Automated Modular Verification for Relaxed Communication Protocols

Andreea Costea[1], Wei-Ngan Chin[1], Shengchao Qin[2,3], and Florin Craciun[4]

[1] School of Computing, National University of Singapore
{andreeac,chinwn}@comp.nus.edu.sg
[2] School of Computing, Teesside University s.qin@tees.ac.uk
[3] Shenzhen University
[4] Faculty of Mathematics and Computer Science, Babes-Bolyai University
craciunf@cs.ubbcluj.ro

Abstract. Ensuring software correctness and safety for communication-centric programs is important but challenging. In this paper we introduce a solution for writing communication protocols, for checking protocol conformance and for verifying implementation safety. This work draws on ideas from both multiparty session types, which provide a concise way to express communication protocols, as well as from separation-style logics for shared-memory concurrency, which provide strong safety guarantees for resource sharing. On the one hand, our proposal improves the expressiveness and precision of session types, without sacrificing their conciseness. On the other hand, it increases the applicability of software verification as well as its precision, by making it protocol aware. We also show how to perform the verification of such programs in a modular and automatic fashion.

1 Introduction

Asynchronous distributed systems are ubiquitous in digital applications, yet achieving their safe design and implementation is notoriously hard. The difficulties in building such systems are many-fold. First, these systems are normally described in the designing phase using communication protocols. The problem at this phase is that, because of the lack of formal, yet easy-to-use specification languages for communication protocols, designers prefer to draft the communication using RFC documents. But these drafts lack mathematical rigorosity, and, therefore, lead to ambiguous interpretations of the communication. Secondly, a developer often validates a system's correct implementation via testing. However, in the case of distributed systems, where reproductibility of execution is challenging, testing is rarely exhaustive. This kind of behavior commonly harbors difficult-to-detect bugs. Thirdly, the safe coordination of independent entities interacting with each other is problematic: on the one hand, the developer must ensure exclusive access to shared resources in the case of tightly coupled entities, and, on the other hand, it must offer safe communication guarantees for the loosely coupled ones. Lastly, it is often the case that the code refactoring of one of the communicating entities requires the re-validation of the entire system. Since validation might be expensive, or difficult to achieve if the source code of certain components is not available, it is desirable for the developer to be able to only validate her changes locally, rather than at the global level.

Over the last decades, behavioral types [35,26] have been studied as specifications of the interactions in communicating systems. In particular, multiparty session types [23], or MPSTs, provide a user-friendly syntax for writing choreographic specifications of distributed systems, and a lightweight mechanism for enforcing communication safety. Communication is considered correct when the system's constituent processes are statically type-checked against the end-point projections of the MPST. This formalism and its numerous extensions are attractive in checking if the implementation follows the intended communication pattern, but it lacks the strong safety and correctness guarantees normally provided by the resource-aware verification systems. Specifically, the MPST approach checks if a transmission's exchanged type is the expected one. However, in their most common form, MPSTs are unable to assert something about the message's numerical properties, and even less so about its carried resources in the case of tightly coupled systems. All these, while numerical properties and resource sharing constitute the pièce de résistance for separation logic [38], a logic for reasoning about resource sharing. In this work, we attach a communication logic in the user-friendly style of MPST, to a separation logic for program verification. Even though we draw on ideas from MPST, the proposed logic differs from MPST in a number of features which yield a more expressive communication specification - without compromising its friendly syntax. The current proposal ultimately leads to stronger guarantees w.r.t. the safety and correctness of distributed system. We shall next highlight these differences.

Writing Multiparty Communication Protocols. The language we propose for writing communication protocols is described in Fig. 1a. Similar to MPST, the language contains the terminal notation $\mathtt{S}{\rightarrow}\mathtt{R} : \mathtt{c}\langle \mathtt{v}{\cdot}\Delta\rangle$ to describe a transmission from sender $\mathtt{S}$ to receiver $\mathtt{R}$, over channel $\mathtt{c}$. Different from type approaches where a message abstracts a type, the exchanged message $\mathtt{v}$ is expressed in the logical form Δ (defined in Fig. 1b). Do note that $\mathtt{v}{\cdot}\Delta$ is in fact a shorthand for the lambda function $(\lambda \mathtt{v}\,.\,\Delta)$. This language uses $\mathtt{G}_1 * \mathtt{G}_2$ for the concurrency of global protocols $\mathtt{G}_1$ and $\mathtt{G}_2$, and $\mathtt{G}_1 \vee \mathtt{G}_2$ for disjunctive choice between either $\mathtt{G}_1$ or $\mathtt{G}_2$, and finally $\mathtt{G}_1\,;\mathtt{G}_2$ on the implicit sequentialization of $\mathtt{G}_1$ before $\mathtt{G}_2$ for either the same party or the same channel. Let us next consider a series of examples to introduce this language and to highlight the benefits over MPST.

Example 1: We consider a cloud service for video editing, where a client sends to the cloud a file of some video format, and expects back an enhanced version of the original file, see Fig. 2a. A client-server protocol to describe this simple interaction is written as follows:

$$\mathtt{CS_a} \triangleq \mathtt{C}{\rightarrow}\mathtt{S} : \mathtt{c}\langle \mathtt{v}{\cdot}\mathtt{v} : \mathtt{file}\rangle\,;\, \mathtt{S}{\rightarrow}\mathtt{C} : \mathtt{c}\langle \mathtt{v}{\cdot}\mathtt{v} : \mathtt{file}\rangle.$$

The $\mathtt{CS_a}$ lightweight protocol suffices to describe the order of communication and the exchanged message type. A rigorous specification though, also emphasizes that the server applies some filter on the original file:

Global protocol	$\mathtt{G}$ ::=	
Single transmission		$\mathtt{S}{\rightarrow}\mathtt{R} : \mathtt{c}\langle \mathtt{v} \cdot \Delta\rangle$
Concurrency		$\mid \mathtt{G} * \mathtt{G}$
Choice		$\mid \mathtt{G} \vee \mathtt{G}$
Sequencing		$\mid \mathtt{G} \,;\, \mathtt{G}$
Exists		$\mid \exists \mathtt{P}^*, \mathtt{c}^*, \mathtt{v}^* \cdot G$
Protocol Instance		$\mid \mathtt{H}(\mathtt{P}^*, \mathtt{c}^*, \mathtt{v}^*)$
Inaction		$\mid \mathtt{emp}$
(Parties)	$\mathtt{S}, \mathtt{R}, \mathtt{P} \in \mathcal{R}\mathtt{ole}$	
(Channels)	$\mathtt{c} \in \mathcal{C}\mathtt{han}$	
(Messages)	$\mathtt{v} \cdot \Delta$	
	v: first-order variable	

(a) Global Protocol Specification

Formula	$\Phi ::= \bigvee \Delta \qquad \Delta ::= \exists\ \mathtt{v}^* \cdot \kappa \wedge \pi$	
Separation	$\kappa ::= \mathtt{emp} \mid \mathtt{v} \mapsto \mathtt{d}(\mathtt{v}^*) \mid p(\mathtt{v}^*)$	
	$\mid \mathcal{C}(\mathtt{c}, \mathtt{P}, \mathtt{L}) \mid \kappa * \kappa \mid \mathtt{V}$	
Pure	$\pi ::= \mathtt{v} : \mathtt{t} \mid b \mid a \mid \pi \wedge \pi \mid \pi \vee \pi$	
	$\mid \neg \pi \mid \exists v \cdot \pi \mid \forall v \cdot \pi$	
Boolean	$b ::= \mathtt{true} \mid \mathtt{false} \mid b = b$	
Ineq.	$a ::= \mathtt{s} = \mathtt{s} \mid \mathtt{s} \leq \mathtt{s}$	
Presbg.	$s ::= \mathtt{k}^{\mathtt{int}} \mid \mathtt{v} \mid \mathtt{s} + \mathtt{s} \mid -\mathtt{s}$	
where	$\mathtt{k}^{\mathtt{int}}$: integer constant;	
	$\mathtt{d}$: data structure; $\mathtt{t}$: type;	
	$\mathtt{V}$: second-order variable;	
	$\mathtt{L}$: local protocol (Fig. 6)	

(b) The (Program) Specification Language

Fig. 1: Mercurius

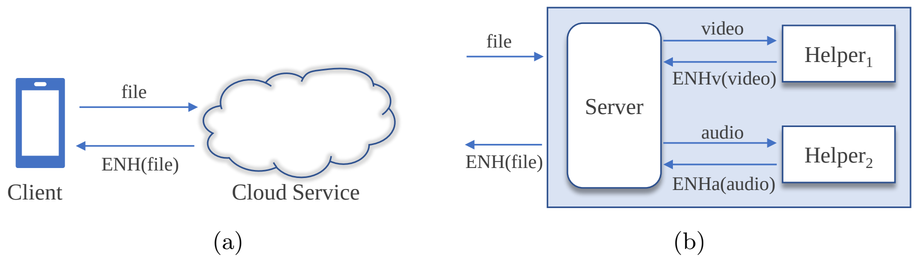

Fig. 2: A Multimedia Cloud Service: A Client requests the Server for a video file enhancement (a). The Server engages two helpers to process the desired enhancement (b)

$$\mathtt{CS_b} \triangleq \exists \mathtt{fd} : \mathtt{file} \cdot (\mathtt{C}{\rightarrow}\mathtt{S} : \mathtt{c}\langle \mathtt{fd}\rangle \,;\, \mathtt{S}{\rightarrow}\mathtt{C} : \mathtt{c}\langle \mathtt{ENH}(\mathtt{fd})\rangle).$$

where $\mathtt{C}{\rightarrow}\mathtt{S} : \mathtt{c}\langle\mathtt{fd}\rangle$ is a prettyprint for $\mathtt{C}{\rightarrow}\mathtt{S} : \mathtt{c}\langle \mathtt{v}{\cdot}\mathtt{v} : \mathtt{file} \wedge \mathtt{v}{=}\mathtt{fd}\rangle$, and $\mathtt{ENH}(\mathtt{fd})$ is a logical predicate describing the enhanced file, e.g. such as applying a brightness, or a slow motion filter. For the simplicity of this explanation we do not define ENH, keeping it abstract, however a user can always attach a definition to ENH to reflect the changes over the file referenced by `fd`. To note that this protocol not only highlights that the server returns an enhanced file, but that it actually returns the enhancement of the original file since both transmissions reference the same file via `fd`.

Moreover, the protocol could also be instrumented to capture the server's enhancement action:

$$\mathtt{CS_c} \triangleq \exists \mathtt{fd} : \mathtt{file} \cdot (\mathtt{C}{\rightarrow}\mathtt{S} : \mathtt{c}\langle \mathtt{fd}\rangle \,;\, \mathtt{FILTER} \,;\, \mathtt{S}{\rightarrow}\mathtt{C} : \mathtt{c}\langle \mathtt{ENH}(\mathtt{fd})\rangle).$$

If the server were to delegate its task to some helper processes - Fig. 2b, where, for example, one processes the video and one handles the audio component of the original media file, the enhancement protocol could be defined as follows:

$$\begin{aligned}\texttt{FILTER} \triangleq \exists \mathtt{H}_1, \mathtt{H}_2, \mathtt{c}_1, \mathtt{c}_2 \cdot\ & (\mathtt{S}{\rightarrow}\mathtt{H}_1 : \mathtt{c}_1\langle \texttt{fd.vid}\rangle \,;\, \mathtt{H}_1{\rightarrow}\mathtt{S} : \mathtt{c}_1\langle \texttt{ENHv}(\texttt{fd.vid})\rangle) \ast \\ & (\mathtt{S}{\rightarrow}\mathtt{H}_2 : \mathtt{c}_2\langle \texttt{fd.aud}\rangle \,;\, \mathtt{H}_2{\rightarrow}\mathtt{S} : \mathtt{c}_2\langle \texttt{ENHa}(\texttt{fd.aud})\rangle).\end{aligned}$$

where `ENHv` and `ENHa` describe some video and audio effects, respectively. The $\ast$ operator which denotes concurrent interactions, intentionally resembles the separating conjunction of separation logic to express a clear separation of communication. In this context, all of the following four possible C-like implementations of the server faithfully follow the `FILTER` protocol:

(i)
```
send(c1,fd.vid);
send(c2,fd.aud);
fd.vid = receive(c1);
fd.aud = receive(c2);
```

(ii)
```
send(c1,fd.vid);
fd.vid = receive(c1);
send(c2,fd.aud);
fd.aud = receive(c2);
```

(iii)
```
(send(c1,fd.vid); fd.vid = receive(c1);)
                    ||
(send(c2,fd.aud); fd.aud = receive(c2);)
```

(iv)
```
    (send(c1,fd.vid) || send(c2,fd.aud);)
                     ;
(fd.vid = receive(c1) || fd.aud = receive(c2);)
```

and the combinations could continue with the sequential permutations between sends or receives in *(i)* and *(iv)*, or the parallelzation of selected pairs of interactions in *(i)* and *(ii)* as long as the sending on a certain channel precedes the local receive on the same channel. To the best of our knowledge, the current state of the art in formalizing communication protocols does not allow such permissive protocols, where the *relaxed order of transmissions* is explicitly captured by the communication protocol. We stress on the fact that the relaxed order of transmissions is not restricted to just inter-party parallel composition specific to MPST, but also comprises the intra-party parallel composition as exemplified above. There is an attempt to tackle the arbitrary order of transmissions in MPST [10], but instead of writing a relaxed protocol, the authors engage a swap relation to check whether the interleaving of transmissions should be allowed at the implementation level. The approach of [10] only checks against cases *(i)* and *(ii)* though, and fail to recognize *(iii)* and *(iv)* as correct implementations of the server described by the `FILTER` protocol. Any other MPST extension would require four different global types to capture the four different kinds of implementation exemplified earlier.

Another subtle point of this example is the *careful usage of the resources,* the file in this case. The file pointed by `fd` is split into its two components, `fd.vid` and `fd.aud`, respectively, and exclusively shared between helpers $\mathtt{H}_1$ and $\mathtt{H}_2$. The server gains back the ownership of the two components only after the two helpers have finished their job and return the resources back to the server. Any attempt to access a resource before the helper returns its ownership to the server is regarded as unsafe in our approach. To the best of our knowledge, this is the first such approach where the safe resource usage is captured in a lightweight, yet expressive multiparty protocol even in the case of hybrid communication, with both loosely ($\mathtt{C}$ and $\mathtt{S}$) and tightly ($\mathtt{S}$, $\mathtt{H}_1$ and $\mathtt{H}_2$) coupled communicating entities. Better yet, the communication protocol does not need to distinguish between the loosely and the tightly coupled scenarios, consigning the choice of the coupling degree to the developer.

Example 2: In the cloud service examples we described the interaction between exactly one client and a cloud server. Clearly this is too restrictive, since a server should be allowed to serve multiple clients. To support a dynamic number of participants, we describe the protocol using recursive parameterized protocols:

$$\begin{aligned}
&\mathtt{CLOUD}(\mathtt{S},\mathtt{c}) \triangleq \exists \mathtt{C}, \mathtt{c}' \cdot \mathtt{C}{\rightarrow}\mathtt{S} : \mathtt{c}\langle \mathtt{c}' \rangle \,;\, (\mathtt{CS_d}(\mathtt{C},\mathtt{S},\mathtt{c}') * \mathtt{CLOUD}(\mathtt{S},\mathtt{c})).\\
&\mathtt{CS_d}(\mathtt{C},\mathtt{S},\mathtt{c}) \triangleq \exists \mathtt{fd} : \mathtt{file} \cdot (\mathtt{C}{\rightarrow}\mathtt{S} : \mathtt{c}\langle \mathtt{fd} \rangle \,;\, \mathtt{FILTER} \,;\, \mathtt{S}{\rightarrow}\mathtt{C} : \mathtt{c}\langle \mathtt{ENH}(\mathtt{fd}) \rangle).
\end{aligned}$$

where the client-server $\mathtt{CS_d}$ protocol is similar to $\mathtt{CS_c}$, except that it is now parameterized with the communicating entities and corresponding channel. The client first sends the server a private channel $\mathtt{c}'$, which is then used for the communication within the $\mathtt{CS_d}$ protocol. The $*$ between $\mathtt{CS_d}$ and the recursive instance of `CLOUD` servers two purposes. On the one hand, it denotes exclusive resource usage between the two protocols. On the other hand, it permits a *relaxed* implementation of the cloud application, where the server could either (1) serve one client at a time, or it could (2) serve multiple clients concurrently spawning a new process once it receives the private channel of some client.

Contributions and outline. The contributions of this paper are as follows:

- An *expressive session logic* called Mercurius, that is both precise (supports logical message) and concise (in the style of session types) for modelling multi-party protocols. Through its support for relaxed protocols, Mercurius offers wider communication design choices than the current state of the art.
- A deductive verification system which embeds Mercurius for automatically checking protocol conformance and safe implementation. This system copes with both distributed as well as tightly coupled systems.
- A projection mechanism for each communicating party such that each party follows its local specification. This enables modular verification, where each party is verified independently from the other communication participants.
- A projection mechanism for each communication channel w.r.t. a party. The verifier is instructed to manipulate channel specifications, exploiting thus the possibility to delegate the communication to third parties in a natural way, without breaking locality and without the need of additional communication primitives, except for the usual `send`/`receive`, `open`/`close`.

After formalizing the global protocol in Sec. 2, we describe the projection rules in Sec. 3, and then embed the logic into a verification system in Sec. 4.

2 Global Protocols

We now formalize Mercurius, whose syntax is depicted in Fig. 1a. We first describe the communication model, and then list down the elements of the protocol and discuss their properties.

Communication model. To support a wide range of communication interfaces, the current session logic is designed for a permissive communication model, where:

- The transfer of a message dissolves *asynchronously*, that is to say that sending is non-blocking while receiving is blocking.
- The communication interface of choice manipulates *linear FIFO channels* in the style of [3] (i.e. a message is delivered without interference from other participants: the receiver is able to determine who the sender is without any ambiguity).
- For simplicity, the communication assumes unbounded buffers.

Transmission. As described in Sec. 1, a *transmission* $\mathtt{S}{\rightarrow}\mathtt{R} : \mathtt{c}\langle \mathtt{v} \cdot \Delta\rangle$ involves a sender $\mathtt{S}$ and a receiver $\mathtt{R}$ transmitting a message $\mathtt{v}$ expressed in logical form Δ over a buffered channel $\mathtt{c}$. To access the components of a transmission we define the following auxiliary functions: $\mathtt{send}(\mathtt{S}{\rightarrow}\mathtt{R} : \mathtt{c}\langle \mathtt{v} \cdot \Delta\rangle) \overset{\text{def}}{=} \mathtt{S}$, $\mathtt{recv}(\mathtt{S}{\rightarrow}\mathtt{R} : \mathtt{c}\langle \mathtt{v} \cdot \Delta\rangle) \overset{\text{def}}{=} \mathtt{R}$, $\mathtt{chan}(\mathtt{S}{\rightarrow}\mathtt{R} : \mathtt{c}\langle \mathtt{v} \cdot \Delta\rangle) \overset{\text{def}}{=} \mathtt{c}$ and $\mathtt{msg}(\mathtt{S}{\rightarrow}\mathtt{R} : \mathtt{c}\langle \mathtt{v} \cdot \Delta\rangle) \overset{\text{def}}{=} \mathtt{v} \cdot \Delta$. We shall often quantify over the existing transmissions using the literal i. Transmissions are irreflexive, $\mathtt{send}(i) \neq \mathtt{recv}(i)$. We define a function $\mathtt{TR}(\mathtt{G})$ which decomposes a given protocol $\mathtt{G}$ to collect a set of all its constituent transmissions, and a function $\mathtt{TR}^{\mathtt{fst}}(\mathtt{G})$ to return the set of all possible first transmissionsg.

Two messages are said to be *disjoint*, denoted by $\mathtt{v}_1 \cdot \Delta_1 \# \mathtt{v}_2 \cdot \Delta_2$, if $\mathtt{UNSAT}(\Delta_1 \wedge [\mathtt{v}_1/\mathtt{v}_2]\Delta_2)$. We next abuse the set membership symbol, $\in$, to denote the followings (and, correspondingly, $\notin$ to denote their negation):

$(\in_{\text{transm.}})$ $i \in \mathtt{G} \Leftrightarrow i \in \mathtt{TR}(\mathtt{G})$
$(\in_{\text{channel}})$ $\mathtt{c} \in i \Leftrightarrow \mathtt{chan}(i) = \mathtt{c}$
$(\in_{\text{channel}})$ $\mathtt{c} \in \mathtt{G} \Leftrightarrow \exists i \in \mathtt{G} \cdot \mathtt{c} \in i$
$(\in_{\text{party}})$ $\mathtt{P} \in i \Leftrightarrow \mathtt{send}(i) = \mathtt{P}$ or $\mathtt{recv}(i) = \mathtt{P}$
$(\in_{\text{party}})$ $\mathtt{P} \in \mathtt{G} \Leftrightarrow \exists i \in \mathtt{G} \cdot \mathtt{P} \in i$

The parallel composition of global protocols forms a commutative monoid $(\mathtt{G}, *, \mathtt{emp})$ with $\mathtt{emp}$ as identity element, while disjunction and sequence form semigroups, $(\mathtt{G}, \vee)$ and $(\mathtt{G}, ;)$, with the former also satisfying commutativity. $\mathtt{emp}$ acts as the left identity element for sequential composition:

$$(\mathtt{G}_1 \; ; \; \mathtt{G}_2) \; ; \; \mathtt{G}_3 \equiv \mathtt{G}_1 \; ; \; (\mathtt{G}_2 \; ; \; \mathtt{G}_3) \qquad \mathtt{G}_1 * \mathtt{G}_2 \equiv \mathtt{G}_2 * \mathtt{G}_1 \qquad \mathtt{G} * \mathtt{emp} \equiv \mathtt{G}$$
$$(\mathtt{G}_1 * \mathtt{G}_2) * \mathtt{G}_3 \equiv \mathtt{G}_1 * (\mathtt{G}_2 * \mathtt{G}_3) \qquad \mathtt{G}_1 \vee \mathtt{G}_2 \equiv \mathtt{G}_2 \vee \mathtt{G}_1 \qquad \mathtt{emp} \; ; \; \mathtt{G} \equiv \mathtt{G}$$
$$(\mathtt{G}_1 \vee \mathtt{G}_2) \vee \mathtt{G}_3 \equiv \mathtt{G}_1 \vee (\mathtt{G}_2 \vee \mathtt{G}_3)$$

Sequential composition is not commutative, unless it satisfies certain disjointness properties:
$\mathtt{G}_1 \; ; \; \mathtt{G}_2 \equiv \mathtt{G}_2 \; ; \; \mathtt{G}_1$ when $\forall \mathtt{c}_1{\in}\mathtt{G}_1, \mathtt{c}_2{\in}\mathtt{G}_2 \Rightarrow \mathtt{c}_1{\neq}\mathtt{c}_2$ and $\forall \mathtt{P}_1{\in}\mathtt{G}_1, \mathtt{P}_2{\in}\mathtt{G}_2 \Rightarrow \mathtt{P}_1{\neq}\mathtt{P}_2$.

The equivalence of protocols could be reduced to that of graph isomorphism, by interpreting the protocol as a graph whose vertexes are actions (message sending or receiving), and whose directed edges are transmissions from a sending to a receiving action. For lack of space and since these proofs are not of interest for the current work, we treat the above equivalences as axioms.

2.1 Well-Formedness

Concurrency. The $*$ operator offers support for arbitrary-ordered (concurrent) transmissions, where the order of their completion is not important for the final outcome.

Definition 1 (Well-Formed Concurrency) *A protocol specification,* $\mathtt{G}_1 * \mathtt{G}_2$*, is said to be well-formed w.r.t.* $*$ *if and only if* $\forall \mathtt{c}{\in}\mathtt{G}_1 \Rightarrow \mathtt{c}{\notin}\mathtt{G}_2$*, and vice versa.*

This restriction avoids non-determinism of concurrent communications over the same channel.

Choice. The $\vee$ operator is essential for the expressiveness of Mercurius, but its usage must be carefully controlled:

Definition 2 (Well-Formed Choice) *A disjunctive protocol specification,* $\mathtt{G}_1 \vee \mathtt{G}_2$*, is said to be well-formed with respect to* $\vee$ *if and only if all of the following conditions hold, where* $\mathtt{T}_1$ *and* $\mathtt{T}_2$ *account for all first transmissions of* $\mathtt{G}_1$ *and* $\mathtt{G}_2$*, respectively, namely* $\mathtt{T}_1{=}\mathtt{TR}^{\mathtt{fst}}(\mathtt{G}_1)$ *and* $\mathtt{T}_2{=}\mathtt{TR}^{\mathtt{fst}}(\mathtt{G}_2)$*:*

(a) (same first channel) $\forall i_1, i_2 \in \mathtt{T}_1 \cup \mathtt{T}_2 \Rightarrow \mathtt{chan}(i_1) = \mathtt{chan}(i_2)$;
(b) (same first sender) $\forall i_1, i_2 \in \mathtt{T}_1 \cup \mathtt{T}_2 \Rightarrow \mathtt{send}(i_1) = \mathtt{send}(i_2)$;
(c) (same first receiver) $\forall i_1, i_2 \in \mathtt{T}_1 \cup \mathtt{T}_2 \Rightarrow \mathtt{recv}(i_1) = \mathtt{recv}(i_2)$;
(d) (mutually exclusive "first" messages)

$$\forall i_1, i_2 \in \mathtt{T}_1 \cup \mathtt{T}_2 \Rightarrow \mathtt{msg}(i_1) \# \mathtt{msg}(i_2) \vee i_1{=}i_2;$$

(e) (same pattern) Except for the parties in $\mathtt{T}_1$ *and* $\mathtt{T}_2$*, the rest of the participants must have a uniform local view of the communications across all disjuncts (to avoid informing all the participants of the choice being made).*
(f) (recursive well-formedness) $\mathtt{G}_1$ *and* $\mathtt{G}_2$ *are well-formed with respect to* $\vee$*.*

Definition 3 (Well-Formed Protocol) *A protocol* $\mathtt{G}$ *is said to be well-formed, if and only if* $\mathtt{G}$ *contains only well-formed concurrent transmissions, and well-formed choices.*

To ensure the correctness of our approach, Mercurius disregards as unsound any usage of $*$ or $\vee$ which is not well-formed.

3 Local Projection

Based on the communication interface, but also on the verifier's requirements, the projection of the global protocol to local specifications goes through a couple of automatic projection phases before being used by the verification process. This way, the projections of phase one (we call them *per-party projections*) describe how each party is contributing to the communication. More granularly, the projections in the second phase (called *per-channel projections*) describe how each communication channel is used by their respective communicating parties.
Projection Overview and Protocol Refinement.
Example 3: Consider the following protocol between some parties $\mathtt{C}_1$, $\mathtt{C}_2$ and $\mathtt{P}$, communicating via channels $\mathtt{c}_1$ and $\mathtt{c}_2$:

$$\mathtt{G} \triangleq \mathtt{C}_1{\rightarrow}\mathtt{P} : \mathtt{c}_1\langle \mathtt{v}{\cdot}\Delta_1\rangle ; \mathtt{C}_2{\rightarrow}\mathtt{P} : \mathtt{c}_2\langle \mathtt{v}{\cdot}\Delta_2\rangle ; \mathtt{C}_2{\rightarrow}\mathtt{P} : \mathtt{c}_2\langle \mathtt{v}{\cdot}\Delta_3\rangle ; \mathtt{C}_1{\rightarrow}\mathtt{P} : \mathtt{c}_1\langle \mathtt{v}{\cdot}\Delta_4\rangle.$$

We visually represent the protocol $\mathtt{G}$ using sequence diagrams, as per Fig. 5, where the arrows show the direction of transmission, and its labels show the engaged channel and/or the transmitted message. The per-party projection (middle diagram) only highlights the view of party $\mathtt{P}$, and the per-channel projection (rightmost diagram) highlights the views of channels $\mathtt{c}_1$ and $\mathtt{c}_2$, respectively, w.r.t. party $\mathtt{P}$ (ignoring the dashed arrows for now).

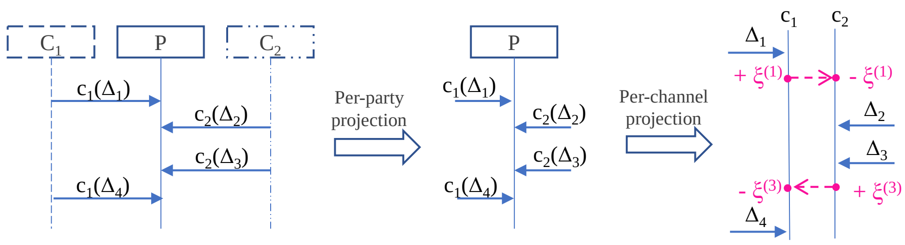

Fig. 5: A visual representation of protocol $\mathtt{G}$ (left sequenced diagram) projected onto party $\mathtt{P}$ (middle diagram), and then projected over channel $\mathtt{c}_1$ and $\mathtt{c}_2$, respectively (right diagram).

The specification of channel $\mathtt{c}_1$ features only the transmissions over this channel, loosing thus the information that Δ_1 should be transmitted before Δ_2. Similarly for channel $\mathtt{c}_2$, Δ_3 should be transmitted before Δ_4. To support such fine granularity, namely the per-channel specification, without breaking the sequence of transmissions when a party is engaging multiple channels, we propose the usage of a fencing mechanism. The fencing mechanism enforces a party to respect the correct order of transmissions across multiple channels (fences are represented by dashed arrows in the rightmost diagram of Fig. 5).

A fence is introduced w.r.t. a set of parties and a channel, say $\{\mathtt{C}_1, \mathtt{P}\}$ and $\mathtt{c}_1$ for the first transmission of $\mathtt{G}$, and must be proved to hold, locally, before $\mathtt{P}$ can engage channel $\mathtt{c}_2$ and before $\mathtt{C}_1$ can engage any other channel. Generally, a fence is denoted by $\xi(\{\mathtt{P}^*\}, \mathtt{c}, \mathtt{n})$, and is uniquely identified by the id $\mathtt{n}$. We employ a refinement mechanism which introduces fences after each transmission of a global protocol, and assume from now on that each global protocol is refined. The details of this refinement are trivial, and therefore omitted. For the ongoing example, protocol $\mathtt{G}$ is refined to:

$$\mathtt{C}_1{\rightarrow}\mathtt{P} : \mathtt{c}_1\langle \mathtt{v}{\cdot}\Delta_1\rangle; \xi(\{\mathtt{C}_1, \mathtt{P}\}, \mathtt{c}_1, 1); \mathtt{C}_2{\rightarrow}\mathtt{P} : \mathtt{c}_2\langle \mathtt{v}{\cdot}\Delta_2\rangle; \xi(\{\mathtt{C}_2, \mathtt{P}\}, \mathtt{c}_2, 2);$$
$$\mathtt{C}_2{\rightarrow}\mathtt{P} : \mathtt{c}_2\langle \mathtt{v}{\cdot}\Delta_3\rangle; \xi(\{\mathtt{C}_2, \mathtt{P}\}, \mathtt{c}_2, 3); \mathtt{C}_1{\rightarrow}\mathtt{P} : \mathtt{c}_1\langle \mathtt{v}{\cdot}\Delta_4\rangle; \xi(\{\mathtt{C}_1, \mathtt{P}\}, \mathtt{c}_1, 4).$$

Projection Language. Fig. 6 describes the two kinds of specification mentioned above. The per party specification language is depicted in Fig. 6a. Here, each send and receive specification refers to the communication instrument $\mathtt{c}$ along with a message $\mathtt{v}$ described by a formula Δ. The congruence of all the compound terms described in Sec. 2 holds for the projected languages as well, with the exception of sequential commutativity since the disjointness conditions for the latter do not hold (e.g. either the peer or the channel are implicitly the same for the entire projected specification). To note that, for brevity of this presentation, we denote the fences in the endpoint specification by the shorter notation $\xi^{(\mathtt{n})}$ since the party and the channel are implicit. Moreover, to note the notation for fence assumption, $\oplus\,\xi^{(\mathtt{n})}$, and that for fence guard, $\ominus\,\xi^{(\mathtt{n})}$, where the former is assumed to hold, and the latter needs to be proved to hold.

Local protocol	$\Upsilon\ ::=$	$\mathtt{L}\ ::=$
Send/Receive/Transmission	$\mathtt{c!v}\cdot\Delta \mid \mathtt{c?v}\cdot\Delta$	$\mathtt{!v}\cdot\Delta \mid \mathtt{?v}\cdot\Delta$
HO variable	$\mid \mathtt{V}$	$\mid \mathtt{V}$
Concurrency	$\mid \Upsilon{*}\Upsilon$	
Choice	$\mid \Upsilon\vee\Upsilon$	$\mid \mathtt{L}\vee\mathtt{L}$
Sequence	$\mid \Upsilon;\Upsilon$	$\mid \mathtt{L};\mathtt{L}$
Exists	$\mid \exists \mathtt{c}^*, \mathtt{v}^*\cdot\Upsilon$	$\mid \exists \mathtt{v}^*\cdot\mathtt{L}$
Fence	$\mid \xi(\{\mathtt{P}\},\mathtt{c},\mathtt{n})$	$\mid \oplus\,\xi^{(\mathtt{n})} \mid \ominus\,\xi^{(\mathtt{n})}$
Inaction	$\mid \mathtt{emp}$	$\mid \mathtt{emp}$
	(a) Per party	(b) Per channel

Fig. 6: Mercurius: The Projection Language

Automatic projection. Using different projection granularities should not permit event re-orderings (modulo $*$ composed events).

Proposition 1 (Projection Fidelity) *The projection to a decomposed specification, such as global protocol to per party, or per party to per channel, does not alter the communication pattern specified before the projection.*

To support the above proposition, we have designed a set of structural projection rules, described in Fig. 7. The rules Fig. 7a, describing per party projection rules, are standard, with the exception of disjunction and fences. As opposed to MPST, which projects the choice constructs to branching and selection, respectively, Mercurius maintains the disjunction through all the projection phases. It is able to do that since it relies on the verification system to reason about the underlying conditional constructs, verifying them against the disjunctive specification: sending expects a disjunctive abstract state, while receiving is creating a disjunctive abstract state. As expected, the per channel projection rules, Fig. 7b, strips the channel information from the per party specifications, since it will be implicitly available.

The projection of fences is a bit more subtle, and it obeys the following rules for per party and per channel projection, respectively:

$$(\xi(\{\mathtt{P}^*\},\mathtt{c},\mathtt{n}))\lfloor_{\mathtt{P}} \ := \begin{cases}\xi(\{\mathtt{P}\},\mathtt{c},\mathtt{n}) & \text{if}\ \mathtt{P}\in\{\mathtt{P}^*\}\\ \mathtt{emp} & \text{otherwise}\end{cases}$$

$$(\xi(\{\mathtt{P}\},\mathtt{c}_0,\mathtt{n}))\lfloor_{\mathtt{c}} \ := \begin{cases}\oplus\,\xi^{(\mathtt{n})} & \text{if}\ \mathtt{c}{=}\mathtt{c}_0\\ \ominus\,\xi^{(\mathtt{n})} & \text{if}\ \mathtt{c}{\neq}\mathtt{c}_0\end{cases}$$

Inserting a fence guard $\ominus\,\xi^{(\mathtt{n})}$ between adjacent transmissions on different channels on the same party ensures that the order of transmissions is accurately inherited from the corresponding per party specification across different channels. Fences are assumed to hold $\oplus\,\xi^{(\mathtt{n})}$, after consuming the transmission which introduced this fence.

Example 4: To emphasize the behavior of fences we consider the following sequence of receiving events captured by a per party specification, say $(\mathtt{G})\lfloor_{\mathtt{P}}$:

$$
\begin{array}{lcl}
(\mathtt{S}{\rightarrow}\mathtt{R} : \mathtt{c}\langle\Delta\rangle)\lfloor_{\mathtt{P}} & := & \begin{cases} \mathtt{c}!\mathtt{v}\cdot\Delta & \text{if}\, \mathtt{P}{=}\mathtt{S} \\ \mathtt{c}?\mathtt{v}\cdot\Delta & \text{if}\, \mathtt{P}{=}\mathtt{R} \\ \mathtt{emp} & \text{otherwise} \end{cases} \\
(\mathtt{G}_1{*}\mathtt{G}_2)\lfloor_{\mathtt{P}} & := & (\mathtt{G}_1)\lfloor_{\mathtt{P}} * (\mathtt{G}_2)\lfloor_{\mathtt{P}} \\
(\mathtt{G}_1{\vee}\mathtt{G}_2)\lfloor_{\mathtt{P}} & := & (\mathtt{G}_1)\lfloor_{\mathtt{P}} \vee (\mathtt{G}_2)\lfloor_{\mathtt{P}} \\
(\mathtt{G}_1;\mathtt{G}_2)\lfloor_{\mathtt{P}} & := & (\mathtt{G}_1)\lfloor_{\mathtt{P}}\, ;\, (\mathtt{G}_2)\lfloor_{\mathtt{P}} \\
(\exists \mathtt{P}_0^*, \mathtt{c}^*, \mathtt{v}^* \cdot \mathtt{G})\lfloor_{\mathtt{P}} & := & \exists \mathtt{c}^*, \mathtt{v}^* \cdot (\mathtt{G})\lfloor_{\mathtt{P}} \\
(\mathtt{emp})\lfloor_{\mathtt{P}} & := & \mathtt{emp}
\end{array}
$$

(a) global spec → per party spec

$$
\begin{array}{lcl}
(\mathtt{c}_0!\mathtt{v}\cdot\Delta)\lfloor_{\mathtt{c}} & := & \begin{cases} !\mathtt{v}\cdot\Delta & \text{if}\, \mathtt{c}{=}\mathtt{c}_0 \\ \mathtt{emp} & \text{otherwise} \end{cases} \\
(\mathtt{c}_0?\mathtt{v}\cdot\Delta)\lfloor_{\mathtt{c}} & := & \begin{cases} ?\mathtt{v}\cdot\Delta & \text{if}\, \mathtt{c}{=}\mathtt{c}_0 \\ \mathtt{emp} & \text{otherwise} \end{cases} \\
(\Upsilon_1{*}\Upsilon_2)\lfloor_{\mathtt{c}} & := & \begin{cases} (\Upsilon_{\mathtt{j}})\lfloor_{\mathtt{c}} & \text{if}\, \mathtt{c}{\in}\Upsilon_{\mathtt{j}}, \mathtt{j}{=}1,2 \\ \mathtt{emp} & \text{otherwise} \end{cases} \\
(\Upsilon_1{\vee}\Upsilon_2)\lfloor_{\mathtt{c}} & := & (\Upsilon_1)\lfloor_{\mathtt{c}} \vee (\Upsilon_2)\lfloor_{\mathtt{c}} \\
(\Upsilon_1;\Upsilon_2)\lfloor_{\mathtt{c}} & := & (\Upsilon_1)\lfloor_{\mathtt{c}}\, ;\, (\Upsilon_2)\lfloor_{\mathtt{c}} \\
(\exists \mathtt{c}_0^*, \mathtt{v}^* \cdot \Upsilon)\lfloor_{\mathtt{c}} & := & \exists \mathtt{v}^* \cdot (\Upsilon)\lfloor_{\mathtt{c}} \\
(\mathtt{emp})\lfloor_{\mathtt{c}} & := & \mathtt{emp}
\end{array}
$$

(b) per party spec → per endpoint spec

Fig. 7: Projection rules

$$
\begin{array}{l|l}
(\mathtt{G})\lfloor_{\mathtt{P}} & : \mathtt{c}_1?\mathtt{v}\cdot\Delta_1\, ;\, \xi(\{\mathtt{P}\},\mathtt{c}_1,1)\, ;\, \mathtt{c}_2?\mathtt{v}\cdot\Delta_2\, ;\, \xi(\{\mathtt{P}\},\mathtt{c}_2,2)\, ;\, \mathtt{c}_2?\mathtt{v}\cdot\Delta_3\, ;\, \xi(\{\mathtt{P}\},\mathtt{c}_2,3)\, ;\, \mathtt{c}_1?\mathtt{v}\cdot\Delta_4 \\
\hline
(\mathtt{G})\lfloor_{\mathtt{P},\mathtt{c}_1} & : \; ?\mathtt{v}\cdot\Delta_1 \;;\; \oplus\,\xi^{(1)} \;;\; \mathtt{emp} \;;\; \ominus\,\xi^{(2)} \;;\; \mathtt{emp} \;;\; \ominus\,\xi^{(3)} \;;\; \boxed{?\mathtt{v}\cdot\Delta_4} \\
(\mathtt{G})\lfloor_{\mathtt{P},\mathtt{c}_2} & : \; \mathtt{emp} \;;\; \ominus\,\xi^{(1)} \;;\; \boxed{?\mathtt{v}\cdot\Delta_2} \;;\; \oplus\,\xi^{(2)} \;;\; ?\mathtt{v}\cdot\Delta_3 \;;\; \oplus\,\xi^{(3)} \;;\; \mathtt{emp}
\end{array}
$$

The above local specification snapshot highlights how local fidelity is secured: the events marked with red boxes are guarded by their immediately preceding events, since they are handled by different channels. A subsequent refinement removes redundant guards, grayed in the example above, since adjacent same channel events need to guard only the last event on the considered channel.

Given the congruence of global protocols and local specifications, the projection is an isomorphism (closed under all operators). Specifically, given two protocols $\mathtt{G}_1$ and $\mathtt{G}_2$, with $\mathtt{P}_1..\mathtt{P}_\mathtt{n}{\in}\mathtt{G}_1$ such that $\forall\mathtt{P}{\in}\mathtt{G}_1 \Rightarrow \mathtt{P}{\in}\{\mathtt{P}_1..\mathtt{P}_\mathtt{n}\}$, and $\forall\mathtt{P}{\in}\mathtt{G}_2 \Rightarrow \mathtt{P}{\in}\{\mathtt{P}_1..\mathtt{P}_\mathtt{n}\}$, and $\forall\mathtt{P}{\in}\{\mathtt{P}_1..\mathtt{P}_\mathtt{n}\} \Rightarrow \mathtt{P}{\in}\mathtt{G}_2$, and with $\mathtt{c}_1..\mathtt{c}_\mathtt{m}{\in}\mathtt{G}_1$ such that $\forall\mathtt{c}{\in}\mathtt{G}_1 \Rightarrow \mathtt{c}{\in}\{\mathtt{c}_1..\mathtt{c}_\mathtt{m}\}$, and $\forall\mathtt{c}{\in}\mathtt{G}_2 \Rightarrow \mathtt{c}{\in}\{\mathtt{c}_1..\mathtt{c}_\mathtt{m}\}$, and $\forall\mathtt{c}{\in}\{\mathtt{c}_1..\mathtt{c}_\mathtt{m}\} \Rightarrow \mathtt{c}{\in}\mathtt{G}_2$ the following isomorphism holds:

$$\mathtt{G}_1{\equiv}\mathtt{G}_2{\Leftrightarrow}\{(\mathtt{G}_1)\lfloor_{\mathtt{P}_\mathtt{j}}\}_{\mathtt{j}=1..\mathtt{n}}{\equiv}\{(\mathtt{G}_2)\lfloor_{\mathtt{P}_\mathtt{j}}\}_{\mathtt{j}=1..\mathtt{n}}$$
$$\mathtt{G}_1{\equiv}\mathtt{G}_2{\Leftrightarrow}\{(\mathtt{G}_1)\lfloor_{\mathtt{P}_\mathtt{j},\mathtt{c}_\mathtt{k}}\}_{\mathtt{j}=1..\mathtt{n},\mathtt{k}=1..\mathtt{m}}{\equiv}\{(\mathtt{G}_2)\lfloor_{\mathtt{P}_\mathtt{j},\mathtt{c}_\mathtt{k}}\}_{\mathtt{j}=1..\mathtt{n},\mathtt{k}=1..\mathtt{m}}$$

4 Verification of C-like Programs

The user provides the global protocol which is then automatically refined according to the methodology described in Sec. 3. The refined protocol is then automatically projected onto a per party specification, followed by a per channel endpoint basis. Using such a modular approach where we provide a specification for each channel endpoint adds natural support for delegation, where a channel (as well as its specification) could be delegated to a third party in the style of binary session logic [14]. These communication specifications are made available in the program abstract state using a combination of ghost assertions and release lemmas (detailed in the subsequent). The verification could then automatically check whether a certain implementation follows the global protocol, after it had first bounded the *program elements* (processes and channel endpoints) to the *logical ones* (parties and channels).

Program	$\mathcal{P}$	$::= datat^*\ meth^*$
Data Struct.	$datat$	$::=$ `struct` $d\ \{\ (t\ f)^*\ \}$
Method Definitions	$meth$	$::= t\ mn\ ((t\ v)^*)\ requires\ \Phi,\ ensures\ \ \Phi\ \{e\}$
Types	t	$::= d \mid \tau$ $\qquad \tau\ ::=$ `int` $\mid$ `bool` $\mid$ `float` $\mid$ `void`
Expressions	e	$::=$ `NULL` $\mid k^\tau \mid v \mid$ `new` $d(v^*) \mid t\ v;\ e \mid v.f \mid mn(v^*) \mid$ `skip`
		$\mid v{:=}e \mid v.f{:=}e \mid e;e \mid e\|e \mid$ `if` $(b)\ e$ `else` $e \mid$ `return` e
		$\mid$ `open()` `with` $($`c`$,$`P`$^*) \mid$ `close`$(v) \mid$ `receive`$(v) \mid$ `send`(v,e)
Boolean Expressions	b	$::= e{=}{=}e \mid !(b) \mid b\&b \mid b\|b$

where $\mathtt{k}^\tau$ is a type τ constant, `v` is a program variable, `f` denotes a field

Fig. 8: A Core Imperative Language

Language. Fig. 8 depicts the syntax of a core language with support for communication primitives, where a program contains data and method definitions. Each method is decorated with a set of pre-/postconditions meant to guide the verification process. All of the program constructs are standard, with the exception of `open()` `with` $($`c`$,$ `P`$^*)$, which binds a logical channel `c` and parties `P`* to the channel reference returned by `open()`.

Concurrent Separation Logics. Due to its expressive power and elegant proofs, we choose to integrate our session logic on top of concurrent separation logic. Separation logic is an attractive extension of Hoare logic in which assertions are interpreted w.r.t. some relevant portion of the heap. Spatial conjunction, the core operator of separation logic, $\mathtt{P}*\mathtt{Q}$ divides the heap between two disjoint heaps described by assertions `P` and `Q`, respectively. The main benefit of this approach is the local reasoning: the specifications of a program code need only mention the portion of the resources which it uses, the rest are assumed unchanged. The details of the model and the semantics of the state assertions can be found in [12].

Verification. To check whether a user program follows the stipulated communication scenario, a traditional analysis would need to reason about the behaviour of a program using the operational semantics of the primitives' implementation. Since our goal is to emphasize on the benefits of implementing a protocol guided communication, rather than deciding the correctness of the primitives machinery, we adopt a specification strategy using abstract predicates [37,17] to describe the behavior of the program's primitives. Provided that the primitives respect their abstract specification, developers could then choose alternative communication libraries, without the need to re-construct the correctness proof of their underlying program.

The verification process follows the traditional forward verification rules, where the pre-conditions are checked for each method call, and if the check succeeds it adds their corresponding postcondition to the poststate. The verification of the method definition starts by assuming its precondition as the initial abstract state, and then inspects whether the postcondition holds after progressively checking each of the method's body instructions.

$$\frac{}{\{\mathtt{init(c)}\}\ \mathtt{open()\ with\ (c,P^*)}\ \{\mathtt{opened(c,P^*,res)}\}}\ [\underline{\textbf{OPEN}}]$$

$$\frac{}{\{\ \mathtt{empty(c,\tilde{c})}\}\ \mathtt{close(\tilde{c})}\ \{\ \mathtt{emp}\ \}}\ [\underline{\textbf{CLOSE}}]$$

$$\frac{\mathcal{I}\triangleq \mathtt{Peer(P)} * \mathtt{opened(c,P^*,\tilde{c})} \wedge \mathtt{P}{\in}\mathtt{P^*}}{\{\mathcal{C}(\mathtt{c,P,!v\cdot V(v);L}){*}\mathtt{V(x)}{*}\mathcal{I}\}\ \mathtt{send(\tilde{c},x)}\ \{\mathcal{C}(\mathtt{c,P,L}){*}\mathcal{I}\}}\ [\underline{\textbf{SEND}}]$$

$$\frac{\mathcal{I}\triangleq \mathtt{Peer(P)} * \mathtt{opened(c,P^*,\tilde{c})} \wedge \mathtt{P}{\in}\mathtt{P^*}}{\{\mathcal{C}(\mathtt{c,P,?v\cdot V(v);L}){*}\mathcal{I}\}\ \mathtt{recv(\tilde{c})}\ \{\mathcal{C}(\mathtt{c,P,L}){*}\mathtt{V(res)}{*}\mathcal{I}\}}\ [\underline{\textbf{RECV}}]$$

(a) Annotated communication primitives.

$$\begin{array}{ll}
\mathtt{G(\{P_1..P_n\},c^*)} & \Rightarrow \mathtt{Party(P_1,c^*,(G)\lfloor_{P_1})} * \ldots * \mathtt{Party(P_n,c^*,(G)\lfloor_{P_n})}\ *\ \mathtt{initall(c^*)}.\\
\mathtt{Party(P,\{c_1..c_m\},(G)\lfloor_P)} & \Rightarrow \mathcal{C}(\mathtt{c_1,P,(G)\lfloor_{P,c_1}}) * \ldots * \mathcal{C}(\mathtt{c_m,P,(G)\lfloor_{P,c_m}}) *\ \mathtt{bind(P,\{c_1..c_m\})}.\\
\mathtt{initall(\{c_1..c_m\})} & \Rightarrow \mathtt{init(c_1)} * \ldots * \mathtt{init(c_m)}.
\end{array}$$

(b) Splitting lemmas

$$\begin{array}{l}
\mathcal{C}(\mathtt{c,P_1,emp}) * \ldots * \mathcal{C}(\mathtt{c,P_n,emp}) \backslash *\ \mathtt{opened(c,\{P_1..P_n\},\tilde{c})}\ \Rightarrow\ \mathtt{empty(c,\tilde{c})}\\
\mathcal{C}(\mathtt{c_1,P,emp}) * \ldots * \mathcal{C}(\mathtt{c_m,P,emp}) \backslash *\ \mathtt{bind(P,\{c_1..c_m\})}\ \Rightarrow\ \mathtt{Party(P,c^*,emp)}
\end{array}$$

(c) Joining simpagation rules

$$\begin{array}{l}
\mathcal{C}(\mathtt{c,P},\oplus\,\xi^{(\mathtt{n})};\mathtt{L}) \Rightarrow \mathcal{C}(\mathtt{c,P,L}) \wedge \xi^{(\mathtt{n})}\\
\mathcal{C}(\mathtt{c,P},\ominus\,\xi^{(\mathtt{n})};\mathtt{L}) \wedge \xi^{(\mathtt{n})} \Rightarrow \mathcal{C}(\mathtt{c,P,L})
\end{array}$$

(d) Lemmas to handle fences

Fig. 9: Communication primitives

Abstract Specification. We define a set of abstract predicates to support session specification of different granularity. Some of these predicates have been progressively introduced across the paper, but for brevity we have omitted certain details. We resume their presentation here with more details:

$\texttt{Party}(\texttt{P}, \texttt{c}^*, \Upsilon)$	associates a local protocol projection Υ to its corresponding party $\texttt{P}$ and the set of channels $\texttt{c}^*$ used by $\texttt{P}$ to communicate with its peers;
$\texttt{Peer}(\texttt{P})$	flow-sensitively tracks the executing party, since the execution of parties can either be in parallel or sequentialized;
$\mathcal{C}(\texttt{c}, \texttt{P}, \texttt{L})$	associates an endpoint specification $\texttt{L}$ to its corresponding party $\texttt{P}$ and channel $\texttt{c}$;
$\texttt{initall}(\texttt{c}^*)/$ $\texttt{init}(\texttt{c})$	hold only when the specifications corresponding to logical channels $\texttt{c}^*/\texttt{c}$ are available (have been released into the abstract program state - Fig. 9b);
$\texttt{bind}(\texttt{P}, \texttt{c}^*)$	binds a party $\texttt{P}$ to all the channels $\texttt{c}^*$ it uses;
$\texttt{opened}(\texttt{c}, \texttt{P}^*, \tilde{\texttt{c}})$	binds a program channel $\tilde{\texttt{c}}$ to a logical one $\texttt{c}$ and to the peers sharing $\tilde{\texttt{c}}$;
$\texttt{empty}(\texttt{c}, \tilde{\texttt{c}})$	holds only when all the transmissions on $\tilde{\texttt{c}}$ have been consumed (Fig. 9c).

To cater for each verification phase, the session specifications with the required granularity are made available in the program's abstract state via the lemmas in Fig. 9.

Channel endpoint creation and closing described by the [**OPEN**] and [**CLOSE**] triples in Fig. 9, have mirrored specification: `open` associates the specification of a channel `c` to its corresponding program endpoint $\tilde{\texttt{c}}$. The keyword `res` is a dedicated ghost variable denoting the result returned by `open` in this particular case, and the result of evaluating the underlying expression in the general case. `close` regards the closing of a channel endpoint as safe only when all the parties have finished their communication w.r.t. the closing endpoint.

To support send and receive operations, we decorate the corresponding methods with dual generic specifications. The precondition of [**SEND**] ensures that indeed a send operation is expected, $!\texttt{v} \cdot \texttt{V}(\texttt{v})$, with the transmitted message `v` being described using a higher-order relation over `v`, namely $\texttt{V}(\texttt{v})$. To ensure memory safety, the verifier also checks whether the program state indeed owns the message to be transmitted and that it adheres to the properties described by the freshly discovered relation, $\texttt{V}(\texttt{x})$. Dually, [**RECV**] ensures that the receiving state gains the ownership of the transmitted message. Both specifications guarantee that the transmission is consumed by the expected party, $\texttt{Peer}(\texttt{P})$.

The proof obligations generated by this verifier are discharged to a Separation Logic solver in the form of enatailment checks, detailed in the subseqent.

Entailment. Traditionally, the logical entailment between formalae written in the symbolic heap fragment of separation logic is expressed as follows: $\Delta_a \vdash \Delta_c * \Delta_r$, where Δ_r comprises those residual resources described by Δ_a, but not by Δ_c. Intuitively, a valid entailment suggests that the resource models described by Δ_a are sufficient to conclude the availability of those described by Δ_c.

Since the proposed logic is tailored to support reasoning about communication primitives with generic protocol specifications, the entailment should also be able to interpret and instantiate such generic specifications. Therefore we equip

$$\underline{[\textbf{ENT-CHAN}-\textbf{MATCH}]}$$
$$\frac{\begin{array}{c}\Delta_{\mathtt{a}}\Rightarrow\mathtt{c}_1=\mathtt{c}_2 \quad \mathcal{C}(\mathtt{c}_1,\mathtt{P}_1,\mathtt{L}_{\mathtt{a}}) \vdash \mathcal{C}(\mathtt{c}_2,\mathtt{P}_2,\mathtt{L}_{\mathtt{c}}) \leadsto \mathtt{S}_1\\ \mathtt{S}_2=\{\pi^{\mathtt{e}}_{\mathtt{i}} \mid \pi^{\mathtt{e}}_{\mathtt{i}}\in\mathtt{S}_1 \text{ and } \mathtt{SAT}(\Delta_{\mathtt{a}}\wedge\pi^{\mathtt{e}}_{\mathtt{i}}) \text{ and } \mathtt{SAT}(\Delta_{\mathtt{c}}\wedge\pi^{\mathtt{e}}_{\mathtt{i}})\} \quad \bigvee_{\pi^{\mathtt{e}}\in\mathtt{S}_2}(\Delta_{\mathtt{a}}\wedge\pi^{\mathtt{e}})\vdash\Delta_{\mathtt{c}} \leadsto \mathtt{S}\end{array}}{\mathcal{C}(\mathtt{v}_1,\mathtt{P},\mathtt{L}_{\mathtt{a}})*\Delta_{\mathtt{a}} \vdash \mathcal{C}(\mathtt{v}_2,\mathtt{P},\mathtt{L}_{\mathtt{c}})*\Delta_{\mathtt{c}} \leadsto \mathtt{S}}$$

$$\underline{[\textbf{ENT-SEND}]}\quad \frac{[\mathtt{v}_1/\mathtt{v}_2]\Delta_{\mathtt{c}} \vdash \Delta_{\mathtt{a}} \leadsto \mathtt{S}' \quad \mathtt{S}=\{\pi^{\mathtt{e}}_{\mathtt{i}}|\pi^{\mathtt{e}}_{\mathtt{i}}\in\mathtt{S}'\}}{!\mathtt{v}_1\cdot\Delta_{\mathtt{a}} \vdash !\mathtt{v}_2\cdot\Delta_{\mathtt{c}} \leadsto \mathtt{S}} \qquad \underline{[\textbf{ENT-RECV}]}\quad \frac{\Delta_{\mathtt{a}} \vdash [\mathtt{v}_1/\mathtt{v}_2]\Delta_{\mathtt{c}} \leadsto \mathtt{S}' \quad \mathtt{S}=\{\pi^{\mathtt{e}}_{\mathtt{i}}|\pi^{\mathtt{e}}_{\mathtt{i}}\in\mathtt{S}'\}}{?\mathtt{v}_1\cdot\Delta_{\mathtt{a}} \vdash ?\mathtt{v}_2\cdot\Delta_{\mathtt{c}} \leadsto \mathtt{S}}$$

$$\underline{[\textbf{ENT-SEQ}]}\quad \frac{\square_{\mathtt{a}} \vdash \square_{\mathtt{c}} \leadsto \mathtt{S}_1 \quad \mathtt{L}_{\mathtt{a}} \vdash \mathtt{L}_{\mathtt{c}} \leadsto \mathtt{S}_2 \quad \text{where } \square ::= ?\mathtt{v}\cdot\Delta \mid !\mathtt{v}\cdot\Delta}{\square_{\mathtt{a}};\mathtt{L}_{\mathtt{a}} \vdash \square_{\mathtt{c}};\mathtt{L}_{\mathtt{c}} \leadsto \{\mathtt{emp}\wedge\pi_1\wedge\pi_2 \mid \pi_1\in\mathtt{S}_1 \text{ and } \pi_2\in\mathtt{S}_2\}} \qquad \underline{[\textbf{ENT-RHS}-\textbf{PVAR}]}\quad \frac{\mathtt{S}=\{\mathtt{emp}\wedge\mathtt{V}=\mathtt{L}_{\mathtt{a}}\}}{\mathtt{L}_{\mathtt{a}} \vdash \mathtt{V} \leadsto \mathtt{S}}$$

$$\underline{[\textbf{ENT-CHAN}]}\quad \frac{\mathtt{P}_1=\mathtt{P}_2 \quad \mathtt{L}_{\mathtt{a}} \vdash \mathtt{L}_{\mathtt{c}} \leadsto \mathtt{S}' \quad \mathtt{S}=\{\pi^{\mathtt{e}}_{\mathtt{i}}|\pi^{\mathtt{e}}_{\mathtt{i}}\in\mathtt{S}'\}}{\mathcal{C}(\mathtt{c},\mathtt{P}_1,\mathtt{L}_{\mathtt{a}}) \vdash \mathcal{C}(\mathtt{c},\mathtt{P}_2,\mathtt{L}_{\mathtt{c}}) \leadsto \mathtt{S}}$$

$$\underline{[\textbf{ENT-LHS}-\textbf{OR}]}\quad \frac{\mathtt{L}_{\mathtt{i}};\mathtt{L}_{\mathtt{a}} \vdash \mathtt{L}_{\mathtt{c}} \leadsto \mathtt{S}_{\mathtt{i}} \quad \mathtt{S}=\{\bigvee_{\mathtt{i}}\Delta_{\mathtt{i}} \mid \Delta_{\mathtt{i}}\in\mathtt{S}_{\mathtt{i}}\}}{(\bigvee_{\mathtt{i}}\mathtt{L}_{\mathtt{i}});\mathtt{L}_{\mathtt{a}} \vdash \mathtt{L}_{\mathtt{c}} \leadsto \mathtt{S}} \qquad \underline{[\textbf{ENT-RHS}-\textbf{OR}]}\quad \frac{\mathtt{L}_{\mathtt{a}} \vdash \mathtt{L}_{\mathtt{i}};\mathtt{L}_{\mathtt{c}} \leadsto \mathtt{S}_{\mathtt{i}} \quad \mathtt{S}=\bigcup\mathtt{S}_{\mathtt{i}}}{\mathtt{L}_{\mathtt{a}} \vdash (\bigvee_{\mathtt{i}}\mathtt{L}_{\mathtt{i}});\mathtt{L}_{\mathtt{c}} \leadsto \mathtt{S}}$$

$$\underline{[\textbf{ENT-LHS}-\textbf{HO}-\textbf{VAR}]}\quad \frac{\mathtt{V}\notin \text{fv}(\Delta_{\mathtt{c}}) \quad \mathtt{SAT}(\Delta_{\mathtt{c}}) \quad \text{fresh } \mathtt{w} \quad \mathtt{S}=\{\mathtt{emp}\wedge\mathtt{V}(\mathtt{w})=[\mathtt{w}/\mathtt{v}]\Delta_{\mathtt{c}}\}}{\mathtt{V}(\mathtt{v}) \vdash \Delta_{\mathtt{c}} \leadsto \mathtt{S}}$$

$$\underline{[\textbf{ENT-RHS}-\textbf{HO}-\textbf{VAR}]}\quad \frac{\mathtt{V}\notin \text{fv}(\Delta_{\mathtt{a}}) \quad \Delta_{\mathtt{a}} \vdash \Delta_{\mathtt{c}} \leadsto \mathtt{S}' \quad \text{fresh } \mathtt{w} \quad \mathtt{S}=\{\mathtt{emp}\wedge\mathtt{V}(\mathtt{w})=[\mathtt{w}/\mathtt{v}]\Delta_{\mathtt{i}}|\Delta_{\mathtt{i}}\in\mathtt{S}'\}}{\Delta_{\mathtt{a}} \vdash \mathtt{V}(\mathtt{v})*\Delta_{\mathtt{c}} \leadsto \mathtt{S}}$$

Fig. 10: Selected entailment rules: $\pi^{\mathtt{e}}$ is a shorthand for $\mathtt{emp}\wedge\pi$, $\mathtt{fv}(\Delta)$ returns all free variables in Δ, and `fresh` denotes a fresh variable.

the entailment checker to reason about formulae which contain second-order variables. Consequently, the proposed entailment is designed to support the instantiation of such variables. However, the instantiation might not be unique, so we collect the candidate instantiations in a set of residual states. The entailment has thus the following form: $\Delta_a \vdash \Delta_c \leadsto S$, where S is the set of possible residual states. Note that S is derived and its size should be of at least 1 in order to consider the entailment as valid. The entailment rules needed to accommodate session reasoning are given in Fig. 10. Other rules used for the manipulation of general resource predicates are adapted from Separation Logic [38].

To note also how [**ENT-RECV**] and [**ENT-SEND**] are soundly designed to be the dual of each other: while the former checks for covariant subsumption of the communication models, the latter enforces contravarinat subsumption since the information should only flow from a stronger constraint towards a weaker one. Considering the example below, a context expecting to read an integer greater than or equal to 1 could engage a channel designed with a more relaxed specification *(i)*. However, a context expecting to transmit an integer greater than or equal to 1 should only be allowed to engage a more specialized channel, such as one which designed to transmit solely 1 *(ii)*.

(i)

$$\dfrac{\dfrac{\mathtt{v_1{\geq}1} \vdash [\mathtt{v_1}/\mathtt{v_2}]\mathtt{v_2{\geq}0}}{?\mathtt{v_1}\cdot\mathtt{v_1{\geq}1} \vdash ?\mathtt{v_2}\cdot\mathtt{v_2{\geq}0}}\ {\scriptstyle\text{ENT-RECV}}}{\mathcal{C}(\mathtt{c},\mathtt{P},?\mathtt{v_1}\cdot\mathtt{v_1{\geq}1}) \vdash \mathcal{C}(\mathtt{c},\mathtt{P},?\mathtt{v_2}\cdot\mathtt{v_2{\geq}0})}\ {\scriptstyle\text{ENT-CHAN}}$$

(ii)

$$\dfrac{\dfrac{[\mathtt{v_1}/\mathtt{v_2}]\mathtt{v_2{=}1} \vdash \mathtt{v_1{\geq}1}}{!\mathtt{v_1}\cdot\mathtt{v_1{\geq}1} \vdash !\mathtt{v_2}\cdot\mathtt{v_2{=}1}}\ {\scriptstyle\text{ENT-SEND}}}{\mathcal{C}(\mathtt{c},\mathtt{P},!\mathtt{v_1}\cdot\mathtt{v_1{\geq}1}) \vdash \mathcal{C}(\mathtt{c},\mathtt{P},!\mathtt{v_2}\cdot\mathtt{v_2{=}1})}\ {\scriptstyle\text{ENT-CHAN}}$$

Soundness. The soundness of our verification rules is defined with respect to the operational semantics of [13] by proving progress and preservation. For lack of space we omit the soundness statement and its corresponding proofs, but their full details can be found in [13].

5 Implementation

We have implemented Mercurius in OCaml and attached it to a well established software verifier [12] for C-like languages. Even though this prototype implementation was build to tackle C-like programs, its design may be used to handle other languages as well, provided that this languages support communication primitives in the style of `send`/`receive`/`open`/`close`.

Moreover, the implementation is highly modular treating the communication primitives as function definitions annotated with generic specifications. On the one hand, thanks to the support for higher order variables, the `send`/`receive` functions exhibit a polymorphic behavior being used to transmit different types of values and different kinds of resources. On the other hand, the abstract behavior of the communication primitives may be changed by simply changing its abstract specification rather than changing the verifier's behavior. Using lemmas to handle the auxiliary predicates allows us to support changes to the logic by simply introducing new lemmas or changing the existing ones, lifting thus the burden of changing the underlying verifier. The prototype comprises about 6K lines of OCaml code, excluding the communication primitives (30 lines) and lemmas (103 lines), considered specifications and given as input files to the verifier.

We run the verifier to check the cloud service discussed in Sec. 1 for protocol conformance and communication safety. The results are depicted in Table 1. We checked the client-sever protocol against different implementations of the server, which are either purely sequential (Server-seq[1-3]) or contain some parallelism (Server-par[1-3]). No verification time took more than 8 seconds, despite the high number of generated proofs. The reason why the solver needs to handle so many proofs is that for each implementation, the verifier needs to re-check for well-formedness all the specifications and predicates decorating the program, as well as those within the configuration files for the primitives and lemmas.

Moreover we experimented both with the version of the cloud service which handles only one client (`CS`), as well as the one which supports multiple clients (`CLOUD`) sequentially (Server-seq) or in parallel (Server-par). For the (`CLOUD`) protocol, we picked only the implementation of the `CS` which communicates with the helpers concurrently, namely Server-par1. The verification worked seamlessly in both cases, without the need to tweak the specification in any way irrespective of the underlying implementation.

We also report our results on verifying a simple calculator adopted from [39]. As opposed to [39] though, Mercurius is unable to handle a memoizing calculator, since our lightweight approach did not instrument the global protocol to assert anything about how the communication affects the local state of each party. The changes in local states are only reflected by the generic specifications of the communication primitives (not by the protocol itself) indicating what is released into or consumed from the local state.

Lastly, we also report our results w.r.t. the "Rock, Paper, Scissors" protocol adopted from [15]. In [15] the authors claim that this kind of protocol and its logical pitfalls are common when building smart contracts - a form of distributed programs often engaged in cryptocurrency transactions. The logical bugs mentioned in [15], such as imprecise payments and inaccessible resources, may be avoided with a rigorous verification system. More examples and their detailed proofs can be found online [1], where the interested reader can also test Mercurius with her own protocols.

Example 5: In the subsequent we show how the specification of party `P` from example 4, guides the verification process to identify a buggy implementation:

//$\mathcal{C}(\mathtt{c1},\mathtt{P},?\mathtt{v}\cdot\Delta_1;\oplus\xi^{(1)};\ominus\xi^{(3)};?\mathtt{v}\cdot\Delta_4)*\mathcal{C}(\mathtt{c2},\mathtt{P},\ominus\xi^{(1)};?\mathtt{v}\cdot\Delta_2;\oplus\xi^{(2)};?\mathtt{v}\cdot\Delta_3;\oplus\xi^{(3)})$

1 `x = receive(c1);`

//$\mathcal{C}(\mathtt{c1},\mathtt{P},\oplus\xi^{(1)};\ominus\xi^{(3)};?\mathtt{v}\cdot\Delta_4)*\mathcal{C}(\mathtt{c2},\mathtt{P},\ominus\xi^{(1)};?\mathtt{v}\cdot\Delta_2;\oplus\xi^{(2)};?\mathtt{v}\cdot\Delta_3;\oplus\xi^{(3)})*\Delta_1$

//============ `fire assume lemma to release` $\xi^{(1)}$============

//$\mathcal{C}(\mathtt{c1},\mathtt{P},\ominus\xi^{(3)};?\mathtt{v}\cdot\Delta_4)*\mathcal{C}(\mathtt{c2},\mathtt{P},\ominus\xi^{(1)};?\mathtt{v}\cdot\Delta_2;\oplus\xi^{(2)};?\mathtt{v}\cdot\Delta_3;\oplus\xi^{(3)})*\Delta_1*\xi^{(1)}$

//============ `fire guard lemma on` $\xi^{(1)}$============

//$\mathcal{C}(\mathtt{c1},\mathtt{P},\ominus\xi^{(3)};?\mathtt{v}\cdot\Delta_4)*\mathcal{C}(\mathtt{c2},\mathtt{P},?\mathtt{v}\cdot\Delta_2;\oplus\xi^{(2)};?\mathtt{v}\cdot\Delta_3;\oplus\xi^{(3)})*\Delta_1*\xi^{(1)}$

2 `y = receive(c2);`

//$\mathcal{C}(\mathtt{c1},\mathtt{P},\ominus\xi^{(3)};?\mathtt{v}\cdot\Delta_4)*\mathcal{C}(\mathtt{c2},\mathtt{P},\oplus\xi^{(2)};?\mathtt{v}\cdot\Delta_3;\oplus\xi^{(3)})*\Delta_1*\xi^{(1)}*\Delta_2$

//============ `fire assume lemma to release` $\xi^{(2)}$============

//$\mathcal{C}(\mathtt{c1},\mathtt{P},\ominus\xi^{(3)};?\mathtt{v}\cdot\Delta_4)*\mathcal{C}(\mathtt{c2},\mathtt{P},?\mathtt{v}\cdot\Delta_3;\oplus\xi^{(3)})*\Delta_1*\xi^{(1)}*\Delta_2*\xi^{(2)}$

//`FAIL to verify the next receive on c1 since` $\xi^{(3)}$ `is not available`

3 `t = receive(c1);`

5 `z = receive(c2);`

where program's statements are numbered 1-4, and the program's abstract state is prefixed by //. A correct implementation expects lines `3` and `4` to be swapped such that fence $\xi^{(3)}$ required by `c1` is available in the program's abstract state. $\xi^{(3)}$ is only released in the after the second receive on `c2` is consumed.

6 Related Work

Behavioral Types. The behavioral types specify the expected interaction pattern of communicating entities. Most of seminal works develop type systems on the π-calculus [26] for deadlock [27]and livelock [25] detection. However, these system do not account for communication protocols, nor do they express messages in a logical form. To improve on the latter, Igarashi and Kobayashi propose

Component	LOC	Proofs.	Verification time (sec)
Multimedia Cloud Service **(CS) - 29 lines of spec**			
Client	2	958	1.1
Server-seq1	23	3987	6.7
Server-seq2	23	3987	6.7
Server-seq3	23	3987	6.7
Server-par1	38	3162	7.1
Server-par2	38	3345	7.2
Server-par3	34	3848	7.0
Multimedia Cloud Service **(CLOUD) - 32 lines of spec**			
Client	3	1342	1.4
Server-seq	40	4569	7.7
Server-par	45	4348	7.9
Simple Calculator **- 6 lines of spec**			
Client	3	575	0.8
Server-seq	4	1534	1.6
Server-par	7	933	1.4
"Rock, Paper, Scissors" **- 11 lines of spec**			
Client	2	728	1.0
Server-seq	6	2252	2.3
Server-par	8	1774	3.0

Table 1: Evaluation of Mercurius

an abstraction of the behavior of pi-calculus processes as generic types [24]. However, the generic type system finally throws away the information about base values such as integers, as opposed to our proposal which uses the messages' logical description to guide the verification of the implementation.

The session types [22,23] proposed by Honda et. al are probably the most intensely studied refinement of the behavioral type systems, since they offer the means of writing formal communication protocols in a concise and user-friendly manner. Extensions of session types add support for: exception handling [9,8], multithreaded functional languages [33,30,36], for MPI [31], for OO languages [16,18], and, similar to our approach, for `C`-like languages [34]. However, none of these approaches exploit the possibility of expressing messages in a more precise manner, since the type system constraints the messages to be abstracted to just types. Expressing the messages in logical form could uncover implementation bugs that would otherwise easily bypass a simple type check. Works such as [6,42,29,11] draw a correspondence between linear logic and different session types, while [40,4] combine session types with dependet type. While these works have the potential to exploit their results in linear logic, they solely tackle the type and numerical properties of the exchanged data. Our proposal goes beyond numerical properties to resources sharing.

Closer to our goal, Caires and Seco [7] propose behavioral separation for disciplining the interference of higher-order programs in the presence of concurrency, sharing and aliasing. Behavioural separation types build upon the knowledge of behavioural type theories, behavioral-spatial types [5], and separation logic.

More recently, [2] also promotes non-determinism and shared channels in an extension of linear logic-based session types. Even though these works permit inter-party resource sharing, they do not explore the idea of relaxed protocols in the sense described in this paper, where $*$ permits intra-party concurrency, adding thus less constraints over the underlying protocol implementation.

Concurrent Logics for Message Passing. The idea of coupling together the model theory of concurrent separation logic with that of Communicating Sequential Processes [20] is studied in [21]. The processes are modeled by using trace semantics, drawing an analogy between channels and heap cells, and distinguishing between separation in space from separation in time. Our proposal shares the same idea of distinguishing between separation in space and separation in time, by using the $*$ and ; operators, respectively. However, their model relies on process algebras, while we propose an expressive logic based on separation logic able to also tackle memory management.

Heap-Hop [41,32] is a sound proof system for copyless message passing managed by contracts. The system is integrated within a static analyzer which checks whether messages are safely transmitted. Similar with our proposal, this work is also based on separation logic. As opposed to ours, its communication model is limited to solely two party communication.

IronFleet [19], embedded in Dafny [28], supports the verification of large system focusing on their liveness and safety properties, and going as far as being able to tackle consensus protocols. However, though important, their verification efforts are not reusable, using highly specialized primitives and predicates to express each verified system. We propose a lighter, yet more generic, verification mechanisms, where the same communication primitives and predicates can be reused for most of the verification scenarios. Moreover, our specification language is designed to be accessible to less specialized system designers and developers, while still offering safety guarantees.

Designed concurrently with out logic, DISEL [39] is a domain specific language for describing, implementing and verifying distributed systems. The protocols are described in DISEL using state-transition systems, as opposed to the more concise protocols of session types. The authors have also exploited the benefits of separation logic for providing strong safety guarantees, embeddeding their proofs in Coq. On contrast, we promote automated verification, where instead of using mechanized proofs, we rely on our verifier to automatically find the proof or correctness or to identify bugs.

7 Discussions and Final Remarks

We have designed a multi-party session logic that goes beyond the traditional type checking system, by embedding the communication protocols as guiding tools for verification systems. We have shown how the messages can be described in the more precise and expressive logical form, without sacrificing the conciseness of type approaches. We have shown how to write relaxed protocols that offer wider design choices for the implementation of protocols. Moreover, we have shown how a lightweight specification system in the style of session types can be embedded into a deductive verification system to offer stronger correctness and safety guarantees than those offered by type-checking. Moreover, automation is achieved without sacrificing modularity. [13,1] discuss deadlock checking, delegation and recursion in Mercurius.

As part of future work, we investigate how to improve the expressiveness further such that Mercurius is able to handle more distributed properties, such as consensus. Moreover, we intend to extend this work to other less mainstream, yet important communication models, such as those using non-linear channels. We also intend to go beyond the current limits of our well-formed disjunctions.

References

1. Mercurius. `http://loris-5.d2.comp.nus.edu.sg/Mercurius`
2. Balzer, S., Pfenning, F.: Manifest Sharing with Session Types. PACMPL **1**(ICFP) (2017)
3. Bettini, L., Coppo, M., DAntoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global Progress in Dynamically Interleaved Multiparty Sessions. In: CONCUR. pp. 418–433. Springer (2008)
4. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A Theory of Design-by-Contract for Distributed Multiparty Interactions. In: CONCUR. vol. 6269. Springer (2010)
5. Caires, L.: Spatial-Behavioral Types for Concurrency and Resource Control in Distributed Systems. Theoretical Computer Science **402**(2-3), 120–141 (2008)
6. Caires, L., Pfenning, F.: Session Types As Intuitionistic Linear Propositions. In: CONCUR. pp. 222–236. Springer-Verlag, Berlin, Heidelberg (2010)
7. Caires, L., Seco, J.C.: The Type Discipline of Behavioral Separation. In: ACM SIGPLAN Notices. pp. 275–286. ACM (2013)
8. Capecchi, S., Giachino, E., Yoshida, N.: Global Escape in Multiparty Sessions. MSCS **26**, 156–295 (2014)
9. Carbone, M., Honda, K., Yoshida, N.: Structured Interactional Exceptions in Session Types. In: CONCUR. pp. 402–417. Springer (2008)
10. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: Multiparty Asynchronous Global Programming. SIGPLAN Not. **48**(1), 263–274 (Jan 2013)
11. Carbone, M., Montesi, F., Schrmann, C., Yoshida, N.: Multiparty Session Types as Coherence Proofs. In: CONCUR. vol. 42, pp. 412–426 (2015)
12. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Automated Verification of Shape, Size and Bag Properties via User-defined Predicates in Separation Logic. Sci. Comput. Program. **77**(9), 1006–1036 (Aug 2012)
13. Costea, A.: A Session Logic for Relaxed Communication Protocols. PhD dissertation, School of Computing, National University of Singapore (2017)
14. Craciun, F., Kiss, T., Costea, A.: Towards a Session Logic for Communication Protocols. In: ICECCS. pp. 140–149 (2015)
15. Delmolino, K., Arnett, M., Kosba, A., Miller, A., Shi, E.: Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In: FC. pp. 79–94. Springer (2016)
16. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session types for object-oriented languages. In: ECOOP. pp. 328–352. Springer-Verlag (2006)
17. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent Abstract Predicates. In: ECOOP. pp. 504–528. Springer-Verlag (2010)

18. Gay, S.J., Vasconcelos, V.T., Ravara, A., Gesbert, N., Caldeira, A.Z.: Modular Session Types for Distributed Object-oriented Programming. In: POPL. pp. 299–312 (2010)
19. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: IronFleet: Proving Practical Distributed Systems Correct. In: SOSP. pp. 1–17 (2015)
20. Hoare, C.A.R.: Communicating Sequential Processes. In: The origin of concurrent programming, pp. 413–443. Springer (1978)
21. Hoare, T., O'Hearn, P.: Separation Logic Semantics for Communicating Processes. Electronic Notes in Theoretical Computer Science **212**, 3–25 (2008)
22. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Discipline for Structured Communication-Based Programming. In: ESOP (1998)
23. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. Journal of the ACM **63**, 1–67 (2016)
24. Igarashi, A., Kobayashi, N.: A Generic Type System for the Pi-Calculus. Theoretical Computer Science **311**(1), 121 – 163 (2004)
25. Kobayashi, N.: Type Systems for Concurrent Processes: From Deadlock-freedom to Livelock-freedom, Time-boundedness. In: IFIP TCS. pp. 365–389. Springer (2000)
26. Kobayashi, N.: A Type System for Lock-free Processes. IC **177**(2) (2002)
27. Kobayashi, N., Laneve, C.: Deadlock Analysis of Unbounded Process Networks. IC **252**, 48–70 (2017)
28. Leino, K.R.M., Müller, P.: A Basis for Verifying Multi-Threaded Programs. In: ESOP. pp. 378–393. Springer (2009)
29. Lindley, S., Morris, J.G.: A Semantics for Propositions as Sessions. In: ESOP. pp. 560–584. Springer (2015)
30. Lindley, S., Morris, J.G.: Embedding Session Types in Haskell. In: Proceedings of the 9th International Symposium on Haskell. pp. 133–145. ACM (2016)
31. Lopez, H.A., Marques, E.R.B., Martins, F., Ng, N., Santos, C., Vasconcelos, V.T., Yoshida, N.: Protocol-Based Verification of Message-Passing Parallel Programs. In: OOPSLA'15. ACM (2015)
32. tienne Lozes, Villard, J.: Shared Contract-Obedient Channels. Science of Computer Programming (2015)
33. Neubauer, M., Thiemann, P.: An Implementation of Session Types. In: PADL. pp. 56–70. Springer (2004)
34. Ng, N., Yoshida, N., Honda, K.: Multiparty Session C: Safe Parallel Programming with Message Optimisation. In: TOOLS 2012. LNCS, vol. 7304. Springer (2012)
35. Nielson, F., Nielson, H.R.: From CML to its Process Algebra. Theoretical Computer Science **155**(1), 179–219 (1996)
36. Orchard, D., Yoshida, N.: Effects As Sessions, Sessions As Effects. In: POPL. pp. 568–581. ACM, New York, NY, USA (2016)
37. Parkinson, M., Bierman, G.: Separation Logic and Abstraction. In: ACM SIGPLAN Notices. pp. 247–258. ACM (2005)
38. Reynolds, J.C.: Separation Logic: A Logic For Shared Mutable Data Structures. In: LICS. pp. 55–74 (2002)
39. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and Proving with Distributed Protocols. POPL **2**, 28:1–28:30 (2018)
40. Toninho, B., Caires, L., Pfenning, F.: Dependent Session Types Via Intuitionistic Linear Type Theory. In: PPDP. pp. 161–172. ACM (2011)
41. Villard, J., Lozes, E., Calcagno, C.: Proving Copyless Message Passing. In: APLAS. pp. 194–209. Springer (2009)
42. Wadler, P.: Propositions As Sessions. In: ICFP. pp. 273–286. ACM (2012)