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 functional correctness of programs that combine multiparty message-passing communication with shared-memory concurrency. The foundation of our work is a novel concept of multiparty *protocol consistency*, which guarantees safe communication among a set of parties, provided each party adheres to its prescribed protocol. Our concept of protocol consistency is inspired by the bottom-up approach for multiparty session types. However, by considering it in the context of separation logic instead of a type system, we go further in terms of generality by supporting new notions of *implicit transfer of knowledge* and *implicit transfer of resources*. 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.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Separation logic**.

Additional Key Words and Phrases: Message passing, multiparty, session types, separation logic, Iris, Coq

ACM Reference Format:
Jonas Kastberg Hinrichsen, Jules Jacobs, and Robbert Krebbers. 2024. Multris: Functional Verification of Multiparty Message Passing in Separation Logic. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 322 (October 2024), 29 pages. https://doi.org/10.1145/3689762

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 [20, 21], 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 [8, 13, 34, 11, 37, 47, 16, 17, 26]. We categorize these as:

(1) Whether the verification system supports message-passing communication between *two parties* (binary) only, or between *multiple parties* (multiparty) [22, 23].

Authors' Contact Information: Jonas Kastberg Hinrichsen, Aarhus University, Denmark, hinrichsen@cs.au.dk; Jules Jacobs, Cornell University, USA, julesjacobs@gmail.com; Robbert Krebbers, Radboud University Nijmegen, The Netherlands, mail@robbertkrebbers.nl.

ACM 2475-1421/2024/10-ART322
https://doi.org/10.1145/3689762

(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 message-passing programs in Go and Scala [42, 45] 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 [14].
(4) Whether the verification system comes with a *foundational proof* [3] 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 [38].

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. [8] 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. [47] 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 [16, 17, 25] in the Iris framework in Coq [29, 27, 31, 28] covers (2–4), but not (1)—Actris only supports message passing between two parties.

This paper works towards closing this gap by presenting **Multris**, which is (to our knowledge) the most powerful verification system for message passing to date, and support reasoning about *multiparty message passing* combined with shared memory and higher-order message passing. Multris features a foundational soundness proof in Coq using Iris. 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:**
$\texttt{let}\, l = \texttt{ref}\, 40\, \texttt{in}$	$\texttt{let}\, x = c_B[A].\texttt{recv}()\, \texttt{in}$	$\texttt{let}\, l = c_C[B].\texttt{recv}()\, \texttt{in}$
$c_A[B].\texttt{send}(l);\ c_A[C].\texttt{recv}();$	$c_B[C].\texttt{send}(x)$	$l \leftarrow !\, l + 2;$
$\texttt{assert}(!\, l == 42)$		$c_C[A].\texttt{send}()$

The individual parties operate on their respective *channel endpoints* c_i, over which they send and receive messages to/from other parties, via $c_i[j].\texttt{send}(v)$ and $c_i[j].\texttt{recv}()$, respectively. 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* [22, 23] 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 can 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:

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

An important ingredient of the Multris protocols is the assertion $\{P\}$. Unlike prior work on multiparty verification [8, 47], 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 $\{c_i \rightarrowtail p_i\}$ **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* [22, 23] and *bottom-up* [38]. 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.

In the context of type systems, Scalas and Yoshida [38] already pointed out that not all consistent systems of local protocols can be projected from a global protocol, *i.e.,* the bottom-up approach is more general than the top-down approach. By considering a separation logic instead of a type system, we can use the binders $\vec{x}:\vec{\tau}$ and assertions $\{P\}$ in protocols to go even further in terms of generality—we allow for *implicit transfer of information* and *implicit transfer of resources*.

To demonstrate these notions, let us take a look at the example. Party A sends a reference $(\ell : \mathsf{Loc})$ to Party B, who forwards that reference to Party C. Since Party B just forwards the reference to Party C, protocol p_B just says that Party B receives and forwards an arbitrary value $(v : \mathsf{Val})$. That is, protocol p_B hides to Party B that it needs to forward a reference, and instead this information is implicitly transferred from Party A to Party C. Similarly, since Party B does not need access to the stored value of the reference, there are no assertions $\{P\}$ in protocol p_B. That is, the ownership $\ell \mapsto x$ is implicitly transferred from Party A to Party C.

Our key idea to enable 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 keeps track of the information and takes ownership of resources that are implicitly transferred. Note that while this concrete example is somewhat artificial, it demonstrates 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 [38] show that protocol consistency is undecidable for an asynchronous semantics of simply-typed multiparty session types (with recursion). However, since the binders $\vec{x}:\vec{\tau}$ and assertions $\{P\}$ in protocols range over arbitrary objects in higher-order logic and may contain 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* [36, 5].

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 [31, 30] 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* [3] 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.

We emphasize that the Multris proof rules for **`send`** and **`recv`** closely resemble those of Actris—the only difference has to do with the participant annotations $[i]$ since we are in the multiparty instead of the binary case. We consider this similarity to be a strength of Multris—with limited extensions to the surface logic, we can now verify multiparty programs. This is similar to the way the protocols and typing rules of binary session types are extended to the multiparty case. Yet, the soundness proof of Multris is fundamentally different from that of Actris. It involves a different channel implementation, a new notion of protocol consistency (which does not exist in the binary case), and new corresponding meta-theory and Coq tactics.

Contribution #4: Multiparty verification benchmark. Most literature on multiparty session types have used the two- and three-buyer protocols [22, 6] as canonical benchmarks. While we do support these benchmarks (and have mechanized them, see our accompanying artifact [18]), we propose a new benchmark based on Chang and Robert's [10] 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 [18].

Limitations. Similar to Iris (and other concurrent program logics that build on it) we verify *partial correctness*, which means that we do not prove termination nor deadlock freedom. Proving termination of concurrent programs is an open problem for higher-order logics such as Iris, even in the absence of message passing. Deadlock freedom has only recently been investigated for Iris-based logics in a context where the message-passing operations are primitives [26], instead of being implemented using low-level heap operations as done in this work.

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, permitting programs such as the following:

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

§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, we establish that $\textbf{assert}(x = 150)$ cannot fail.

§2.3 Extends the 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*. For instance, checking that the protocols (p_0, p_1, p_2) above are consistent involves the proof obligation $(x + 20) + 30 = x + 50$.

§2.6 Presents a brute-force procedure for automatically resolving the large majority of proof effort associated with verifying protocol consistency assertions.

These sections go into each component 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 the endpoints $(c_0, \ldots, c_{n-1})$.
$c_i[j].\textbf{send}(v)$ sends a value v to party j via endpoint c_i
$c_i[j].\textbf{recv}()$ receives a value from party j via endpoint c_i

The communication structure is such that a $c_i[j].\textbf{send}(v)$ is matched by a $c_j[i].\textbf{recv}()$. 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.

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

Fig. 1. The grammar of the Multris programming language.

The $\texttt{fork}\,\{e\}$ construct forks a new thread to execute e. The $\texttt{assert}(e)$ construct asserts that the expression e evaluates to $\texttt{true}$. 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 $\texttt{false}$. 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 ℓ 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 lets us prove correctness of 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 $\{P\}\ e\ \{\Phi\} \triangleq P \vdash \mathrm{wp}\ e\ \{\Phi\}$. The Multris separation logic has an adequacy theorem that states that if we can derive a weakest precondition assertion $\mathrm{wp}\ e\ \{v.\, \phi(v)\}$, where $\phi(v)$ is a purely logical assertion, then the program e will not crash, and any returned value v will satisfy the predicate $\phi(v)$. This is formally stated as:

Theorem 2.1 (Multris Adequacy). *A proof of* $\mathrm{wp}\ e\ \{v.\, \phi(v)\}$ *implies that* e *is* ***safe****, i.e., if* $([e], \emptyset) \to_{\mathrm{t}}^{*} ([e_0 \ldots e_n], h)$*, then for each* $i \le n$ *either* e_i *is a value or* (e_i, h) *can step. Furthermore, any returned value* v *of* e *satisfies* $\phi(v)$*.*

The notion of safety depends on the operational semantics $(\vec{e}, h) \to_{\mathrm{t}}^{*} (\vec{e}', h')$ 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,

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:

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

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

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.

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 [29, 27, 28, 31, 32], and as such inherits many of its features. The basic rules of Iris are shown in Fig. 2, and include rules for reasoning about weakest preconditions and mutable references of the heap via $\ell \mapsto v$. The star on the right side of an inference rule $\frac{P}{Q}*$ indicates that the rule is a separating implication inside the separation logic, that is, $\vdash (P \mathrel{-\!\!*} Q)$ holds true. This is stronger than $\frac{P}{Q}$ without a star, which indicates a meta-level implication $(\vdash P) \Rightarrow (\vdash Q)$. Multiple premises are joined via separating conjunction, that is $\frac{P \quad Q}{R}*$ corresponds to $\vdash ((P * Q) \mathrel{-\!\!*} R)$. The only rule without a star is the Löb rule, where the stronger rule $\frac{\triangleright P \mathrel{-\!\!*} P}{P}*$ does not hold. Even so, the given Löb rule is strong enough for all our use cases. We refer the reader to Jung et al. [28] for a detailed explanation of the rules of Iris.

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 ownership of the reference ℓ, that it initially sent to party 1, which 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. The logic also supports recursive protocols with parameters (*i.e.,* fixpoints over $A \to \mathsf{iProto}$).

The Multris protocol language fully integrates with the surrounding higher-order separation logic. As a result, we can leverage functionality of higher-order logic, such as pattern matching, allowing for branching structures *inside* the protocols, and define protocols such as the following:

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

Here, 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 leverage this type of branching protocol in §3.1.

We now cover how the rules of the Multris logic guarantee 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 $n > 0$ parties using **new_chan**(n), we can pick n consistent protocols $(p_0, \ldots, p_{n-1})$ and get an n-ary separation

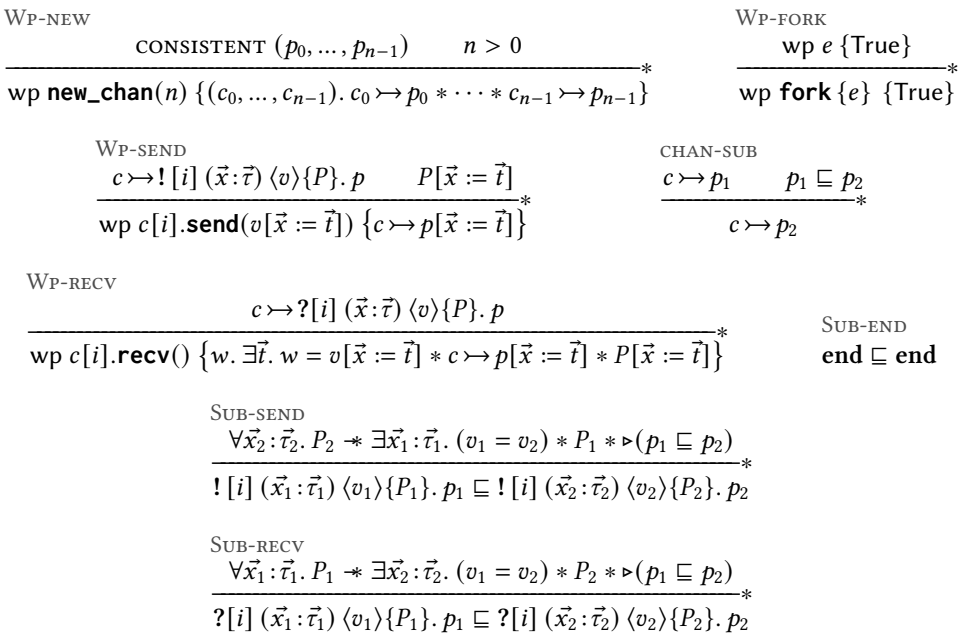

Fig. 3. The Multris rules for multiparty message-passing concurrency.

conjunction of channel ownership assertions $c_0 \rightarrowtail p_0 * \cdots * c_{n-1} \rightarrowtail p_{n-1}$ 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 $c \rightarrowtail ![i]\ (\vec{x}:\vec{\tau})\ \langle v \rangle \{P\}.\ p$, 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 $c \rightarrowtail p[\vec{x} := \vec{t}]$.

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 to fork a thread e, it is enough to verify that e does not crash.

Rule CHAN-SUB**:** This rule captures that for any owned channel endpoint with protocol p_1, we can instead use its subprotocol p_2 (where $p_1 \sqsubseteq p_2$).

Rules SUB-SEND**,** SUB-RECV**,** SUB-END**:** These rules capture how to prove the subprotocol relation. 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 these rules to prematurely "satisfy" sending protocols. For example, by giving up ownership of $\ell \mapsto 42$, we can prove $![i]\ (\ell : \mathrm{Loc}, x : \mathbb{Z})\ \langle \ell \rangle \{\ell \mapsto x\}.\ p \sqsubseteq ![i]\ \langle \ell \rangle.\ p$

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.

$$p_0 \triangleq \,!\,[1]\ (\ell : \mathrm{Loc})(x : \mathbb{Z})\ \langle \ell \rangle \{\ell \mapsto x\}.\ ?[2]\ \langle () \rangle \{\ell \mapsto x + 50\}.\ \mathbf{end}$$
$$p_1 \triangleq\ ?[0]\ (\ell : \mathrm{Loc})(x : \mathbb{Z})\ \langle \ell \rangle \{\ell \mapsto x\}.\ !\,[2]\ \langle \ell \rangle \{\ell \mapsto x + 20\}.\ \mathbf{end}$$
$$p_2 \triangleq\ ?[1]\ (\ell : \mathrm{Loc})(x : \mathbb{Z})\ \langle \ell \rangle \{\ell \mapsto x\}.\ !\,[0]\ \langle () \rangle \{\ell \mapsto x + 30\}.\ \mathbf{end}$$

$$\xrightarrow{c_0 \text{ sends to } c_1 \text{ (message: } \ell\text{, resources: } \ell \mapsto x)}$$

$$p_0' \triangleq\ ?[2]\ \langle () \rangle \{\ell \mapsto x + 50\}.\ \mathbf{end}$$
$$p_1' \triangleq\ !\,[2]\ \langle \ell \rangle \{\ell \mapsto x + 20\}.\ \mathbf{end}$$
$$p_2 \triangleq\ ?[1]\ (\ell : \mathrm{Loc})(x : \mathbb{Z})\ \langle \ell \rangle \{\ell \mapsto x\}.\ !\,[0]\ \langle () \rangle \{\ell \mapsto x + 30\}.\ \mathbf{end}$$

$$\xrightarrow{c_1 \text{ sends to } c_2 \text{ (message: } \ell\text{, resources: } \ell \mapsto x + 20)}$$

$$p_0' \triangleq\ ?[2]\ \langle () \rangle \{\ell \mapsto x + 50\}.\ \mathbf{end}$$
$$p_1'' \triangleq\ \mathbf{end}$$
$$p_2' \triangleq\ !\,[0]\ \langle () \rangle \{\ell \mapsto x + 50\}.\ \mathbf{end}$$

$$\xrightarrow{c_2 \text{ sends to } c_0 \text{ (message: } ()\text{, resources: } \ell \mapsto x + 50)}$$

$$p_0'' \triangleq\ \mathbf{end}$$
$$p_1'' \triangleq\ \mathbf{end}$$
$$p_2'' \triangleq\ \mathbf{end}$$

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

Note that the rules of Multris resemble those of (binary) Actris. They are generalized by adding participant annotations $[i]$, similarly to how binary session types are extended to the multiparty case. The differences are twofold. First, Wp-new involves our novel notion of protocol consistency, which does not exist in the binary case. Second, the model (§4) is fundamentally different.

2.5 Multris Protocol Consistency

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

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 in Fig. 4. 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 non-deterministic 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 $(p_0, \dots, p_i, \dots, p_j, \dots, p_{n-1})$ 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 $(p_0, \dots, p_i', \dots, p_j', \dots, p_{n-1})$ after the communication has happened.

Implicit transfer of information. 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:

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

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.

Implicit transfer of resources. We allow accumulating resources from previous interactions. Once a communication between $\mathbf{!}\,[j]\ (\vec{x}_1 : \vec{\tau}_1)\ \langle v_1 \rangle \{P_1\}.\, p_i'$ and $\mathbf{?}[i]\ (\vec{x}_2 : \vec{\tau}_2)\ \langle v_2 \rangle \{P_2\}.\, p_j'$ 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:

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

Here, the resource $\ell \mapsto x$ 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 resource $\ell \mapsto x$ through the exchanges with 1.

Soundness of implicit transfer. The reader may wonder why implicit transfer of resources is sound. To understand this, let us first consider where the resources come from, and where they go. The resources are initially provided by the sender, when the sender sends a message and applies the Wp-send rule. The Wp-send rule requires that the precondition P holds, and the sender must prove this. Second, the Wp-send rule requires a channel ownership assertion $c \rightarrowtail \mathbf{!}\,[i]\ (\vec{x} : \vec{\tau})\ \langle v \rangle \{P\}.\ p$. The channel ownership assertion holds a reference to the channel invariant. The soundness proof for the Wp-send rule places the resources P in the channel invariant, and the channel invariant is updated to reflect the new state of the channel.

Some time later, a receiver receives a message, and applies the Wp-recv rule. The key to the soundness of the Wp-recv rule is that the channel invariant also stores a witness of consistent $\vec{p}$ for the set of protocols $\vec{p}$ that the currently active parties are following. Because of the way consistent $\vec{p}$ works, the resources p can be used to take a simultaneous step in the protocol of the sender and the receiver, and obtain consistent $\vec{p}'$ for the new set of protocols, as well as the resources Q that are returned to the receiver.

In particular, note that P is not necessarily equal to Q (if they were equal, that would be *explicit* transfer of resources). Instead, the consistent $\vec{p}$ witness can keep (part of) P in the channel invariant, and conversely, consistent $\vec{p}$ can give additional resources to a subsequent receiver that were not explicitly sent by the corresponding sender (but were implicitly sent by a previous sender). The proof obligations that have to be satisfied when proving consistent $\vec{p}$ ensure that the resources put in the channel invariant and taken out of the channel invariant are always consistent with the trace of communication that has happened so far, and all possible interleavings of communication that could happen in the future.

Recursive protocols. We allow for consistency proofs of recursive protocols using Löb induction. If during the course of the consistency proof, we loop back to the initial set of protocols, 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* 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 lets us prove consistency of protocols, such as the following recursive variant of the above protocol:

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

Protocol consistency rules. The consistency of protocols is formally defined by the rules in Fig. 5. The key is the second line of the premise for proving dual $\vec{p}\ i\ j$:

$$\forall \vec{x_1} : \vec{\tau_1}.\ P_1 \mathrel{-\!\!*} (\exists \vec{x_2} : \vec{\tau_2}.\ v_1 = v_2 * P_2 * \triangleright\, \text{consistent}\ (\vec{p}[i := p_1][j := p_2]))$$

This rule allows the binders to differ, because we only need to 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 is available to prove the receiver's proposition P_2, and to prove the continuation protocols consistent. The later modality $\triangleright$ is used for Löb induction to prove the consistency of recursive protocols.

The present $\vec{p}\ i$ obligation exists 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.

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

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

Fig. 5. The Multris rules for protocol consistency.

2.6 Protocol Consistency Proof Automation

Proving protocol consistency is a non-trivial part of verifying multiparty programs in Multris. We develop an effective (albeit naïve) brute-force procedure to resolve such consistency assertions. The protocol consistency proof obligation includes separation logic entailment, and is thus inherently undecidable. When the procedure fails on a sub-goal, the user must make some progress manually before re-invoking the procedure on the remainder of the sub-goal. The majority of all protocol consistency proof obligations in this paper were proved automatically, with only a few steps requiring manual reasoning, such as unfolding recursive protocols and applying Löb induction.

The procedure is as follows. For any protocol system $\vec{p}$ the procedure iterates over each pair $i, j \in \vec{p}$. It first checks whether the pair synchronizes; *i.e.,* whether i is a sender to j, and *vice versa* if j is a receiver from i. For every unsynchronized pair, the proof is complete. For every synchronized pair, we proceed as follows:

(1) Introduce binders $\vec{x}_1 : \vec{\tau}_1$ and resources P_1 of the sender.
(2) Instantiate binders $\vec{x}_2 : \vec{\tau}_2$ of receiver with evars, delegating concrete instantiation to term unification in the subsequent steps.
(3) Resolve obligation value equality $v_1 = v_2$ via term unification; yield a sub-goal on failure.
(4) Resolve resource obligation of receiver P_2 via *framing*; yield a sub-goal on failure. Framing tries to resolve the assertion P_2 by searching for a unifiable resource in the proof context $R_0 * \ldots * R_n$ of previously introduced assertions (via Iris's `iFrame` in Coq).
(5) Update protocols of synchronized pairs to their tails p_1 and p_2 and recursively resolve consistency, retaining any resources of P_1 not used to resolve P_2.

The procedure lets us automatically resolve intricate protocol consistency assertions, such as the example for implicit resource transfer in §2.5. First we consider the synchronization between party 0 and 1. The binders of party 0, ℓ and x, are introduced alongside the resource $\ell \mapsto x$. The binder of party 1, v, is instantiated with an evar, and concretely instantiated as ℓ (up to implicit coercion from reference to value) through term unification when resolving $\ell = v$. The resource obligations of party 1 is True and so we retain $\ell \mapsto x$. The next synchronization we need to consider is between party 1 and 2. We instantiate ℓ when resolving the value equality, as before. We resolve the resource obligation $\ell \mapsto x$ via term unification between the introduced x of party 0, and the evar x of party 2. The last synchronization between party 2 and 0 is trivial, as it aligns syntactically.

Despite its simplicity, the automation procedure was sufficient to automatically resolve nearly all of the proof effort associated with protocol consistency assertions presented in this paper. The crux of the automation procedure lies in unification for instantiating binders and framing of resources: while binders and resources may not immediately align syntactically, they often align up to unification after several synchronization steps. Our implementation leverages the Iris Proof Mode [32, 30] embedded in the Coq proof assistant.

3 Multiparty Verification Benchmark: Ring Leader Election

We demonstrate the expressive power of Multris by verifying a variant of Chang and Roberts's [10] ring leader election algorithm (§3.1). We then discuss the limitations of our approach and propose a ring leader election benchmark challenge for the verification community (§3.2).

The key aspects of leader election are *leader uniqueness* and *leader agreement*: only one leader is elected, and all participants agree on the leader. These properties are challenging to verify as they require reasoning about the global state of the system, while each participant only interacts locally. We show that our protocol consistency relation is well suited to verify both properties. We also show that using separation logic, there is an elegant way to encode both properties. For leader uniqueness, we let the local protocols compete for some exclusive resource, and guarantee that only one participant succeeds, as a result of protocol consistency. Similarly, for leader agreement, we guarantee that any subsequent protocol that depends on the locally elected leader is still sound.

3.1 Verifying Chang and Roberts's Ring Leader Election Algorithm

Chang and Robert's [10] ring leader election algorithm assumes that n participants, with unique IDs $id_0 \dots id_{(n-1)}$, are arranged in a ring. Every participant i receives messages from counter-clockwise participant $(i - 1)\%n$, and sends messages to clockwise participant $(i + 1)\%n$. The algorithm deterministically elects the participant with the highest numerical ID $\mathtt{max}(id_0, \dots, id_{(n-1)})$ as the leader. We present a simplified version of the algorithm that allows at most one ongoing election. The version supporting multiple concurrent elections is left for future work as part of our proposed benchmark (§3.2).

The simplified algorithm is as follows. There are two types of messages: election(k) messages and elected(k) messages. Election(k) messages received by participant i are compared to its ID and:

(1.1) If $k > id_i$, send election(k), else
(1.2) If $k = id_i$, we are elected, send elected(id_i), else
(1.3) If $k < id_i$, send election(id_i).

Received elected(k) messages received by participant i are compared to its ID and:

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

An election is initiated when a participant i sends an election(id_i) message clockwise.

To illustrate how the algorithm works, let us consider a concrete example with 4 participants, where each participant's ID corresponds to its position in the ring, *i.e.,* $id_i = i$. Let participant 0 initiate the election by sending the initial election(0) message, which is iteratively increased through case (1.3) until the election(3) message is sent by participant 3. The election(3) message is passed through the ring via case (1.1), and eventually returns to participant 3, resulting in case (1.2), where participant 3 sends the elected(3) message. Similarly, the elected(3) message is passed through the ring via case (2.2), until it returns to participant 3, resulting in case (2.1), concluding the algorithm.

In the rest of this section we certify leader uniqueness and agreement of Chang and Roberts's algorithm. We do this by giving a topology-agnostic implementation and specification for the individual participants, and a proof of two examples using the aforementioned concrete configuration with 4 participants. A key step is to define local Multris protocols for the individual participants (based on the prose algorithm above), which are agnostic to the topology of the ring. By proving that a system consisting of the local protocol for each participant is consistent, we can leverage our adequacy theorem to obtain leader uniqueness and agreement.

Participant processes. We encode election(i) messages as $\mathtt{inl}\ i$ and elected(i) messages as $\mathtt{inr}\ i$. We write i_l and i_r for the left and right neighbors of participant i, respectively. They are defined

as $i_l \triangleq (i+1)\%n$ and $i_r \triangleq (i-1)\%n$, for the given ring of size n. Note that the participants do not need to be aware of the size n of the ring, they just need to know the positions of their neighbors. The leader election process for each participant is implemented as follows:

$$
\begin{array}{l}
\mathsf{process}\ c\ i \triangleq \\
\quad \mathbf{match}\ c[i_r].\mathbf{recv}()\ \mathbf{with} \\
\quad \mid \mathbf{inl}\ i' \Rightarrow \mathbf{if}\ i < i'\ \mathbf{then}\ c[i_l].\mathbf{send}(\mathbf{inl}\ i');\ \mathsf{process}\ c\ i \qquad (1.1) \\
\qquad\qquad \mathbf{else\ if}\ i = i'\ \mathbf{then}\ c[i_l].\mathbf{send}(\mathbf{inr}\ i);\ \mathsf{process}\ c\ i \qquad (1.2) \\
\qquad\qquad \mathbf{else}\ c[i_l].\mathbf{send}(\mathbf{inl}\ i);\ \mathsf{process}\ c\ i \qquad (1.3) \\
\quad \mid \mathbf{inr}\ i' \Rightarrow \mathbf{if}\ i = i'\ \mathbf{then}\ i' \qquad (2.1) \\
\qquad\qquad \mathbf{else}\ c[i_l].\mathbf{send}(\mathbf{inr}\ i');\ i' \qquad (2.2) \\
\quad \mathbf{end}
\end{array}
$$

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

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

Top-level program for leader uniqueness. To verify that ring leader election guarantees leader uniqueness, we use the following top-level program:

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

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 initiate 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.e.,* $i = i'$). Because the leader frees ℓ, the program is safe only if there is at most one leader (avoiding a double free), which is the key property we aim to verify.

Verification of leader uniqueness. We define the local ring leader election protocol for each participant i as well as a local protocol for the participant that initiates the election:

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

The branching protocol structure from § 2.3 allows us to write the ring_prot protocol in 1-to-1 correspondence with the participant process, reflecting each of its cases. The protocol is parametric in the resources P that are up for election, which are given to the elected leader in case (2.1). The protocol is parameterized by a tail p, which in turn is parametric in the elected leader i'. The resource P is given up when the election is initiated (first message in init_prot) and only given to

the leader in the end. This is achieved using our implicit resources transfer technique, which avoids having the participant pass the resource around explicitly in all messages of the ring_prot protocol.

With these protocols we verify that the processes of each participant and the initiator satisfy:

$$\{c \rightarrowtail \mathsf{ring\_prot}\ i\ P\ p\}\ \mathsf{process}\ c\ i\ \{i'.\ c \rightarrowtail (p\ i') * (i = i' \mathrel{-\!\!*} P)\}$$
$$\{c \rightarrowtail (\mathsf{init\_prot}\ i\ P\ p) * P\}\ \mathsf{init}\ c\ i\ \{i'.\ c \rightarrowtail (p\ i') * (i = i' \mathrel{-\!\!*} P)\}$$

The key step of verifying our top-level program ring_ref_prog for the case of 4 participants is proving consistency of the following protocol system:

$$c_0 \rightarrowtail \mathsf{init\_prot}\ 0\ (\ell \mapsto 42)\ (\lambda i'.\ \mathbf{end})$$
$$c_i \rightarrowtail \mathsf{ring\_prot}\ i\ (\ell \mapsto 42)\ (\lambda i'.\ \mathbf{end}) \quad \text{for each } i \in [1, 2, 3]$$

Consistency is established by repeatedly aligning all synchronizations and ultimately observing that only one participant attempts to obtain the resource $\ell \mapsto 42$. The consistency proof was fully automated by our brute-force tactic, except the unfolding of recursive definitions and application of Löb induction. We chose 4 participants as it was the lowest number where some participants do not interact directly (*e.g.,* participant 0 and 2). We remark that our automation scales up beyond this benchmark, having been tested up to 10 participants.

With the above protocols we verify the following specification for the top-level program:

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

This Hoare triple guarantees that the program is safe to execute via our adequacy theorem (Theorem 2.1), thus certifying that the algorithm implementation achieves leader uniqueness.

Top-level program for leader agreement. The top-level program ring_ref_prog certifies leader uniqueness, since at most one participant deallocates the reference. However, it does not certify leader agreement—that all participants agree on the elected process after the election. Suppose participant 0 believes that participant 1 is elected. While this result would violate leader agreement, it would not result in a safety violation as it would not result in a double free.

To certify leader agreement, we allocate an additional binary channel, which we use as a coordinator that receives each elected ID, and asserts that they are all the same. This will rule out the aforementioned faulty election result as the central coordinator would crash when trying to assert that the elected leader (3) is the same as the leader believed by participant 0 (1).

To verify the new top-level program, 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.

The implementation of the top-level program is as follows:

$$\begin{array}{l}
\mathsf{relay_del}\ c\ c'\ i\ i' \triangleq \\
\quad \mathbf{if}\ i = i'\ \mathbf{then}\ c'[0].\mathbf{send}(i'); c[i_l].\mathbf{send}(); c[i_r].\mathbf{recv}() \\
\quad \mathbf{else}\ c[i_r].\mathbf{recv}(); c'[0].\mathbf{send}(i'); c[i_l].\mathbf{send}() \\
\\
\mathsf{ring_del_prog}\ n \triangleq \\
\quad \mathbf{let}\ (c_r, c_s) = \mathbf{new_chan}(2)\ \mathbf{in} \\
\quad \mathbf{fork}\ \{\mathbf{let}\ i' = c_r[1].\mathbf{recv}()\ \mathbf{in}\ \mathbf{while}(\mathbf{true})\{\mathbf{assert}(c_r[1].\mathbf{recv}() = i')\}\} \\
\quad \mathbf{let}\ (c_0, \ldots, c_{n-1}) = \mathbf{new_chan}(n)\ \mathbf{in} \\
\quad \mathbf{for}(i = 1 \ldots (n-1))\ \{\mathbf{fork}\ \{\mathbf{let}\ i' = \mathsf{process}\ c_i\ i\ \mathbf{in}\ \mathsf{relay_del}\ c_i\ c_s\ i\ i'\}\} \\
\quad \mathbf{let}\ i' = \mathsf{init}\ c_0\ 0\ \mathbf{in}\ \mathsf{relay_del}\ c_0\ c_s\ 0\ i'
\end{array}$$

We first define a joined relay/delegate procedure relay_del for relaying the elected ID and delegating the channel endpoint as follows: 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 clockwise.

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

Verification of leader agreement. We start the verification by specifying the relay/delegate procedure using the following protocols:

$$\begin{aligned}
\mathsf{relay_prot}\ i' &\triangleq \mu \mathit{rec}.\, \mathbf{!}\,[0]\ \langle i' \rangle.\ \mathit{rec} \\
\mathsf{init_relay_prot} &\triangleq \mathbf{!}\,[0]\ (i' : \mathbb{N})\ \langle i' \rangle.\ \mathsf{relay_prot}\ i' \\
\mathsf{del_prot}\ i\ i'\ c' &\triangleq \mathbf{if}\ i = i'\ \mathbf{then}\ \mathbf{!}\,[i_l]\ \langle () \rangle \{c' \rightarrowtail \mathsf{relay_prot}\ i'\}.\ \mathbf{?}[i_r]\ \langle () \rangle \{c' \rightarrowtail \mathsf{relay_prot}\ i'\}.\ \mathbf{end} \\
&\quad\ \mathbf{else}\ \mathbf{?}[i_r]\ \langle () \rangle \{c' \rightarrowtail \mathsf{relay_prot}\ i'\}.\ \mathbf{!}\,[i_l]\ \langle () \rangle \{c' \rightarrowtail \mathsf{relay_prot}\ i'\}.\ \mathbf{end}
\end{aligned}$$

The init_relay_prot protocol captures that the leader first determines the elected ID i', after which point everyone must relay the same ID via the recursively defined relay_prot protocol. 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.

Next, we prove the following specification of the relay/delegate procedure:

$$\left\{c \rightarrowtail (\mathsf{del\_prot}\ i\ i'\ c') * (i = i' \mathrel{-\!\!*} c' \rightarrowtail \mathsf{init\_relay\_prot})\right\}\ \mathsf{relay\_del}\ c\ c'\ i\ i'\ \{\mathsf{True}\}$$

The precondition says that if the participant is the leader, it should own the c' channel endpoint with the initial relay protocol, which it has obtained as the result of the election. The channel c' is used to first relay the elected ID, after which point it sends it clockwise around the ring.

The key to the verification of the relay/delegate procedure is proving consistency of the following two protocol systems:

$$\frac{\begin{array}{l} c_r \rightarrowtail \mathbf{?}[0]\ (i' : \mathbb{N})\ \langle i' \rangle.\ \mu \mathit{rec}.\, \mathbf{?}[0]\ \langle i' \rangle.\ \mathit{rec} \\ c_s \rightarrowtail \mathsf{init\_relay\_prot} \end{array}}{\begin{array}{l} c_0 \rightarrowtail \mathsf{init\_prot}\ 0\ (c_s \rightarrowtail \mathsf{init\_relay\_prot})\ (\lambda i'.\ \mathsf{del\_prot}\ 0\ i'\ c_s) \\ c_i \rightarrowtail \mathsf{ring\_prot}\ i\ (c_s \rightarrowtail \mathsf{init\_relay\_prot})\ (\lambda i'.\ \mathsf{del\_prot}\ i\ i'\ c_s) \quad \text{for each } i \in [1,2,3] \end{array}}$$

The protocol consistency of the second system relies on the fact that there is 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 fully automated the consistency proofs, except the unfolding of recursive definitions and application of Löb induction.

With the above protocol system we verify the top-level 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 covered. 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 [22, 6] 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 Robert's [10] **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 a single ongoing election. 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.

4 Model and Soundness

In this section, we explain how the rules shown in Fig. 3 are proved sound. We first show how the Multris adequacy theorem is a direct instance of the Iris adequacy theorem (§ 4.1), as our channels are implemented on top of the HeapLang language (§ 4.2). We then present how to define channel endpoint ownership and verify the multiparty channel rules of Multris using a novel logical abstraction called the *Multris ghost theory* (§ 4.3). Finally, we discuss how our multiparty dependent separation protocols and protocol consistency are defined, and how we validated the Multris ghost theory (§ 4.4). Together, these proofs validate the rules shown in Fig. 3.

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, the Multris adequacy theorem Theorem 2.1 follows directly from the identical Iris adequacy theorem:

Theorem 4.1 (Iris Adequacy). *A proof of* $\mathsf{wp}\ e\ \{v.\ \phi(v)\}$ *implies that* e *is* ***safe****, i.e., if* $([e], \emptyset) \to_{\mathrm{t}}^{*} ([e_0 \dots e_n], h)$*, then for each* $i \le n$ *either* e_i *is a value or* (e_i, h) *can step. Furthermore, any returned value* v *of* e *satisfies* $\phi(v)$*.*

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 (denoted $m_{i,j}$) acts as a one-sized buffer over which participant i can send a value to participant j. Channel endpoints are

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

```
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) ()
```

```
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
```

Fig. 6. The Multris channel implementation.

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 an $n \times n$ matrix and returns n participant tuples with IDs ranging from 0 to $(n-1)$.

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 $c \rightarrowtail p$ resource. To do so, we leverage the Iris methodology: constructing a *ghost theory* that effectively yields a separation logic approach to a state transition system, which models the domain problem, and then connect it to the implementation. In particular, we construct the so-called *Multris ghost theory* (whose construction is detailed in §4.4), and use it verify our specifications as detailed in this section.

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: prot_ctx $\chi\ n$ and prot_own $\chi\ i\ p$. The resources are associated with each other using the logical identifier χ, called a ghost name. The 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 system n. The 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* $\Rightarrow\!\!\!\!\Rightarrow$, which can be applied during program verification, as made precise by the following associated rule:

$$\textsc{Bupd-wp}\quad \frac{P * R \mathrel{-\!\!*} \mathsf{wp}\ e\ \{v.\ Q\}}{(\Rightarrow\!\!\!\!\Rightarrow P) * R \mathrel{-\!\!*} \mathsf{wp}\ e\ \{v.\ Q\}}$$

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

$$\frac{\textsc{consistent}\ \vec{p}}{\models\!\!\Rightarrow \exists \chi.\ \mathsf{prot\_ctx}\ \chi\ |\vec{p}| * \ast_{i \mapsto p \in \vec{p}}\ \mathsf{prot\_own}\ \chi\ i\ p}*$$

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

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

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.

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 $P_1[\vec{x}_1 := \vec{t}_1]$.

The ghost update PROTO-STEP yields an instantiation of the binders of the receiving protocol $(\vec{t}_2 : \vec{\tau}_2)$, evidence that the protocol values are equal $v_1[\vec{x}_1 := \vec{t}_1] = v_2[\vec{x}_2 := \vec{t}_2]$, and the resources of the receiving protocol $P_2[\vec{x}_2 := \vec{t}_2]$. The update preserves the central coordinator $\mathsf{prot\_ctx}\ \chi\ n$, and updates the protocols to their respective tails $\mathsf{prot\_own}\ \chi\ i\ (p_1[\vec{x}_1 := \vec{t}_1])$ and $\mathsf{prot\_own}\ \chi\ j\ (p_2[\vec{x}_2 := \vec{t}_2])$; the sending protocol continues with the given term instantiation $\vec{t}_1$, while the receiving protocol continues with the returned instantiation of its binders $\vec{t}_2$.

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

Finally, the rule PROTO-LE captures that $\mathsf{prot\_own}\ \chi\ i\ p$ is closed under the subprotocol relation.

In the following section we demonstrate how the Multris ghost theory is 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 $c \rightarrowtail p$, which 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

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

Fig. 8. The channel endpoint ownership definition.

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 define the channel endpoint ownership as shown 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 $\mathsf{prot\_ctx}\ \chi\ n$.
(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 $\mathsf{prot\_own}\ \chi\ i\ p$.

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 $\mathbf{Some}(v)$, 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 ($\mathsf{prot\_own}\ \chi\ i\ (!\,[j]\ \vec{x} : \vec{\tau}\ \langle v \rangle \{P\}.\ p)$). 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 $\boxed{\bullet_E\ (\text{next } p)}^{\gamma E}$ and $\boxed{\circ_E\ (\text{next } p)}^{\gamma E}$. 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 [17, §9.1]. We simply add a participant identifier to the message constructor, to determine the participant with which we communicate:

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

Protocols are either terminating ($\texttt{inl}()$), or exchange a message ($\texttt{inr}$ ...). Message exchanges are defined as predicates over the exchanged values and the exchanged tails, to facilitate the dependent binders. To justify the negative recursive occurence of iProto we make use of Iris's support for solving recursive domain equations [2, 7] based on step-indexing [4, 1, 5]. Crucial is that the recursive occurrence of iProto is guarded with the type-level later ($\blacktriangleright$), which is constructed via next at the term level. We refer the interested reader to Hinrichsen et al. [17, §9.1].

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

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

This protocol consistency relation captures the consistency rules shown in Fig. 5 by definition.

We remark that this definition is syntactically similar to the intensional definition of synchronous multiparty session type consistency, given by Scalas and Yoshida [38]. Even so, it is semantically

distinct, due to the dependent binders binding into subsequent protocols, and from the embedding in separation logic which enables implicit transfer resources.

Subprotocols. The subprotocol definition of the multiparty dependent separation protocols is much similar to the binary one presented in Hinrichsen et al. [17, §9.2]. However, since our setting is synchronous, we do not have the rule pertaining to swapping (⊑-SWAP), where receiving protocols may be related to sending ones [35]. Even so, we inherit all the remaining rules (used to validate SUB-SEND and SUB-RECV). Similar to Actris, this allow us to carry out proofs of programs such as the one pertaining to protocol compositionality, presented in Hinrichsen et al. [17, §6.3].

Validating the Multris ghost theory. With the definition of protocol consistency in hand, we define the resources of the Multris ghost theory akin to the original binary Actris ghost theory in Hinrichsen et al. [17, §9.4]. We let the protocol context resource prot_ctx $\chi\ n$ govern protocol consistency of all protocols, and let the fragments prot_own $\chi\ i\ p$ reflect the current state of the protocols, up to subprotocols. The formal definitions are:

$$\text{prot\_ctx}\ \chi\ n \triangleq \exists \vec{p}.\ |\vec{p}| = n * \boxed{\bullet \vec{p}}^{\chi} * \triangleright\, \text{CONSISTENT}\ \vec{p}$$
$$\text{prot\_own}\ \chi\ i\ p \triangleq \exists p'.\ \boxed{\circ (i, p)}^{\chi} * \triangleright (p' \sqsubseteq p)$$

These resources, along with the definition of protocol consistency, are 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.

Soundness of implicit resource transfer via implicit ownership in separation logic. Resources flowing in and out of the protocol consistency assertion and system context resource prot_ctx $\chi\ n$ facilitates the implicit resource transfer as covered in §2.5, but the mechanism may not be immediately obvious. The crux to this is how separation logic assertions can implicitly capture ownership. Consider the following trivially true separation implication:

$$P \mathrel{-\!\!*} Q \mathrel{-\!\!*} (P * Q)$$

Suppose we have the resources P, and provide them to resolve the first requirement of the assertion, resulting in:

$$Q \mathrel{-\!\!*} (P * Q)$$

While true, the assertion might look odd at first glance; by providing *just* Q, we obtain *both* P *and* Q. This is due to the fact that the assertion implicitly assert ownership of P, on the condition that Q is supplied. Conceptually, this is similar to partial function application resulting in function closures.

This pattern is reflected in the protocol consistency assertion. Consider the protocols of the implicit resource transfer example from §2.5:

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

Unfolding the protocol consistency definition for this specific protocol system, we have something alike the following nesting of separation implications:

$$\begin{array}{ll} \forall(\ell : \mathsf{Loc}), (x : \mathbb{Z}).\ \ell \mapsto x \mathrel{-\!\!*} & \text{(* Send of } 0 \to 1 \text{ *)} \\ \exists(v : \mathsf{Val}).\ \ell = v * \mathsf{True} * & \text{(* Receive of } 0 \to 1 \text{ *)} \\ \mathsf{True} \mathrel{-\!\!*} & \text{(* Send of } 1 \to 2 \text{ *)} \\ \exists(\ell : \mathsf{Loc}).\ v = \ell * \ell \mapsto x * & \text{(* Receive of } 1 \to 2 \text{ *)} \\ \ldots & \end{array}$$

Similar to the initial separation logic assertion above, we can prove that this assertion is true, which is exactly the CONSISTENT $\vec{p}$ side goal of allocating the Multris ghost state (and consequentially allocating a new channel in the program logic with the WP-NEW rule).

We now explain how the above assertion develops during program execution. We first resolve sender of the $0 \to 1$ synchronization, by supplying an instantiation of the universally quantified variables, and using the resources provided by the sender (the premise of the WP-SEND rule) to resolve the $\ell \mapsto x$ resource obligation of the first separation implication, yielding:

$$\begin{array}{ll} \exists(v : \mathsf{Val}).\ \ell = v * \mathsf{True} * & \text{(* Receive of } 0 \to 1 \text{ *)} \\ \mathsf{True} \mathrel{-\!\!*} & \text{(* Send of } 1 \to 2 \text{ *)} \\ \exists(\ell : \mathsf{Loc}).\ v = \ell * \ell \mapsto x * & \text{(* Receive of } 1 \to 2 \text{ *)} \\ \ldots & \end{array}$$

We then resolve the receiving side of the $0 \to 1$ synchronization, by obtaining the existentially quantified variables of the receiver, along with the term unifications, and the (empty) resources True, yielding:

$$\begin{array}{ll} \mathsf{True} \mathrel{-\!\!*} & \text{(* Send of } 1 \to 2 \text{ *)} \\ \exists(\ell : \mathsf{Loc}).\ v = \ell * \ell \mapsto x * & \text{(* Receive at } 1 \to 2 \text{ *)} \\ \ldots & \end{array}$$

At this point the protocol consistency (for the updated protocol system), *implicitly* owns $\ell \mapsto x$, similar to how the generic example above implicitly owned P. When resolving the second synchronization $1 \to 2$, the sender does not need to supply any resources (as per the True resource obligation), yet the receiver still obtains the implicitly owned $\ell \mapsto x$.

Finally, note that the protocol context resource prot_ctx χ n does not explicitly exhibit the state of the protocols as a parameter. Instead, we use the ghost state agreement between the local protocols (owned by the individual participants) and the underlying protocol system ghost state $\boxed{\bullet \vec{p}}^{\chi}$, to determine the state of the protocol consistency CONSISTENT $\vec{p}$ that resides inside prot_ctx χ n.

More precisely, when proving the PROTO-STEP rule, we unfold the protocol context resource prot_ctx χ n, to obtain CONSISTENT $\vec{p}$, for some $\vec{p}$. We unify the protocols of the sender and receiver with the corresponding protocols in $\vec{p}$, via the associated ghost state. We then unfold the protocol consistency definition, to obtain the underlying separation implication between the sender and receiver. We resolve the premise of the implication with the binder instantiations and resources of the sender, and obtain the binder instantiations and resources of the receiver (as illustrated above). We finally update the ghost state to reflect the new state of $\vec{p}$. With the updated state, we can close all of the definitions, ultimately hiding the underlying $\vec{p}$ of the protocol context resource prot_ctx χ n, which now implicitly owns any resources that were sent but not received.

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 [18].

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	160
Proofmode tactics	§5	410
Examples:		
• Basic examples from this paper	§2	292
• Two-Buyer Example	[22]	135
• Three-Buyer Example	[6]	200
• Ring Leader Election	§3	356
Total		3609

Table 1. Overview of the Multris Coq mechanization.

While the structure of our mechanization was significantly influenced by (binary) Actris, our work required significant new mechanization effort: defining our novel notion of multiparty protocol consistency and proving the associated lemmas, verifying the new (synchronous) multiparty ghost state, and verifying the multiparty channel specifications. In Table 1 we give an overview of the lines of Coq code associated with each of the contributions of the paper.

In addition to validating the soundness of Multris, our Coq mechanization comes with tactic support à la Actris for symbolic execution of the multiparty channel primitives as well as for protocol consistency. This effectively means that the Multris Coq mechanization can be used to foundationally verify multiparty programs.

The symbolic execution of the channel primitives largely follows the binary case, and the addition of a correspondent participant can be resolved automatically. A more novel proof effort for a user instead arise from the CONSISTENT $\vec{p}$ obligation of WP-NEW. To resolve such obligations, we provide a tactic implementation of the brute-force procedure presented in §2.6. In particular, we leverage the existing infrastructure of Coq and the Iris Proof Mode for resolving term unification and resource framing, respectively.

6 Related and Future Work

Session types. Session types were developed by Honda et al. [20, 21] 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. [22]. Traditionally, multiparty session types are *top-down*—they rely on a global type that describes all interactions of the system, 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 [38]. Therefore, Scalas and Yoshida [38] 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 this bottom-up method to consistency. The key difference between Scalas and Yoshida [38] and our work is that Scalas and Yoshida [38] develop type systems, while we develop a program logic. This results in several key differences. First, our protocols carry separation logic resources. Second, our protocols are *dependent*, i.e., have binders on which the message, resources, and tail of the protocol

can depend. Third, the soundness proof and model of our logic (*i.e.,* §4: the proofs of the WP-rules and the definition of channel endpoint ownership) has no counterpart in Scalas and Yoshida [38] (the closest analogue would be the type soundness proof, which is quite different). Lastly, our logic is embedded in Iris and proven sound in Coq. 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 [9, 24, 44]. Multris is also in this vein, but instead of mechanization of the meta theory of multiparty type systems, we focus on the verification of the soundness of a program logic.

Design-by-contract and refinement types for multiparty message passing. Based on multiparty session types, Bocchi et al. [8] 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 above), (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. [8] 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. [47] 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 [41] 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. [47] is similar to the ways in which Multris differs from Bocchi et al. [8]. A key similarity is that Zhou et al. [47] 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. [47]) 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" as 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. [47] 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 the development of Multris, this was the most significant effort, see §4). In addition to functional correctness, the verification system of Zhou et al. [47] guarantees deadlock freedom.

Program logics for message passing. Program logics based on concurrent separation logic commonly employ protocol mechanisms, often in the form of state transition systems [12, 46, 40] or the combination of invariants and ghost state [29]. Since these mechanisms are very general, and allow for verification of all kinds of different concurrent data structures, one could use them to reason about message-passing programs directly. In comparison, program logics with a protocol mechanism based on session types are more domain-specific and provide a higher level of abstraction.

The program logic for message passing that is most closely related to our work is Actris [16, 17], which also uses separation logic to reason about message-passing programs via dependent protocols. The key distinction between Actris and Multris is that the former focuses on two-party/binary message-passing, while we focus on multiparty message-passing. As a result, the Wp-send and

WP-RECV proof rules of Multris are quite similar to those of Actris while handling multiparty communication, which we consider a strength of our work. The WP-NEW rule of Multris is quite different, as it requires CONSISTENT $\vec{p}$ to hold, which is absent and unnecessary in Actris, which only considers two-party communication. The implementation of our multiparty channels as well as the Multris soundness proof are also significantly different from those of Actris.

Before we arrived at the current design of Multris, we explored several variations that we did not know how to make work. For instance, the combination of global types with dependent protocols and separation logic results in non-trivial difficulties for soundness, and our proof automation crucially relies on our channels being synchronous. We leave these challenges for future work.

Jacobs et al. [25] 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. [25] 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.

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 [38] 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. [33] 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 Multris would also allow us to consider asynchronous subprotocols from binary Actris [17], which allow swapping of sends over receives as first introduced by Mostrous et al. [35]. It would moreover open the door to generalize the efficient low-level implementation of asynchronous channels with linking by Somers and Krebbers [39] to the multiparty case. Going beyond asynchrony, we would like to consider distributed systems, with processes running on different machines and communicating over a network, such as in Gondelman et al. [15].

Another direction is to restrict the logic to ensure a stronger property, such as deadlock freedom à la Jacobs et al. [26]. Finally, we would like to apply the method of logical type soundness in Iris [43] (which Hinrichsen et al. [19] 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. [24].

Data-Availability Statement

The Coq development for this paper can be found in [18].

Acknowledgments

We thank Jesper Bengtson for valuable discussions and initial work towards a candidate definition for multiparty protocol consistency. We also thank the anonymous reviewers for their feedback and suggestions. This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.

References

[1] Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph. D. Dissertation. Princeton University.

[2] Pierre America and Jan J. M. M. Rutten. 1989. Solving Reflexive Domain Equations in a Category of Complete Metric Spaces. *JCSS* (1989). https://doi.org/10.1016/0022-0000(89)90027-5

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

[4] Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *TOPLAS* (2001). https://doi.org/10.1145/504709.504712

[5] 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

[6] 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

[7] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. 2010. The category-theoretic solution of recursive metric-space equations. *TCS* (2010). https://doi.org/10.1016/j.tcs.2010.07.010

[8] 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

[9] 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

[10] 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

[11] 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

[12] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP*. https://doi.org/10.1007/978-3-642-14107-2_24

[13] 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

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

[15] 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

[16] 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

[17] 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

[18] Jonas Kastberg Hinrichsen, Jules Jacobs, and Robbert Krebbers. 2024. Coq Mechanization of "Multris: Functional Verification of Multiparty Message Passing in Separation Logic". https://doi.org/10.5281/zenodo.13380561

[19] 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

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

[21] 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

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

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

[24] 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

[25] 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

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

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

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

[33] Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. 2023. Complete Multiparty Session Type Projection with Automata. In *CAV 2023*. https://doi.org/10.1007/978-3-031-37709-9_17

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

[35] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *ESOP*. https://doi.org/10.1007/978-3-642-00590-9_23

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

[37] 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

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

[39] Thomas Somers and Robbert Krebbers. 2024. Verified Lock-Free Session Channels with Linking. OOPSLA (2024). https://doi.org/10.1145/3689732

[40] Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *ESOP*. https://doi.org/10.1007/978-3-642-54833-8_9

[41] 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*. https://doi.org/10.1145/2837614.2837655

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

[43] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. To appear in JACM.

[44] Dawit Legesse Tirore, Jesper Bengtson, and Marco Carbone. 2023. A Sound and Complete Projection for Global Types. In *ITP (LIPIcs)*. https://doi.org/10.4230/LIPICS.ITP.2023.28

[45] 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

[46] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. https://doi.org/10.1145/2500365.2500600

[47] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically verified refinements for multiparty protocols. OOPSLA (2020). https://doi.org/10.1145/3428216

Received 2024-04-06; accepted 2024-08-18