma t\ i\ j\ \ell}) \\
\qquad \triangleright (p' \sqsubseteq p) * \boxed{\bullet_E\ (\mathsf{next}\ p')}^{\vec{\gamma E_i}} * \boxed{\circ_E\ (\mathsf{next}\ p')}^{\vec{\gamma E_i}} * \mathsf{prot\_own}\ \chi\ i\ p' \\
\\
\mathsf{chan\_inv}\ \chi\ \gamma E\ \gamma t\ i\ j\ \ell \triangleq \\
\quad (\ell \mapsto \mathbf{None} * \mathsf{tok}\ \gamma t) \qquad \vee \quad (1) \\
\quad \left(\exists v, p.\ \ell \mapsto \mathbf{Some}(v) * \mathsf{prot\_own}\ \chi\ i\ (!\,[j]\ \langle v \rangle.\ p) * \boxed{\circ_E\ (\mathsf{next}\ p)}^{\gamma E}\right) \qquad \vee \quad (2) \\
\quad (\exists p.\ \ell \mapsto \mathbf{None} * \mathsf{prot\_own}\ \chi\ i\ p * \boxed{\circ_E\ (\mathsf{next}\ p)}^{\gamma E}) \qquad (3)
\end{array}
$$

Fig. 8. The channel endpoint ownership definition

protocol $P_2[\vec{x}_2 := \vec{y}]$. The update preserves the central coordinator prot_ctx $\chi\ n$, and updates the protocols to their respective tails prot_own $\chi\ i\ (p_1[\vec{x}_1 := \vec{t}])$ and prot_own $\chi\ j\ (p_2[\vec{x}_2 := \vec{y}])$; the sending protocol continues with the given term instantiation $\vec{t}$, while the receiving protocol continues with the returned instantiation of its binders $\vec{y}$.

The rules PROTO-VALID and PROTO-VALID-PRESENT lets us infer that indices of the participants, and correspondent participants, are bounded by the size of the network.

Finally, the rule PROTO-LE captures that prot_own $\chi\ i\ p$ is closed under the subprotocol relation.

In the following section we demonstrate how the Multris ghost theory was used to prove the rules of the HeapLang instantiation of Multris.

Channel endpoint ownership. The crux of verifying the channel implementation is to come up with the right definition for the channel endpoint ownership, that reflects the implementations correspondence to the Multris ghost theory. To achieve this, we consider the channel implementation as a state transition system, with the following three states: (1) No value has been sent, (2) a value has been sent but not received, and (3) the value has been received but the sender has not finished synchronization. Each of the transitions can then be associated with a transfer of resources, to facilitate the transition in the PROTO-STEP rule of the Multris ghost theory. In particular, we need to transfer the protocol ownership of the sender to the receiver when putting the value in the buffer (go from state (1) to (2)). The receiver can then obtain resources when witnessing the value in the buffer, use them to apply the PROTO-STEP rule, and transfer the updated protocol ownership of the sender back (going from state (2) to (3)) when resetting the buffer. Finally, when the sender finishes the synchronization, by witnessing that the buffer has been reset, it can obtain the updated protocol ownership (going from state (3) back to (1)).

To achieve this formally, we use the standard Iris methodology of encoding such state transition systems using invariants: propositions that hold in between any steps of the program. In particular, we define the invariant for synchronous transfer (`chan_inv`) as shown in Fig. 8. The first state simply captures that the value has not been sent ($\ell \mapsto \mathbf{None}$), alongside an exclusive token ($\mathsf{tok}\ \gamma t$) that lets use distinguish it from the final state. The second state captures that the value has been sent ($\ell \mapsto \mathbf{Some}(v)$), along with the satisfied protocol ownership of the sender ($\mathsf{prot\_own}\ \chi\ i\ (!\,[j]\ \langle v \rangle.\ p)$), and a piece of ghost state that lets us remember the protocol tail when it is returned ($\boxed{\circ_E\ (\mathsf{next}\ p)}^{\gamma E}$). The final state captures that the value has been read, and reset to the empty state ($\ell \mapsto \mathbf{None}$), along with the returned protocol ownership of the sender ($\mathsf{prot\_own}\ \chi\ i\ p$), updated to the original tail (p), as evidenced by the associated ghost state ($\boxed{\circ_E\ (\mathsf{next}\ p)}^{\gamma E}$).

With this invariant, we can define the channel endpoint ownership as seen in Fig. 8. The definition captures that:

(1) The endpoint is the tuple (m, i) of the matrix m, and the participant id i.
(2) Shared access (via an invariant) to the protocol context prot_ctx χ n.
(3) The synchronization invariant of all participant pairs i, j of the matrix m.
(4) The endpoint ownership is closed under subprotocols $p' \sqsubseteq p$.
(5) The unification ghost state for the transferred protocol: $\boxed{\bullet_E\ (\mathsf{next}\ p')}^{\gamma E}$ and $\boxed{\circ_E\ (\mathsf{next}\ p')}^{\gamma E}$.
(6) The protocol ownership of the participant prot_own χ i p.

Channel endpoint verification. The proof of the channel endpoint rules shown in Fig. 3 follows from the channel endpoint definition and the Multris ghost theory.

The Wp-new rule follows trivially from allocating all of the appropriate ghost state for each of the channel endpoints. This includes the Multris ghost theory, and the protocol context invariant, the unification ghost state of each endpoint, the exclusive tokens used in the synchronization invariants, and the synchronization invariants themselves.

To prove the Wp-send rule we follow the transitions of the invariant from state (1) to (2), when we store the sent value **Some**(v), and then from (3) to (1), when we synchronize, by reading that the value has been reset to **None**. We can first infer that we are in state (1) of the invariant, as we have local ownership of the exclusive protocol (prot_own χ i (**!** $[j]$ $\vec{x}$:$\vec{\tau}$ $\langle v \rangle \{P\}.\ p$)). We first satisfy the protocol locally, using the proto-le rule, to instantiate the protocol binders, and provide the protocol resources P. We then update the unification pair to the protocol tail $\boxed{\bullet_E\ (\mathsf{next}\ p)}^{\gamma E}$ and $\boxed{\circ_E\ (\mathsf{next}\ p)}^{\gamma E}$. We can then satisfy invariant transition from (1) to (2), by giving up the satisfied protocol ownership, and one part of the unification ghost state, while taking out the exclusive token tok γt. We subsequently satisfy the invariant transition from (3) to (1), when observing that the reference is **None**, at which point we conclude that we are in (3), via tok γt, after which point we exchange the returned protocol ownership and unification ghost state, with the exclusive token.

To prove the Wp-recv rule we follow the transitions of the invariant from state (2) to (3), when we atomically read the stored value **Some**(v) and update it to **None** via **Xchg**. In particular, we infer that we are in state (2) as the value has been stored. We then take out the satisfied protocol ownership of the sender, and use it along with the local receiving protocol ownership, and the proto-step rule. We immediately put the updated protocol of the sender back in the invariant, along with the unification ghost state, before closing the invariant. We can immediately satisfy the postcondition with the residuals of the proto-step rule.

4.4 Protocol Consistency and Validation of the Multris Ghost Theory

In this section we briefly remark on the model of multiparty dependent separation protocols, the definition of our protocol consistency, and on the multiparty subprotocol relation. We then cover how we validated the Multris ghost theory as a result.

Multiparty dependent separation protocols. The multiparty dependent separation protocols are defined similarly to the binary variant of Actris [12, §9.1]. In particular, we simply add a participant identifier to the message constructor, to determine the participant with which we

communicate:

$$\begin{aligned}
\text{action} &::= \mathbf{send} \mid \mathbf{recv} \\
\text{iProto} &\cong 1 + (\text{action} \times \mathbb{N} \times (\text{Val} \to \blacktriangleright\text{iProto} \to \text{iProp})) \\
\mathbf{end} &\triangleq \texttt{inl}() \\
\mathbf{!}\,[i]\ (\vec{x}:\vec{\tau})\ \langle v\rangle\{P\}.\ p &\triangleq \texttt{inr}(\mathbf{send}, i, (\lambda w, p'.\ \exists \vec{x}:\vec{\tau}.\ (v = w) * P * (p' = \text{next}\ p))) \\
\mathbf{?}\,[i]\ (\vec{x}:\vec{\tau})\ \langle v\rangle\{P\}.\ p &\triangleq \texttt{inr}(\mathbf{recv}, i, (\lambda w, p'.\ \exists \vec{x}:\vec{\tau}.\ (v = w) * P * (p' = \text{next}\ p)))
\end{aligned}$$

The protocols are either terminating ($\texttt{inl}()$), or exchange a message ($\texttt{inr}\ \ldots$). Message exchanges are defined as predicates over the exchanged values and the exchanged tails, to facilitate the dependent binders. We refer the interested reader to the Hinrichsen et al. [12, §9.1].

Protocol consistency. With the multiparty variant of the dependent separation protocols, we give an *intensional* definition to synchronous protocol consistency as follows:

$$\begin{aligned}
\textsc{consistent}\ \vec{p} &\triangleq (\forall i.\ \textsc{present}\ \vec{p}\ i) * (\forall i, j.\ \textsc{dual}\ \vec{p}\ i\ j) \\
\textsc{present}\ \vec{p}\ i &\triangleq \forall a, j, \Phi.\ \vec{p}[i] = \texttt{inr}(a, j, \Phi) \mathrel{-\!\!*} j \in \vec{p} \\
\textsc{dual}\ \vec{p}\ i\ j &\triangleq \forall \Phi_1, \Phi_2.\ \vec{p}[i] = \texttt{inr}(\mathbf{send}, j, \Phi_1) \mathrel{-\!\!*} \vec{p}[j] = \texttt{inr}(\mathbf{recv}, i, \Phi_2) \mathrel{-\!\!*} \\
&\qquad \forall v_1, p_1.\ \Phi_1\ v_1\ (\text{next}\ p_1) \mathrel{-\!\!*} \\
&\qquad\qquad \exists v_2, p_2.\ \Phi_2\ v_2\ (\text{next}\ p_2) * \triangleright(\textsc{consistent}\ (\vec{p}[i := p_1][j := p_2]))
\end{aligned}$$

This protocol consistency definitionally captures the exact consistency rules shown in Fig. 5, provided its intensional nature.

We remark that this definition is closely resembling the intensional definition of synchronous multiparty session type consistency, given by Scalas and Yoshida [33]. Here the subtyping relation is reflected via separation implication, the recursion is reflected via the later, and the choice is reflected in terms of the dependent binders.

Subprotocols. The subprotocol definition of the multiparty dependent separation protocols is much similar to the binary one presented in Hinrichsen et al. [12, §9.2]. However, since our setting is synchronous, we do not have the case pertaining to swapping, where receiving protocols may be related to sending ones. Even so, we still inherit all of the remaining rules (used to validate Sub-send and Sub-recv). Similar to the binary variant of Actris, this allow us to carry out proofs of programs such as the one pertaining to protocol compositionality, presented in Hinrichsen et al. [12, §6.3].

Validating the Multris ghost theory. With the definition of protocol consistency in hand, we can define the resources of the Multris ghost theory akin to the original binary Actris ghost theory in Hinrichsen et al. [12, §9.4]; letting the context govern protocol consistency of all protocols, and let the fragments reflect the current state of the protocols:

$$\begin{aligned}
\text{prot_ctx}\ \chi\ n &\triangleq \exists \vec{p}.\ |\vec{p}| = n * \boxed{\bullet \vec{p}}^{\chi} * \triangleright\,\textsc{consistent}\ \vec{p} \\
\text{prot_own}\ \chi\ i\ p &\triangleq \exists p'.\ \boxed{\circ (i, p)}^{\chi} * \triangleright(p' \sqsubseteq p)
\end{aligned}$$

These resources, along with the definition of protocol consistency, is sufficient for proving the rules of the Multris ghost theory. In particular, we can prove the proto-alloc rule as we can freely allocate the necessary ghost resources, and wrap them up with the provided protocol consistency. The proto-step rule follows directly from dual. The proto-valid rule follows from the underlying list ghost state. The proto-valid-present rule follows from present. Finally, the proto-le rule follows directly, since we explicitly close the protocols of prot_own under the subprotocol relation.

Component	Section(s)	LOC
Multris domain model	§4.4	309
Protocol consistency and Multris ghost theory	§4.3, §4.4	1377
Channel implementation and verification	§4.2, §4.3	370
Matrix library	§4.2	221
Proofmode tactics	§5	405
Examples:		
• Basic examples from this paper	§2	290
• Two-Buyer Example	§3	133
• Three-Buyer Example	§3	198
• Ring Leader Election	§3	356
Total		3659

Table 1. Overview of the Multris Coq mechanization

5 MECHANIZATION

All definitions, theorems, and examples in this paper have been mechanized in Coq using the Iris framework. The full sources are available in our artifact [21].

While we were able to inherit a lot of the structure of the mechanization of the binary variant of Actris, our work incurred significant mechanization effort. In particular, defining the our novel notion of multiparty protocol consistency and proving the associated lemmas, verifying the new (synchronous) multiparty ghost state, and the verification of the multiparty channel specifications.

Tactics. We provide tactic support à la Actris for the multiparty channel primitives, enabling program proofs that follow primarily from symbolic execution. The primary proof effort for a user thus arises from the CONSISTENT $\vec{p}$ obligation of WP-NEW. To alleviate the effort, we provide a brute-force tactic that (1) finds all synchronizing pairs, (2) abstracts over the binders and resources of the sender, and (3) tries to resolve the binders and resources of the receiver via the Iris Proof Mode tactics for framing. The tactic automatically retains ownership of any resources that are not used, thus supporting our non-trivial protocols with implicit resource transfer. Despite the brute-force nature of the tactic, we managed to automate all of the consistency obligations of the examples in this paper within a reasonable time frame (less than 30 seconds on a MacBook Pro 13 inch, M1, 2020). The tactic proved invaluable, avoiding a significant amount of boilerplate proof.

6 RELATED AND FUTURE WORK

6.1 Session Types

Session types were developed by Honda et al. [14, 15] as a typing discipline for message-passing programs. Initially, session types were used to type check two-party communication, but they were later extended to the multiparty setting by Honda et al. [16]. Traditionally, multiparty session types are *top-down*—they rely on a global type that describes all interactions of the network, which is projected to a local type for each participant. These local types are subsequently used to type check individual processes.

The existence of a global type guarantees that the projected local types are consistent, but it turns out that not all consistent systems of local types can be projected from a global type [33]. Therefore, Scalas and Yoshida [33] proposed the *bottom-up* method to consistency, which checks consistency of local types directly, rather than projecting them from a global type. Our approach

is inspired by the bottom-up method to consistency, and we extend it to a program logic setting, where we can reason about functional correctness of the programs via dependent binders, as well as resource transfer and delegation. Our extension of consistency to this richer setting allows us to express protocols such as the leader election protocol in §3.

Due to the intricacies of multiparty session types, there has been recent interest in their mechanization [5, 18]. Multris is also in this vein, but we focus on program logics rather than type systems. These works, on the other hand, focus on type systems and their soundness properties, and do not consider the functional correctness of the programs, nor include separation logic or program logic for verification.

6.2 Design-by-Contract and Refinement Types for Multiparty Message Passing

Based on multiparty session types, Bocchi et al. [4] constructed a design-by-contract verification system in which one can specify global types enriched with binders and assertions, which can then be projected to local types. Our program logic is more expressive in the following ways: (1) our consistency relation is more flexible than global types (see §6.1), (2) our protocols are truly dependent, as the continuation can depend on the binders via an arbitrary meta-level function, whereas in Bocchi et al. [4] the continuation is fixed, and (3) our protocols carry separation logic assertions, which allows us to reason about resource sharing and delegation. Furthermore, Multris is a program logic, and is embedded in Coq, with tactics for carrying out reasoning in the logic. This has the disadvantage that our logic is not decidable, and manual proof is required, but the advantage that it is more expressive and can be used to carry out arbitrarily complex mathematical reasoning to establish functional correctness of the program.

Zhou et al. [38] present a refinement type system for message-passing programs, which allows for specifying and verifying message-passing programs with dependent types. Their work enables the use of the F* proof assistant [34] to manually reason about refinement whenever the proof obligations cannot be automatically discharged by the SMT solver. The ways in which Multris differs from Zhou et al. [38] is similar to the ways in which Multris differs from Bocchi et al. [4]. A key similarity is that Zhou et al. [38] make use of a proof assistant (F*), as do we (Coq): the facilities of the proof assistant are used to reason about refinements (in the case of Zhou et al. [38]) and to reason about separation logic assertions (in our case). Besides the distinction between a refinement type system and a program logic, a key distinction is that Multris establishes a fully developed foundational meta-theory, including a machine-checked adequacy theorem for the logic w.r.t. an operational semantics. As such, the effort of reasoning carried out in the Multris logic "pays off" in the sense that it establishes a formal functional correctness theorem of the program in question inside the proof assistant, via our adequacy theorem (Theorem 2.1). The adequacy theorem is a key contribution of Multris. The analogue of this in Zhou et al. [38] would be to establish a soundness theorem for the refinement type system w.r.t. the operational semantics of the language, which is not done in their work. In addition to their correctness properties, Zhou et al. [38] guarantees deadlock freedom, which is beyond the reach of the step-indexed approach to partial correctness that we use in Multris (see §6.4).

6.3 Program Logics for Binary Message Passing

Hinrichsen et al. [11] developed Actris—a program logic for message-passing programs, which is based on separation logic and Iris. This work is closely related to ours, in that it also uses separation logic to reason about message-passing programs via dependent protocols. The key distinction between Actris and Multris is that the first focuses on two-party message-passing, while we focus on multiparty message-passing. Jacobs et al. [19] showed that construction of a program logic for two-party case can be simplified. The multiparty case is significantly more complex, as it requires

reasoning about the interleaving of multiple processes, and the sharing of resources between them. As such, the method of simplification used in Jacobs et al. [19] is not directly applicable to the multiparty case. Instead, a more complex model is required to prove soundness of the logic, as we have done in §4.

6.4 Future work

Multris is a first step towards a comprehensive program logic for multiparty message-passing programs. There are many directions in which this work can be extended. One direction is to consider asynchronous communication, where messages can be sent and received at arbitrary times. We chose to focus on synchronous communication in this work, as it makes protocol consistency proofs simpler: Scalas and Yoshida [33] point out that consistency of local types is decidable in the synchronous case, but not in the asynchronous case. Our tactics make use of this advantage, and would need to be extended to (heuristically) handle the asynchronous case. In general, we would like to consider alternative approaches to establish consistency of local types, with better abstraction and modularity, as well as compositionality, or with even more automation, such as model checking. Another promising approach to protocol consistency is the top-down approach (defining a global protocol specification which is statically projected to endpoints). Li et al. [28] show that global protocols can support expressive local projections via synthesis. It is interesting to investigate whether the top-down approach scales to functional verification.

An asynchronous version of the logic would also allow us to consider asynchronous subprotocols such as in Hinrichsen et al. [12], which allow swapping of sends over receives as first introduced by Mostrous et al. [30]. Going beyond asynchrony, we would like to consider distributed systems, where processes are running on different machines, and communicate over a network such as in Gondelman et al. [10]. Another direction is to restrict the logic to ensure a stronger property, such as deadlock freedom à la Jacobs et al. [20]. Finally, we would like to apply the method of logical type soundness in Iris [36] (which Hinrichsen et al. [13] used for the binary session types) to obtain a semantic type safety proof for a state-of-the-art multiparty type system such as the one by Jacobs et al. [18].

DATA-AVAILABILITY STATEMENT

The Coq development for this paper can be found in [21].

ACKNOWLEDGMENTS

REFERENCES

[1] Andrew W. Appel. 2001. Foundational Proof-Carrying Code. In *LICS*. https://doi.org/10.1109/LICS.2001.932501
[2] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *POPL*. https://doi.org/10.1145/1190216.1190235
[3] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2008. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR*. https://doi.org/10.1007/978-3-540-85361-9_33
[4] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. 2010. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In *CONCUR*. https://doi.org/10.1007/978-3-642-15375-4_12
[5] David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. 2021. Zooid: A DSL for Certified Multiparty Computation: From Mechanised Metatheory to Certified Multiparty Processes. In *PLDI*. https://doi.org/10.1145/3453483.3454041
[6] Ernest Chang and Rosemary Roberts. 1979. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *CACM* (1979). https://doi.org/10.1145/359104.359108
[7] Florin Craciun, Tibor Kiss, and Andreea Costea. 2015. Towards a Session Logic for Communication Protocols. In *ICECCS*. https://doi.org/10.1109/ICECCS.2015.33

[8] Adrian Francalanza, Julian Rathke, and Vladimiro Sassone. 2011. Permission-Based Separation Logic for Message-Passing Concurrency. *LMCS* (2011). https://doi.org/10.2168/LMCS-7(3:7)2011

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

[10] 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

[11] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: Session-Type Based Reasoning in Separation Logic. POPL (2020). https://doi.org/10.1145/3371074

[12] 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

[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] Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR*. https://doi.org/10.1007/3-540-57208-2_35

[15] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP*. https://doi.org/10.1007/BFb0053567

[16] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *POPL*. https://doi.org/10.1145/1328438.1328472

[17] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* (2016). https://doi.org/10.1145/2827695

[18] 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

[19] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2023. Dependent Session Protocols in Separation Logic from First Principles (Functional Pearl). ICFP (2023). https://doi.org/10.1145/3607856

[20] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2024. Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing. *Proc. ACM Program. Lang.* 8, POPL, Article 47 (jan 2024), 33 pages. https://doi.org/10.1145/3632889

[21] Jonas Kastberg Hinrichsen, Jules Jacobs, Robbert Krebbers. 2024. Supplementary material of "Multris: Functional Verification of Multiparty Message Passing in Separation Logic".

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

[23] 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

[24] 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

[25] 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

[26] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP*. https://doi.org/10.1007/978-3-662-54434-1_26

[27] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*. https://doi.org/10.1145/3009837.3009855

[28] Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. 2023. Complete Multiparty Session Type Projection with Automata. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 13966)*, Constantin Enea and Akash Lal (Eds.). Springer, 350–373. https://doi.org/10.1007/978-3-031-37709-9_17

[29] Étienne Lozes and Jules Villard. 2012. Shared Contract-Obedient Endpoints. In *ICE*. https://doi.org/10.4204/EPTCS.104.3

[30] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 316–332. https://doi.org/10.1007/978-3-642-00590-9_23

[31] Hiroshi Nakano. 2000. A modality for recursion. In *LICS*. https://doi.org/10.1109/LICS.2000.855774

[32] Wytse Oortwijn, Stefan Blom, and Marieke Huisman. 2016. Future-based Static Analysis of Message Passing Programs. In *PLACES*. https://doi.org/10.4204/EPTCS.211.7

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

[34] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F*. In *POPL*. 256–270. https://doi.org/10.1145/2837614.2837655

[35] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. 2013. Why Do Scala Developers Mix the Actor Model with other Concurrency Models?. In *ECOOP (LNCS, Vol. 7920)*. https://doi.org/10.1007/978-3-642-39038-8_13

[36] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. Manuscript.

[37] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *ASPLOS*. 865–878. https://doi.org/10.1145/3297858.3304069

[38] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 148 (nov 2020), 30 pages. https://doi.org/10.1145/3428216