A multiparty multi-session logic

Laura Bocchi[1], Romain Demangeon[2], and Nobuko Yoshida[3]
[1]University of Leicester, [2]Queen Mary, University of London, [3]Imperial College London

Abstract. Recent work on the enhancement of typing techniques for multiparty sessions with logical annotations enables, not only the validation of structural properties of the conversations and on the sorts of the messages, but also properties on the actual values exchanged. However, specification and verification of mutual effects of multiple cross-session interactions are still an open problem. We introduce a multiparty logical proof system with virtual states that enables the tractable specification and validation of fine-grained inter-session correctness properties of processes participating in several interleaved sessions. We present a sound and relatively complete static verification method.

1 Introduction

In extensively distributed computing environments, application scenarios often centre around structured conversations among multiple distributed participants. A fundamental challenge is to establish an effective specification and verification method to ensure safety in distributed software, where correctness depends on the state of individual participants and span over multiple conversations and applications. This requirement emerged from our ongoing collaboration with the Ocean Observation Initiative OOI [21], an NSF program to develop a long-term computational infrastructure for environmental ocean observation. The principals within the OOI infrastructure perform interactive activities involving distributed resources, e.g., remote instruments, off-shore sensors, data. It is important to: (1) ensure that the principals carry out each activity (session) in a way that conform a well-defined protocol, (2) express properties that span the single activities (e.g., associate each principal with a credit for resource usage, and ensure that this will always be non negative across sessions[1]).

A promising direction is the logical elaboration of types for programming languages [16]. Types offer a stable linkage between the fundamental dynamics of programs and their mathematical abstractions, serving as a highly effective basis for safety assurance. In the context of process algebras, approaches like [5, 13, 18] allow tractable[2] (e.g., with respect to model checking techniques) validation of properties such as session fidelity, progress, and error freedom. Furthermore, they enable the specification of *global* properties of multiparty interactions, yet enabling modular *local* verification of each principal. The key idea is that conversations are built as the composition of units of design called *sessions* which are specified from a global perspective (i.e., a global session type). Each global type is then *projected*, making the responsibilities of each

[1] This example is taken from the OOI Instrument Control case study and is illustrated in the Appendix of the online report [4].

[2] In [13, 18] verification is decidable and has linear complexity.

endpoint explicit. Validation guarantees that when each endpoint conforms to its projected specification(s), the resulting conversation conforms to the corresponding global specification(s).

These approaches require to build applications starting from a set of global types that have to be agreed upon by the principals in the network. This assumption, which poses some limitations to the flexibility with which the single local processes are modelled, is reasonable in many scenarios, provided that local processes can be built as the composition of multiple, possibly interleaved, types of sessions. However, one limitation of these approaches is that the properties that they verify are confined to the single multiparty sessions and do not treat stateful specifications incorporating mutual effects of multiple sessions run by a principal.

This paper presents a simple but powerful extension of multiparty session specifications, by enriching the assertion language studied in [5] with capability to refer to *virtual states* local to each network principal. The resulting protocol specifications are called *multiparty stateful assertions* (*MPSA*s), and model the skeletal structure of the interactions of a session, the constraints on the exchanged messages and on the branches to be followed, and the *effects* of each interaction on the virtual state. We use *invariants* to express properties, on the state of each principal, that must hold even when several sessions are executed in parallel. Principals in a network hence serve as units of verification: static validation ensures that principals behave as prescribed by *MPSA*s and their invariants are satisfied.

To see the kind of properties we are interested in, consider the following fragment of specification for the dialogue between a ticket allocation server (`S`) and its client (`C`), where the server allocates numbered tickets of increasing value to each client in consecutive, *separate* sessions:

$$\mathtt{S} \to \mathtt{C} : (y{:}\mathtt{int})\{y = \mathtt{S.x}\}\langle \mathtt{S.x{+}{+}} \rangle$$

The protocol between the server and each client is the single message-passing action where `S` sends `C` a message of type `int`. The description of this simple distributed application implies behavioural constraints of greater depth than the basic communication actions. The (sender-side) *predicate and effect* for the interaction step, $\{y = \mathtt{S.x}\}\langle \mathtt{S.x{+}{+}} \rangle$, asserts that the message y sent to each client must equal the current value of `S.x`, a state variable x allocated to the *principal* serving as `S`; and that the local effect of this message send is to increment `S.x`. In this way, `S` is specified to send incremental values across *consecutive* sessions.

The behaviour described above cannot be encoded by only using the primitives in [5]. In fact, in order to ensure inter-session properties one must discipline concurrent state updates with some mechanism of lock/unlock or atomic access/update, but lock-/unlock and atomic access/update can only be described as properties that span over multiple sessions.

Contribution We present a sound and relatively complete validation method for *MPSA*s, based on statically-verifiable proof rules. The most distinctive feature with respect to [5] is the possibility of expressing properties that span several sessions. The decidability/complexity of verification depends on the decidability/complexity of predicate evaluation in the logic that is chosen to express constraints and invariants (Proposition 10). We prove that our analysis is sound (Theorem 13) and complete (Theorem 14)

w.r.t. the semantical satisfaction relation induced by the two labelled transition systems for processes and specifications.

Synopsis In § 2 we present *MPSA*s, that is the language for (stateful) protocol specifications, and their consistency criteria (i.e., well-assertedness). In § 3 we present the calculus for networks of principals, each running a process. In § 4 we give the validation rules of networks against *MPSA*s; their properties are presented in § 5. Related work is discussed in § 6. Use cases from [21], auxiliary definitions, and full proofs can be found in the online report [4].

2 Multiparty assertions with virtual states

In the proposed framework, applications are built as the composition of units of design called *sessions*. Each type of session is specified as a *MPSA*, that is an abstract description of the interactions of the roles of a multiparty session.

The syntax of *MPSA*s is given in Figure 1. *Global assertions* ($\mathcal{G}, \mathcal{G}', \ldots$) describe a multiparty session from a global perspective; and *local assertions* ($\mathcal{L}, \mathcal{L}', \ldots$) describe it from the perspective of one role. U are types of the message contents, which can be sorts S or local assertions $\langle\mathcal{L}\rangle$ (i.e., for delegation).

$A ::= \mathsf{true} \mid \mathsf{false} \mid e_1 = e_2 \mid \neg A \mid A_1 \wedge A_2 \mid \exists x.A, \quad U ::= S \mid \langle\mathcal{L}\rangle, \quad S ::= \mathtt{bool} \mid \mathtt{int} \mid ..$

$\mathcal{G} ::=$			$\mathcal{L} ::=$		
	$\mathtt{p} \to \mathtt{q} : \{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{G}_i\}_{i\in I}$	**(G-int)**		$\mathtt{p}!\{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{L}_i\}_{i\in I}$	**(L-sel)**
$\mid$	$\mathcal{G}_1 \mid \mathcal{G}_2$	**(G-par)**	$\mid$	$\mathtt{p}?\{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{L}_i\}_{i\in I}$	**(L-bra)**
$\mid$	$\mu \mathsf{t}\langle y : A'\rangle(x : S)\{A\}.\mathcal{G}$	**(G-def)**	$\mid$	$\mu \mathsf{t}\langle y : A'\rangle(x : S)\{A\}.\mathcal{L}$	**(L-def)**
$\mid$	$\mathsf{t}\langle y : A'\rangle$	**(G-call)**	$\mid$	$\mathsf{t}\langle y : A'\rangle$	**(L-call)**
$\mid$	end	**(G-end)**	$\mid$	end	**(L-end)**

Fig. 1. Global and local MPSAs

For expressing constraints we use *predicates* ($A, A', \ldots$) with the syntax illustrated in Figure 1, although we may use other predicates than equality in examples. Predicates are defined on *interaction variables*, modelling the content of a message exchanged by the roles in the session, and on *state variables*, which are variables of the virtual state local to one role.

Global Assertions Interaction **(G-int)**, $\mathtt{p} \to \mathtt{q} : \{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{G}_i\}_{i\in I}$, models a message exchange where role $\mathtt{p}$ sends $\mathtt{q}$ one of the branch labels l_i and an interaction variable x_i, with x_i binding its occurrences in A_i, E_i, and $\mathcal{G}_i$. A_i is the predicate which needs to hold for $\mathtt{p}$ to select l_i, and which may constrain the values to be sent for x_i. Note that A_i is at the same time an assumption for the receiver $\mathtt{q}$ and a constraint for the sender $\mathtt{p}$ (i.e., if A_i is violated then the blame is on $\mathtt{p}$). E_i is the update prescribed on the virtual states of $\mathtt{p}$ and $\mathtt{q}$, modelling the persistent effects (i.e., with respect to the lifetime of the single session) of that interaction. An update is a vector of assignments of the form $\mathtt{x} := e$, where $\mathtt{x}$ is updated by the result of evaluating e in the current state.

We assume E does not contain two assignments to the same state variable, and is an atomic action. Assertion **(G-par)** is for parallel composition. The recursive definition **(G-def)** is the guarded recursion definition and defines a recursion parameter x initially set equal to a value satisfying the initialisation predicate A', with A being an invariant predicate. Global assertions are unfolded implicitly, following an equi-recursive view on types. **(G-call)** is the recursive instantiation and **(G-end)** is the termination.

Hereafter we omit true predicates, empty vectors of variables/updates, and labels of single branches.

Example 1. Consider a session with two roles, C and S. C makes an offer x to S for buying a ticket; S either accepts or refuses the offer. In the former case C spends x credits and receives a ticket, and S earns x credits. Tickets are modelled as serial numbers; they must all be increasing numbers not exceeding 1000. $\mathcal{G}_T$ below specifies this scenario:

$$\begin{aligned}\mathcal{G}_T \;&= \mathtt{C} \to \mathtt{S} : (x : \mathtt{int})\{x \geqslant 0 \wedge \mathtt{C.credit} \geqslant x\}\langle \mathtt{C.credit} := \mathtt{C.credit} - x\rangle.\\ &\quad\; \mathtt{S} \to \mathtt{C} : \{\mathsf{ok}(y : \mathtt{int})\{\mathtt{S.count} < 1000 \wedge y = \mathtt{S.count}\}\langle E_{ok}\rangle.\mathsf{end},\\ &\qquad\qquad \mathsf{ko}\langle \mathtt{C.credit} := \mathtt{C.credit} + x\rangle.\mathsf{end}\ \}\\ E_{ok} &= \mathtt{S.credit} := \mathtt{S.credit} + x, \mathtt{S.count} := \mathtt{S.count} + 1\end{aligned}$$

C has state variable `credit`, and S has state variables `credit` and `count` (a counter for serial numbers). The first interaction requires that the offer x does not exceed C's credit, and decrements the credit by x. S selects one of the two branches by either label ok or ko. The former branch can be selected only if $\mathtt{S.count} < 1000$.

We denote with $var(\mathcal{G})$ the set of (interaction/state) variables and recursion parameters in $\mathcal{G}$, and with $var(A)$ the free variables of A (same for e). The set of variables that $\mathtt{p} \in \mathcal{G}$ *knows*, written $var(\mathcal{G}) \upharpoonright \mathtt{p}$, consists of: (i) the state variables of the form $\mathtt{p}.x$ for some x, (ii) the interaction variables sent or received by p in $\mathcal{G}$, and (iii) the parameters of the recursive definitions $\mu \mathsf{t}\langle y : A'\rangle(x : S)\{A\}.\mathcal{G}'$ in $\mathcal{G}$ such that p knows all the free variables in initialisation A', and all free variables in A'' for all $\mathsf{t}\langle y : A''\rangle$ in $\mathcal{G}'$ (we assume each recursion parameter known by exactly two participants).

Well-assertedness Our theory relies on two consistency principles: *history-sensitivity* and *temporal-satisfiability*. These principles were first introduced in [5]; we discuss them here as their adaptation to our stateful scenario requires non-trivial extensions.

By history-sensitivity each role must have enough information to fulfil the specified obligations, namely it requires that: (1) each role p knows all free variables in the predicates that p must guarantee, and (2) each role has enough information to perform the prescribed updates, that is (i) when to make an update, and (ii) which values to assign.

Definition 2 (History-sensitivity). $\mathcal{G}$ is history-sensitive if for each interaction, of the form $\mathtt{p} \to \mathtt{q} : \{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{G}_i\}_{i\in I}$, occurring in $\mathcal{G}$, for all $i \in I$:

1. $var(\mathcal{G}) \upharpoonright \mathtt{p} \supseteq var(A_i)$ (i.e., p knows all variables in $var(A_i)$),
2. for all $\mathtt{r}.x := e$ in E_i: (i) either $\mathtt{r} = \mathtt{p}$ or $\mathtt{r} = \mathtt{q}$, and (ii) $var(\mathcal{G}) \upharpoonright \mathtt{r} \supseteq var(e)$.

A checker for history-sensitivity can be found in the online report [4]. Consider the assertions:

$$\begin{aligned}\mathcal{G} \;&= \mathtt{p} \to \mathtt{q} : (x{:}\mathtt{int}).\ \mathtt{q} \to \mathtt{r} : (y{:}\mathtt{int}).\ \mathtt{r} \to \mathtt{s} : (z{:}\mathtt{int})\{z > x\}\\ \mathcal{G}' &= \mathtt{p} \to \mathtt{q} : (y{:}\mathtt{int}).\ \mathtt{q} \to \mathtt{r} : \{\mathsf{ok}(w{:}\mathtt{int})\langle \mathtt{r.x}_1 := y, \mathtt{p.x}_2 := y\rangle, \mathsf{ko}\}\end{aligned}$$

$\mathcal{G}$ violates (1) because $\mathtt{r}$ has to send a value for z that is greater than x without knowing x. $\mathcal{G}'$ violates both clauses of (2): (i) because $\mathtt{p}$ must update $\mathtt{x}_2$ not knowing whether and when the update should be done, and (ii) because in the second interaction $\mathtt{r}$ has to update $\mathtt{x}_1$ with y without knowing y.[3]

By temporal-satisfiability, for each participant $\mathtt{p} \in \mathcal{G}$, whenever it is $\mathtt{p}$'s turn to send a value, $\mathtt{p}$ can find at least one selection branch and one value which satisfies the specified constraint. Temporal satisfiability is defined (and checked) using a function $\mathtt{ts}(\mathcal{G}, A)$ which returns true only if $\mathcal{G}$ always allows a path of interactions going through $\mathcal{G}$ in *any possible state*. Considering all possible states makes the specification robust with respect to arbitrary interactions the same principal may be engaged in through other sessions. Predicate A is incrementally built as a conjunction of the predicates that appear in $\mathcal{G}$ in all the recursive invocations and models the current set of assumptions.

Definition 3 (Temporal-satisfiability). Let $\mathcal{G}$ be a global specification, and A a predicate. $\mathtt{ts}(\mathcal{G}, A)$ is given by:

1. $\mathtt{ts}(\mathtt{p}\to\mathtt{q} : \{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{G}_i\}_{i\in I}, A) = \begin{cases} \bigwedge_{i\in I} \mathtt{ts}(\mathcal{G}_i, A \wedge \underline{A_i}) & \text{if } A \supset \bigvee_{i\in I} \exists x_i.A_i \\ \mathsf{false} & \text{otherwise} \end{cases}$
2. $\mathtt{ts}(\mathcal{G}_1 \mid \mathcal{G}_2, A) = \mathtt{ts}(\mathcal{G}_1, A) \wedge \mathtt{ts}(\mathcal{G}_2, A)$
3. $\mathtt{ts}(\mu \mathtt{t}\langle e\rangle(x\!:\!S)\{A'\}.\mathcal{G}', A) = \begin{cases} \mathtt{ts}(\mathcal{G}', A \wedge A') & \text{if } A \supset (A'[e/x]) \\ \mathsf{false} & \text{otherwise} \end{cases}$
4. $\mathtt{ts}(t_{A'(x)}\langle e\rangle, A) = \begin{cases} \mathsf{true} & \text{if } A \supset A'[e/x] \\ \mathsf{false} & \text{otherwise} \end{cases}$
5. $\mathtt{ts}(\mathsf{end}, A) = \mathsf{true}$

$\mathcal{G}$ satisfies temporal satisfiability if $\mathtt{ts}(\mathcal{G}, \mathsf{true}) = \mathsf{true}$.[4]

In (1) the first condition for "if" demands that there exists at least one branch for which it is possible to find a value for x_i that satisfies the current predicate A_i. The function is called recursively extending the set of preconditions A with with the closure $\underline{A_i}$ of predicate A_i (see Remark 4 below). (2) demands both parts of the composition are satisfiable. (3) and (4) check recursion, the latter relying on the annotation of recursive calls with the invariants of the corresponding recursive definitions.

Remark 4. The closure of a predicate A in $\mathcal{G}$, written $\underline{A}$, is the predicate obtained by closing with existential quantifiers the free state variables of $\mathcal{G}$ in A. Whereas the values of interaction variables in a session do not change after they are introduced[5], state variables can be updated a number of times in a number of different ways. Hence

[3] [6] proposes algorithms to amend assertions that violate history-sensitivity and temporal-satisfiability as in [5]. No such algorithms have yet been investigated for the definitions introduced in this paper. Although relevant, the issue of amending inconsistent assertions is out of the scope of the current work.

[4] This property can be relaxed by starting from a stronger precondition A as long as A is then implied by the principal invariants (which are defined in § 4).

[5] Actually, interaction variables in a recursion body are reused at each iteration. However, their values are due to follow the same constraints at each iteration.

a predicate on state variables may be true at a certain time, and become false at a later time. Hereafter we use $\underline{A}$ when we want to to 'keep' only the persistent assumptions (those on interaction variables) of A.

The following global specification violates temporal satisfiability

$$\mathtt{p} \to \mathtt{q} : (x : \mathtt{int})\{x > 0\}.\mathtt{q} \to \mathtt{p} : (y : \mathtt{int})\{y = x \wedge y > 100\}$$

In fact, in the first interaction p is allowed to choose any positive value for x, for instance 10. In this case, q cannot find any value for y such that $y = 10 \wedge y > 100$.

Proposition 5. *Given a global assertion $\mathcal{G}$, let m be the size of the syntactic tree of $\mathcal{G}$, n be the maximum number of variables occurring in each predicate in $\mathcal{G}$, and $eval(A)$ be the complexity of predicate evaluation (if decidable). History-sensitivity can be checked in $O(m \times n)$. Temporal-satisfiability is decidable if predicate evaluation is decidable and, if decidable, it can be checked in $O(m) \times eval(A)$.*

Hereafter, we assume assertions to be *well-asserted*.

Local Assertions Each local assertion $\mathcal{L}$ refers to a specific role. Syntax is given in the right part of Figure 1. Selection **(L-sel)** $\mathtt{p}!\{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{L}_i\}_{i\in I}$ models an interaction where the role sends p a branch label l_i and a message x_i. A_i and E_i are the predicate and update respectively. **(L-bra)** is the dual branching. The others are as in the global assertions, except that a local assertion cannot be multi-threaded.

Given a global assertion $\mathcal{G}$, we can automatically derive the local assertions for each role $\mathtt{p} \in \mathcal{G}$ by *projection*. The projection rules rely on a few auxiliary definitions: projection of a predicate, and projection of an update. The *projection of a predicate* A on p in $\mathcal{G}$, written $A \upharpoonright \mathtt{p}$, is defined as $\exists\tilde{x}.A$ where $\tilde{x} = var(A)\backslash(var(\mathcal{G}) \upharpoonright \mathtt{p})$ (i.e., the existential closure of the variables that p does not know). The *projection of an update* E on p in $\mathcal{G}$, written $E \upharpoonright \mathtt{p}$ is the update E' containing only the assignments $\mathtt{p}_j.\mathtt{x}_i := e_j$ such that $\mathtt{p}_j = \mathtt{p}$.

The projection rules for global assertions are as in [5], except that updates are now considered; their detailed presentation is not necessary to understand the results in this paper, hence we only give an illustration through Example 6. Henceforth, in $\mathcal{G} \upharpoonright \mathtt{p}$ we shall omit the $\mathtt{p}.\_$ prefix when referring to p's state variables.

Example 6. $\mathcal{L}_\mathtt{C}$ (resp. $\mathcal{L}_\mathtt{S}$) is the projection of $\mathcal{G}_T$ from Example 1 on C (resp. S).

$$\begin{aligned}
\mathcal{L}_\mathtt{C} &= \mathtt{S}\,!\,(x : \mathtt{int})\{x \geqslant 0 \wedge \mathtt{credit} \geqslant \mathtt{x}\}\langle \mathtt{credit} := \mathtt{credit} - x\rangle.\mathcal{L}'_\mathtt{C}\\
\mathcal{L}'_\mathtt{C} &= \mathtt{S}\,?\{\mathsf{ok}(y : \mathtt{int})\{\exists \mathtt{S.count}.\mathtt{S.count} < 1000 \wedge y = \mathtt{S.count}\}.\mathsf{end},\\
&\qquad \mathsf{ko}\langle \mathtt{credit} := \mathtt{credit} + x\rangle.\mathsf{end}\}\\
\mathcal{L}_\mathtt{S} &= \mathtt{C}\,?(x : \mathtt{int})\{\exists \mathtt{C.credit}.x \geqslant 0 \wedge \mathtt{C.credit} \geqslant x\}.\mathcal{L}'_\mathtt{S}\\
\mathcal{L}'_\mathtt{S} &= \mathtt{C}\,!\,\{\mathsf{ok}(y : \mathtt{int})\{\mathtt{count} < 1000 \wedge y = \mathtt{count}\}\\
&\qquad\qquad \langle \mathtt{credit} := \mathtt{credit} + x, \mathtt{count} := \mathtt{count} + 1\rangle.\mathsf{end},\\
&\qquad \mathsf{ko}.\mathsf{end}\}
\end{aligned}$$

The projection of the first interaction of $\mathcal{G}_T$ on sender C (resp. receiver S) is a send/select (resp. a receive/branch). The predicates/updates of the projections on a role are the projections of the predicates/updates on that role.[6] The continuation is projected similarly,

[6] Note that by well-assertedness (clause 1) the projection of a predicate on the sender of an interaction is always the predicate itself.

proceeding point-wise for each branch. Sometimes the projected predicate includes information about constraints of interactions between third parties (without however revealing the actual values exchanged by the third parties), e.g., $\exists \mathtt{S.count}.\mathtt{S.count} < 1000 \wedge y = \mathtt{S.count}$ provides $\mathtt{C}$ with precondition $y < 1000$.

Well-assertedness is easily extended to local assertions.

3 Multiparty networks with local states

We consider networks of interactional entities called *principals* linked by a common global transport, modelled as queues. Each principal runs a *located process*, that is a process with multiparty session primitives [1, 18] (to enable rigorous representation of conversation structures) and with a *local state*.

Syntax The syntax of networks and processes is given in Figure 2 and is a refined version of the multiparty session π-calculus from [1, 10] with local states. A local state σ maps a signature $[\tilde{\mathtt{x}} : \tilde{S}]$ of typed pairwise disjoint state variables $\tilde{\mathtt{x}}$ to their sorts. We use the injective function $\mathtt{id}(\sigma)$ to map each local state to an identifier.

A network can be an empty network $\varnothing$, a located process $[P]\sigma$, a parallel composition of networks $N_1 \mid N_2$, a new session name $(\nu s)N$ which binds s in N, or a queue $s : h$ where h are messages in transit through session channel s. A network is *initial* if it has no new session names and queues, otherwise it is *runtime*. We denote the free session channels in N with $\mathsf{fn}(N)$, similarly for P with $\mathsf{fn}([P]\sigma) = \mathsf{fn}(P)$.

$$
\begin{array}{lll}
\text{(network)} & N ::= \varnothing \mid [P]\sigma \mid N_1 | N_2 \mid (\nu s)N \mid s : h & \\
\text{(state/queue/value)} & \sigma ::= [\tilde{\mathtt{x}} : \tilde{S}] \mapsto \tilde{S} \quad h ::= \varnothing \mid (\mathtt{p}, \mathtt{q}, l\langle v \rangle) \cdot h \quad v ::= \mathtt{n} \mid s[\mathtt{p}] & \\
\text{(process)} \quad P ::= \overline{a}[\mathtt{n}](y).P & \textbf{(P-req)} & \mid P \mid Q \quad \textbf{(P-par)} \\
\quad \mid a[\mathtt{i}](y).P & \textbf{(P-acc)} & \mid (\mu X(x).P)\langle e \rangle \quad \textbf{(P-def)} \\
\quad \mid k[\mathtt{p}, \mathtt{q}]!\{e_i \mapsto l_i\langle e_i' \rangle(x_i)\langle E_i \rangle; P_i\}_{i \in I} & \textbf{(P-sel)} & \mid X\langle e \rangle \quad \textbf{(P-call)} \\
\quad \mid k[\mathtt{p}, \mathtt{q}]?\{l_i(x_i)\langle E_i \rangle.P_i\}_{i \in I} & \textbf{(P-bra)} & \mid \mathbf{0} \quad \textbf{(P-idle)} \\
\text{(channel/update/exp)} \quad k ::= y \mid s \quad E ::= \varnothing \mid E; \mathtt{x} := e \quad e ::= v \mid e \, \mathtt{op} \, e &&
\end{array}
$$

$x, y, \ldots$ interaction variables $\quad \mathtt{x}, \mathtt{y}, \ldots$ state variables $\quad X, Y, \ldots$ process variables
$a, b, \ldots$ shared name $\quad s, s', \ldots$ session name $\quad \mathtt{n}, \mathtt{n}', \ldots$ constants

Fig. 2. Syntax of networks and processes

Session request **(P-req)** multicasts a request to each session accept process **(P-acc)**, e.g., $a[\mathtt{i}](y).P$ with $i \in \{2, .., n\}$, by synchronisation through a shared name a and continuing as P. **(P-sel)** is Dijkstra's guarded command [15] and **(P-bra)** is the branching process; they represent communications through an established session k. **(P-sel)** acts as role $\mathtt{p}$ in session k and sends role $\mathtt{q}$ one of the labels l_i. The choice of the label is determined by boolean expressions e_i, assuming $\vee_{i \in I} e_i = \mathsf{true}$ and $i \neq j$ implies

$e_i \wedge e_j = \mathsf{false}$. Each label l_i is sent with the corresponding expression e'_i which specifies the value for x_i, assuming e'_i and x_i have the same type.[7] **(P-bra)** plays role $\mathtt{q}$ in session k and is ready to receive from $\mathtt{p}$ one of the labels l_i and a value for the corresponding x_i, then behaves as P_i after instantiating x_i with the received value. In guarded command (resp. branching), the local state of the sender (resp. receiver) is updated according to update E_i; in both processes each x_i binds its occurrences in P_i and E_i.

Example 7. Processes $P_{\mathtt{S}}$ and $P_{\mathtt{C}}$ implement $\mathcal{L}_{\mathtt{S}}$ and $\mathcal{L}_{\mathtt{C}}$, respectively, from Example 6.

$$P_{\mathtt{S}} = a[\mathtt{2}](z).z[\mathtt{C},\mathtt{S}]?(x); P'_{\mathtt{S}} \qquad E_{ok} = \mathtt{count} := \mathtt{count} + 1, \mathtt{credit} := \mathtt{credit} + x$$
$$P'_{\mathtt{S}} = z[\mathtt{S},\mathtt{C}]!\{\{\mathtt{count} < 1000 \wedge x \geqslant 10\} \mapsto \mathsf{ok}\langle \mathtt{count}\rangle(y)\langle E_{ok}\rangle.\mathbf{0},\ \{\mathtt{count} \geqslant 1000 \vee x < 10\} \mapsto \mathsf{ko}.\mathbf{0}\}$$
$$P_{\mathtt{C}} = \overline{a}[\mathtt{2}](w).w[\mathtt{C},\mathtt{S}]!\langle 8\rangle\langle x\rangle\langle \mathtt{credit} := \mathtt{credit} - x\rangle; P'_{\mathtt{C}}$$
$$P'_{\mathtt{C}} = w[\mathtt{S},\mathtt{C}]?\{\mathsf{ok}(y).\mathbf{0},\ \mathsf{ko}\langle \mathtt{credit} := \mathtt{credit} + x\rangle.\mathbf{0}\}$$

We let $\mathtt{C} = \mathtt{1}$ and $\mathtt{S} = \mathtt{2}$. $P_{\mathtt{S}}$ accepts a request to participate to a session specified by $\mathcal{G}_T$ (assuming a has type $\mathcal{G}_T$) on channel z as role $\mathtt{2}$. In the established session z, the principal receives an offer x from the co-party. It follows a guarded command with two cases; if $\mathtt{count}$ has not reached its maximum value for serial numbers and the offer is greater than 10 then the first branch (ok) is taken and $\mathtt{count}$ is sent as y, otherwise the second branch (ko) is taken. Dually, $P_{\mathtt{C}}$ sends a request to participate to one instance of session $\mathcal{G}_T$ as the role $\mathtt{1}$. A principal may repeatedly execute a process using recursion, or run concurrent instances of the same type of session (e.g., $[P_{\mathtt{S}} \mid P_{\mathtt{S}}]\sigma$) or different types of session (e.g., $[P_{\mathtt{S}} \mid P_{\mathtt{C}}]\sigma$) as discussed in Example 9.

Operational semantics The LTS is generated from the rules in Figure 3 using the following labels: $\ell ::= \overline{a}[\mathtt{n}]\langle s\rangle \mid a[\mathtt{i}]\langle s\rangle \mid s[\mathtt{p},\mathtt{q}]!l\langle v\rangle \mid s[\mathtt{p},\mathtt{q}]?l\langle v\rangle \mid \tau$. We denote with $\sigma\ \mathtt{after}\ E$ the state σ after the update E. We write $\sigma \models e \downarrow v$ for a closed expression e when it evaluates to v in σ.
The first and second rule are for requesting and accepting a session initialisation. The guarded command checks if condition e_j is satisfied in the current state σ, and sends a message consisting of one of the labels l_j and an expression e'_j (which is evaluated to a value v in state σ), updates σ according to E_j, and behaves as $P[v/x_j]$. Branching is symmetric. The synchronous session initialisation creates a new queue. We omit the standard context/structural equivalence rules.

4 Proof system for multiparty session logic with virtual states

In this section we outline how to obtain the syntactic validation of networks, written $\Gamma \vdash N \rhd \Sigma$, assuming processes typable, following [5]. The proof rules rely on the following environments:

$$\Gamma ::= \varnothing \mid \Gamma, a : \mathcal{G} \mid \Gamma, X : (x : S)\mathcal{L}_1 @ \mathtt{p}_1, \ldots, \mathcal{L}_n @ \mathtt{p}_n, \qquad \Delta ::= \varnothing \mid \Delta, s[\mathtt{p}] : \mathcal{L},$$
$$\Sigma ::= \varnothing \mid \Sigma, [\Delta]\sigma$$

[7] Guarded command can be implemented using selection, if-then-else and lock-unlock. Although our theory is applicable to these primitives, we choose to make these low-level steps atomic for minimising the syntax.

$$[\overline{a}[\mathtt{n}](y).P]\sigma \xrightarrow{\overline{a}[\mathtt{n}]\langle s\rangle} [P[s/y]]\sigma \qquad [a[\mathtt{i}](y).P]\sigma \xrightarrow{a[i]\langle s\rangle} [P[s/y]]\sigma \quad (s \notin \mathsf{fn}(P))$$

$$[s[\mathtt{p},\mathtt{q}]!\{e_i \mapsto l_i\langle e_i'\rangle(x_i)\langle E_i\rangle; P_i\}_{i\in I}]\sigma \xrightarrow{s[\mathtt{p},\mathtt{q}]!l_j\langle v\rangle} [P[v/x_j]]\sigma'$$
$$(j \in I \quad \sigma \models e_j' \downarrow v \quad \sigma \models e_j \quad \sigma' = \sigma \,\mathtt{after}\, E_j[v/x_j])$$

$$[s[\mathtt{p},\mathtt{q}]?\{l_i(x_i)\langle E_i\rangle.P_i\}_{i\in I}]\sigma \xrightarrow{s[\mathtt{p},\mathtt{q}]?l_j\langle v\rangle} [P_j[v/x_j]]\sigma' \qquad (j \in I \quad \sigma' = \sigma \,\mathtt{after}\, E_j[v/x_j])$$

$$\frac{[P_1]\sigma_1 \xrightarrow{\overline{a}[\mathtt{n}]\langle s\rangle} [P_1']\sigma_1 \quad [P_i]\sigma_i \xrightarrow{a[i]\langle s\rangle} [P_i']\sigma_i \quad (2 \leqslant i \leqslant n)}{[P_1]\sigma_1 \mid \cdots \mid [P_n]\sigma_n \xrightarrow{\tau} (\nu s)(s : \varnothing \mid [P_1']\sigma_1 \mid \cdots \mid [P_n']\sigma_n)}$$

$$\frac{[P]\sigma \xrightarrow{s[\mathtt{p},\mathtt{q}]!l_j\langle v\rangle} [P']\sigma'}{[P]\sigma \mid s : h \xrightarrow{\tau} [P']\sigma' \mid s : h \cdot (\mathtt{p},\mathtt{q}, l_j\langle v\rangle)} \qquad \frac{[P]\sigma \xrightarrow{s[\mathtt{p},\mathtt{q}]?l_j\langle v\rangle} [P']\sigma'}{[P]\sigma \mid s : (\mathtt{p},\mathtt{q}, l_j\langle v\rangle) \cdot h \xrightarrow{\tau} [P']\sigma' \mid s : h}$$

Fig. 3. Labelled transition for networks

Γ maps shared names to global assertions and process variables to their parameters. If $\Gamma \vdash a : \mathcal{G}$ then a session specified by $\mathcal{G}$ can be initiated by processes (via session request or accept) using a. By the standard kinding rules, we check if the same free variable appears in different global types in Γ, then they have the same sort. The mapping of process variables is for the validation of recursive assertions. Δ maps session channels/roles to local assertions. If $\Delta \vdash s[\mathtt{p}] : \mathcal{L}$ then a session is active (i.e., it has been initialized) on channel s for role $\mathtt{p}$; $\mathcal{L}$ specifies the (part of the) session that has still to be executed. Σ is the specification of a network; each $[\Delta]_\sigma$ is the specification of a located process with the respective virtual state.

We also use an *assertion environment* $\mathcal{C}$, which is incrementally built by conjunction of the predicates and boolean expressions (i.e., the conditions of a guarded commands) occurring in the processes being validated, and models their assumptions. Hereafter, given a predicate A and an update E, we define $A \,\mathtt{after}\, E$ to be the predicate obtained by substituting, for each assignment $x := e$ in E, each occurrence of x in A with e.

Modelling cross-session properties: the principal invariant Given a located process $[P]\sigma$ in a network, we want to allow the architect to model stable properties (i.e., invariant) over the variables in σ on across multiple sessions. We call these properties *principal invariant* of $[P]\sigma$, that is a predicate (following the syntax for A in Figure 1) over the state variables of σ. Hereafter we assume there exists a function $\mathcal{I}(\sigma)$ that given a local state σ returns the principal invariant for σ. Principal invariants depend from the application domain, and the architect should define them prior to the verification.

Example 8. Consider a located process $[P_\mathtt{c} \mid P_\mathtt{s}]\sigma_p$ with $P_\mathtt{c}$ and $P_\mathtt{s}$ from Example 7. Assume we want to require that the credit is always non-negative (i.e., the principal does not contracts debts) and that the counter does not exceed the maximum number of tickets which is 1000. We can enforce these constraints by setting the principal invariant $\mathcal{I}(\sigma_p)$ to be $\mathtt{credit} \geqslant 0 \wedge 0 \leqslant \mathtt{count} \leqslant 1000$.

Proof rules Figure 4 illustrates the proof rules for initial networks and processes.

$$\frac{\mathtt{id}(\sigma_p) = \mathtt{id}(\sigma_a) \quad \sigma_p, \sigma_a \models \mathcal{I}(\sigma_p) \quad \mathcal{I}(\sigma_p) \wedge \mathcal{C}; \Gamma \vdash P \rhd \Delta}{\mathcal{C}; \Gamma \vdash [P]\sigma_p \rhd [\Delta]\sigma_a} \quad \lfloor \text{N1} \rfloor$$

$$\frac{(\Gamma', \Delta', \sigma') \ni (\Gamma, \Delta, \sigma) \quad \mathcal{C} \supset \mathcal{C}' \quad \mathcal{C}'; \Gamma' \vdash N \rhd [\Delta']\sigma'}{\mathcal{C}; \Gamma \vdash N \rhd [\Delta]\sigma} \quad \lfloor \text{N2} \rfloor$$

$$\frac{-}{\mathcal{C}; \Gamma \vdash \varnothing \rhd \varnothing} \qquad \frac{\mathcal{C}; \Gamma \vdash N \rhd \Sigma \qquad \mathcal{C}; \Gamma \vdash N' \rhd \Sigma'}{\mathcal{C}; \Gamma \vdash N \mid N' \rhd \Sigma, \Sigma'} \quad \lfloor \text{N3/N4} \rfloor$$

$$\frac{\mathcal{C}; \Gamma, a : \mathcal{G} \vdash P \rhd y[\mathtt{i}] : \mathcal{G} \upharpoonright \mathtt{i}, \Delta}{\mathcal{C}; \Gamma, a : \mathcal{G} \vdash a[\mathtt{i}](y).P \rhd \Delta} \quad \lfloor \text{MACC} \rfloor$$

$$\frac{\forall i \in I, \qquad \mathcal{C} \wedge A_i; \Gamma \vdash P_i \rhd \Delta, k[\mathtt{q}] : \mathcal{L}_i \qquad \mathcal{C} \wedge A_i \supset (\mathcal{C} \,\mathtt{after}\, E_i)}{\mathcal{C}; \Gamma \vdash k[\mathtt{p}, \mathtt{q}]?\{l_i(x_i)\langle E_i\rangle.P_i\}_{i \in I} \rhd \Delta, k[\mathtt{q}] : \mathtt{p}?\{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{L}_i\}_{i \in I}} \quad \lfloor \text{BCH} \rfloor$$

$$\frac{\begin{array}{c}\forall i \in I \exists j \in J, \qquad l_i = l_j \qquad \mathcal{C} \wedge \underline{e_i}; \Gamma \vdash P_i[e_i'/x_i] \rhd \Delta, k[\mathtt{p}] : \mathcal{L}_j[e_i'/x_j] \\ \mathcal{C} \wedge e_i \supset (A_j \wedge (E_i = E_j) \wedge (\mathcal{C} \,\mathtt{after}\, E_j))[e_i'/x_i]\end{array}}{\mathcal{C}; \Gamma \vdash k[\mathtt{p}, \mathtt{q}]!\{e_i \mapsto l_i\langle e_i'\rangle(x_i)\langle E_i\rangle; P_i\}_{i \in I} \rhd \Delta, k[\mathtt{p}] : \mathtt{q}!\{l_j(x_j : U_j)\{A_j\}\langle E_j\rangle.\mathcal{L}_j\}_{j \in J}} \quad \lfloor \text{SEL} \rfloor$$

$$\frac{\mathcal{C}; \Gamma \vdash P_1 \rhd \Delta_1 \quad \mathcal{C}; \Gamma \vdash P_2 \rhd \Delta_2}{\mathcal{C}; \Gamma \vdash P_1 \mid P_2 \rhd \Delta_1, \Delta_2} \qquad \frac{\Delta \text{ end only}}{\mathcal{C}; \Gamma \vdash \mathbf{0} \rhd \Delta} \quad \lfloor \text{PAR/END} \rfloor$$

$$\frac{\mathcal{L}_1[e/x], \ldots, \mathcal{L}_n[e/x] \text{ well-asserted}}{\mathcal{C}; \Gamma, X : (x)\mathcal{L}_1 @ \mathtt{p}_1, \ldots, \mathcal{L}_n @ \mathtt{p}_n \vdash X\langle e\rangle \rhd s[\mathtt{p}_1] : \mathcal{L}_1[e/x], \ldots, s[\mathtt{p}_n] : \mathcal{L}_n[e/x]} \quad \lfloor \text{VAR} \rfloor$$

$$\frac{\mathcal{C}; \Gamma, X : (x)\mathcal{L}_1 @ \mathtt{p}_1, \ldots, \mathcal{L}_n @ \mathtt{p}_n \vdash P \rhd s[\mathtt{p}_1] : \mathcal{L}_1, \ldots, s[\mathtt{p}_n] : \mathcal{L}_n}{\mathcal{C}; \Gamma \vdash (\mu X(x).P)\langle e\rangle \rhd s[\mathtt{p}_1] : \mathcal{L}_1[e/x], \ldots, s[\mathtt{p}_n] : \mathcal{L}_n[e/x]} \quad \lfloor \text{REC} \rfloor$$

Fig. 4. Proof rules for networks (top) and proof rules for processes (bottom)

$\lfloor$N1$\rfloor$ decomposes the validation of a network into the validations of each located process against its corresponding specification Δ. The correspondence between principal and specification in checked by the clause $\mathtt{id}(\sigma_p) = \mathtt{id}(\sigma_a)$. Furthermore, local and virtual states must satisfy the principal invariant $\mathcal{I}(\sigma_p)$. P is then validated in the assertion environment extended (i.e., in conjunction with) the principal invariant.

$\lfloor$N2$\rfloor$ is the rule for refinement. This rule is useful to validate processes even if they do not match exactly a given assertion as long as they implement a behaviour that is 'more refined' than the one prescribed. Refinement is also necessary to prove completeness of these rules (Theorem 14). We use the following refinement relation between specifications: $(\Gamma', \Delta', \sigma') \ni (\Gamma, \Delta, \sigma)$ if $(\Gamma', \Delta', \sigma')$ specifies a more refined behaviour than (Γ, Δ, σ), in that it poses more restrictions on the output actions and poses less restrictions on the input actions. $\lfloor$N2$\rfloor$ allows to refine the assertion environment $\mathcal{C}$ by considering, in the premise, a weaker set of assumptions $\mathcal{C}'$.

$\lfloor$N3$\rfloor$ is for empty networks and $\lfloor$N4$\rfloor$ is for decomposing the validation of networks.

$\lfloor$MACC$\rfloor$ validates a session accept on a shared channel a as role $\mathtt{i}$ provided that a is in the domain of Γ, and that the continuation P is validated against the specification Δ extended with the new session $y[\mathtt{i}]$. In the (now active) session $y[\mathtt{i}]$, P must behave as $\Gamma(a)$ projected on role $\mathtt{i}$. The rule for session request is similar hence omitted.

$\lfloor$BCH$\rfloor$ validates the branching process. Δ must include an active session $k[\mathtt{q}]$ on session channel k for the receiver role $\mathtt{q}$. In the premise, the continuation for each branch i is required to be still valid in the assertion environment extended with A_i. In

the second clause of the premise, for each branch i the update E_i must not invalidate $\mathcal{C}$; this ensures that the update does not invalidate the principal invariant. The invariant is not mentioned explicitly (to keep the proof rules concise), but it is implied by $\mathcal{C}$. In fact, $\mathcal{C}$ is the conjunction of (1) the principal invariant (by ⌊N1⌋), (2) possibly some interaction predicates (by ⌊BCH⌋), and (3) possibly some boolean expressions (by ⌊SEL⌋). Since predicates (2), (3) and A_i do not contain free state variables[8], then E_i can only invalidate the principal invariant (1); on the other hand (2), (3) and A_i are necessary premises (i.e., $\mathcal{C} \wedge A_i$ before the implication) as they may constrain interaction variables used by E_i.

In ⌊SEL⌋ each branch i of the process must correspond to a branch j of the specification ($l_i = l_j$). The continuation must be validated in assertion environment $\mathcal{C}$ extended with the closure $\underline{e_i}$ of the condition of the branch e_i. The closure of boolean expression e_i is defined as the closure for predicates (see Remark 4). The clause at the bottom of the premise requires that, under the assumption $\mathcal{C} \wedge e_i$: (1) expression e_i' satisfies A_j, (2) assertion and process have the same effects/updates on the states, (3) update E_j does not invalidate the principal invariant. [9]

⌊PAR⌋ is similar to ⌊N2⌋ but for parallel processes. ⌊END⌋ validates the idle process provided that each active session in the specification Δ is of the form $y[\mathtt{p}] : \mathsf{end}$.

⌊VAR⌋ validates recursive call given that the active sessions in Δ correspond to the roles and local assertions associated to process variable X in Γ and that each $\mathcal{L}_i$ is still well-asserted when the recursion parameter is substituted with e. ⌊REC⌋ is the standard rule for recursion definition. The validation of recursive processes is handled in a similar way to [5]; it uses a refinement rule for processes, similar to ⌊N2⌋ and omitted for simplicity, and the fact that assertions are refined by their unfolding. See [4] for more details.

Example 9. Consider the located process $[P_\mathtt{S} \mid P_\mathtt{C}]\sigma_p$ from Example 8 that executes two parallel threads: one selling a ticket and the other one buying another kind of ticket from another principal (the other principal is not modelled here). We show the validation of $\mathsf{true}; \Gamma \vdash [P_\mathtt{S} \mid P_\mathtt{C}]\sigma_p \rhd [\varnothing]\sigma_a$ proceeding top-down using the rules in Figure 4.

The global specification $[\varnothing]\sigma_a$ is initially empty since there are no active sessions. The active sessions will be added upon session request/accept by $P_\mathtt{S}$ and $P_\mathtt{C}$. We assume $\sigma_p = \sigma_a = \{\mathtt{count} : \mathtt{int}, \mathtt{credit} : \mathtt{int}\} \mapsto \{10, 500\}$ and initially $\mathcal{C} = \mathsf{true}$.

We first apply ⌊N1⌋ with $\mathcal{I}(\sigma_p) = \mathtt{credit} \geqslant 0 \wedge 0 \leqslant \mathtt{count} \leqslant 1000$ from Example 8. For readability we will write $\mathcal{I}$ instead of $\mathcal{I}(\sigma_p)$ in this example. Note that $\mathcal{I}$ is satisfied by the local and virtual state. Next we apply rule ⌊PAR⌋ that decomposes the derivation of two threads for $P_\mathtt{S}$ and $P_\mathtt{C}$. We omit the illustration of the latter thread.

Below we illustrate the application of rule ⌊MACC⌋ and ⌊BCH⌋ to the former thread:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{I} \wedge \{\exists \mathtt{C}.\mathtt{credit}.\mathtt{C}.\mathtt{credit} \geqslant x\}; \Gamma \vdash P_\mathtt{S}' \rhd z[\mathtt{S}] : \mathcal{L}_\mathtt{S}' \\ \mathcal{I} \wedge \{\exists \mathtt{C}.\mathtt{credit}.\mathtt{C}.\mathtt{credit} \geqslant x\} \supset (\mathcal{I}\ \mathtt{after}\ \varnothing)\end{array}}{\mathcal{I}; \Gamma \vdash z[\mathtt{C},\mathtt{S}]?(x).P_\mathtt{S}' \rhd z[\mathtt{S}] : \mathtt{C}?(x : \mathtt{Nat})\{\exists \mathtt{C}.\mathtt{credit}.\mathtt{C}.\mathtt{credit} \geqslant x\}.\mathcal{L}_\mathtt{S}'}\ \lfloor \mathrm{BCH} \rfloor}{\mathcal{I}; \Gamma \vdash P_\mathtt{S} \rhd \varnothing}\ \lfloor \mathrm{MACC} \rfloor$$

[8] By history-sensitivity A_i does not include any free state variable.

[9] ⌊BCH⌋/⌊SEL⌋ can be extended to delegation adding the following clause for $U_i = \langle \mathcal{L} \rangle$: (⌊BCH⌋) $\mathcal{C} \wedge A_i; \Gamma \vdash P_i \rhd \Delta, k[\mathtt{q}] : \mathcal{L}_i, x_i : \mathcal{L}$, and (⌊SEL⌋) $\mathcal{C} \wedge \underline{e_i}; \Gamma \vdash P[e_i'/x_i] \rhd \Delta', k[\mathtt{p}] : \mathcal{L}_j[e_i'/x_j]$ with $\Delta = \Delta'', e_i : \mathcal{L}_i'$ and $\Delta' = \Delta''$.

For readability we will simplify $\mathcal{I} \wedge \{\exists \mathtt{C}.\mathtt{credit}.\mathtt{C}.\mathtt{credit} \geqslant x\}$ with the equivalent predicate $\mathcal{I}$. Next, by $\lfloor$SEL$\rfloor$, setting $e = \mathtt{count} < 1000 \wedge x \geqslant 10$, and $E_{ok} = \mathtt{count} := \mathtt{count} + 1,\ \mathtt{credit} := \mathtt{credit} + x$:

$$\frac{\mathcal{I} \wedge e \supset (\mathtt{count} < 1000 \wedge y = \mathtt{count} \wedge E_{ok} = E_{ok} \wedge \mathcal{I}\,\mathtt{after}\,E_{ok})[\mathtt{count}/y] \quad \mathcal{I}; \Gamma \vdash \mathbf{0} \rhd z[\mathtt{S}] : \mathsf{end} \qquad \mathcal{I} \wedge \neg e \supset \mathsf{true} \quad \mathcal{I}; \Gamma \vdash \mathbf{0} \rhd z[\mathtt{S}] : \mathsf{end}}{\begin{array}{l}\mathcal{I}; \Gamma \vdash z[\mathtt{S}, \mathtt{C}]!\{e \mapsto \mathsf{ok}\langle \mathtt{count}\rangle(y)\langle E_{ok}\rangle.\mathbf{0}, \neg e \mapsto \mathsf{ko}.\mathbf{0}\} \\ \quad \vdash z[\mathtt{S}] : \mathtt{C}!\{\mathsf{ok}(y : \mathtt{Nat})\{\mathtt{count} < 1000 \wedge y = \mathtt{count}\}\langle E_{ok}\rangle.\mathsf{end},\ \mathsf{ko}.\mathsf{end}\}\end{array}}$$

where each line in the premise refers to a branch (i.e., ok and ko). The most delicate clause is $\mathcal{I} \wedge e \supset (\mathtt{count} < 1000 \wedge y = \mathtt{count} \wedge E_{ok} = E_{ok} \wedge \mathcal{I}\,\mathtt{after}\,E_{ok})[\mathtt{count}/y]$ which requires: (1) the interaction predicate to be satisfied under the current assumptions, and in fact $(\mathtt{count} < 1000 \wedge y = \mathtt{count})[\mathtt{count}/y]$ is implied by e, (2) the updates to be consistent, and in fact trivially $E_{ok} = E_{ok}$, and (3) the update to not invalidate the invariant, and in fact $\mathtt{credit} + x \geqslant 0 \wedge 0 \leqslant \mathtt{count} + 1 \leqslant 1000$ is true under the assumptions $\mathtt{credit} \geqslant 0$, $x \geqslant 10$ and $0 \leqslant \mathtt{count}$. Finally we apply $\lfloor$END$\rfloor$ to the second premise of each branch.

The effectiveness of the proof rules depends on the logic chosen for the predicates, which depends on the application scenario. An example which fits these criteria is the Presburger arithmetic, which is often sufficiently expressive: practical uses of multiplication are encodable [17], and formulae with quantifiers may be calculated efficiently [20, 22].

Proposition 10. *The proof of $\mathcal{C}; \Gamma \vdash N \rhd \Sigma$ is decidable if predicate evaluation is decidable.*

5 Soundness and completeness of the validation rules

We define a labelled transition relation for specifications $\langle \Gamma, \Sigma \rangle$ using the same labels as for networks. The main difference with the rules for networks is that predicates must be satisfied for the transition to occur. We illustrate below the most remarkable rules (see [4] for the other rules). The rule for session request:

$$\langle (a : \mathcal{G}, \Gamma); [\Delta]\sigma \rangle \xrightarrow{\overline{a}[\mathtt{n}]\langle s \rangle} \langle (a : \mathcal{G}, \Gamma); [\Delta, s[1] : \mathcal{G} \upharpoonright 1]\sigma \rangle$$

extends Δ with the new session, given that $a : \mathcal{G}$ in Γ and the current state satisfies assertion invariant A. The rule for session accept is dual. The rule for selection/send:

$$\frac{j \in I \quad \sigma \models A_j[\mathtt{n}/x_j] \quad \sigma' = \sigma\,\mathtt{after}\,E_j[\mathtt{n}/x_j]}{\langle \Gamma; [\Delta, s[\mathtt{p}] : \mathtt{q}!\{l_i(x_i : U_i)\{A_i\}\langle E_i\rangle.\mathcal{L}_i\}_{i \in I}]\sigma \rangle \xrightarrow{s[\mathtt{p},\mathtt{q}]!l_j\langle \mathtt{n}\rangle} \langle \Gamma; [\Delta, s[\mathtt{p}] : \mathcal{L}_j[\mathtt{n}/x_j]]\sigma' \rangle}$$

moves to the continuation $\mathcal{L}_j$ of the selected branch with the updated state σ', given that the sent value $\mathtt{n}$ satisfies predicate A_j for branch j in the current state σ.

Semantic conformance is defined using *conditional simulation* [5] to relate networks N to specifications $\langle \Gamma; \Sigma \rangle$.

Definition 11 (Conditional Simulation). A binary relation $\mathcal{R}$ over N and $\langle \Gamma; \Sigma \rangle$ is a *conditional simulation* if, for each $(N, \langle \Gamma; \Sigma \rangle) \in \mathcal{R}$, if $N \xrightarrow{\ell} N'$ with ℓ being:
(1) a branching then $\langle \Gamma; \Sigma \rangle$ is capable to move at the subject of ℓ, and if $\langle \Gamma; \Sigma \rangle \xrightarrow{\ell}$

$\langle \Gamma; \Sigma' \rangle$ then $(N', \langle \Gamma; \Sigma' \rangle) \in \mathcal{R}$;
(2) a select, session request/accept, τ then $\langle \Gamma; \Sigma \rangle \xrightarrow{\ell} \langle \Gamma; \Sigma' \rangle$ and $(N', \langle \Gamma; \Sigma' \rangle) \in \mathcal{R}$.
We write $N \lesssim \langle \Gamma; \Sigma \rangle$ if there exists a conditional simulation $\mathcal{R}$ s.t. $(N, \langle \Gamma; \Sigma \rangle) \in \mathcal{R}$.

Conditional simulation is like standard simulation for all types of actions except for branching, for which it requires N to be simulated only for legal values/labels (i.e., a process must conform to a given specification as long as its environment does so).

Definition 12 (Satisfaction). N satisfies Σ in Γ and $\mathcal{C}$, written $\mathcal{C}; \Gamma \models N \rhd \Sigma$, if for all closing substitutions $\tilde{\sigma}$ over N and Σ respecting Γ and $\mathcal{C}$, $N\tilde{\sigma} \lesssim \langle \Gamma; \Sigma\tilde{\sigma} \rangle$.

We write $\Gamma \models N \rhd \Sigma$ when $\mathcal{C}$ is true (e.g., for initial networks). Soundness and completeness for initial networks are stated below.

Theorem 13 (Soundness of Proof Rules). *Let N be an initial network. Then $\Gamma \vdash N \rhd \Sigma$ implies $\Gamma \models N \rhd \Sigma$.*

Theorem 14 (Completeness of Proof Rules). *Let $N \equiv \prod_{i \in I} [P_i]\sigma_{pi}$ be an initial network and $\Sigma = \prod_{i \in I} [\Delta_i]\sigma_{ai}$ be a specification. Assume that for all $i \in I$:* (1) $\mathtt{id}(\sigma_{pi}) = \mathtt{id}(\sigma_{ai})$, (2) $dom(\sigma_{pi}) = dom(\sigma_{ai})$, *and* (3) $\mathcal{I}(\sigma_{pi})$ *equivalent to* true. *If $\Gamma \models N \rhd \Sigma$ then $\Gamma \vdash N \rhd \Sigma$.*

(1-2) are for symmetry between N and Σ. (3) is necessary since the principals in N can make updates that differ from those made by the corresponding specifications in Σ; this may not compromise the observable behaviour of N with respect to Σ, but N may invalidate some principal invariant which would make the thesis false.

6 Related work and further topics

The preceding integrations of session types with logical constraints include [13], based on concurrent constraints ensuring bi-linear usage of channels, and [5], based on logical annotations on interactions, do not treat stateful properties. The combination of types and logical assertions referring to local state newly proposed in this paper enable fine-grained specifications and validation, which are not possible in [5, 13].

The expressiveness of the session type-based analyses has been greatly extended these past few years. On one side, the conversation calculus [8], contracts [11] and dynamic multirole session types [14] have opened the way to the modelling of protocols complex in their shapes, by describing more accurately how sessions can be joined or left, who is allowed participate. On the other side, works such as [5, 9] improved the way interactions inside a session are described: in [5], an assertion framework ensures logical properties on the communicated values, in [9], a security analysis guarantees that the coherence of the information flow is preserved. Our work improves the session type analyses in both directions: by proposing a division of the process being tested into separate principals that can join one or several sessions independently when conditions are matched and manage their own state, and by giving a description, inside each session, of the internal state of each participant and the property it should satisfy. A recent work [12] examines conditions to ensure that a stateful specification is robust w.r.t.

asynchronous communications. Our work provides a complete proof system ensuring soundness for processes, whereas [12] only addresses properties of types.

The refinement types for channels (e.g. [3]) specify value dependency with logical constraints. For example, one might write $?(x:\mathsf{int}, !\{y : \mathsf{int} \mid y > x\})$ (using the notation from [16]). It specifies a dependency at a *single point* (channel). Our theory, based on multiparty sessions, can verify processes against a contract globally agreed by multiple distributed peers. [2] uses refinement types for channels to verify authentication in multiparty session protocols, but does not consider multi-session properties.

The work [7] investigates a relationship between a dual intuitionistic linear logic and binary session types, and shows that the former defines a proof system for a session calculus which can automatically characterise and guarantee a session fidelity and global progress. None of the above works treat either virtual states or logical specifications for interleaved multiparty sessions.

The use of Rely-Guarantee conditions or other related methods [19] instead of a single invariant does not increase the expressiveness of our system, but could ease proofs for parallel composition.

A future direction is to link between our static analysis and a dynamic monitor-based approach. Using our local specification as a monitor at each end-point, incoming and outgoing messages can be verified and filtered. We are currently working on this topic with [21] based on the logic developed in this paper.

References

1. L. Bettini et al. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
2. K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF*, pages 124–140, 2009.
3. K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *POPL*, pages 445–456, 2010.
4. L. Bocchi, R. Demangeon, and N. Yoshida. A multiparty multi-session logic (extended report). http://www.cs.le.ac.uk/people/lb148/statefulassertions.html.
5. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR*, volume 6269 of *LNCS*, pages 162–176, 2010.
6. L. Bocchi, J. Lange, and E. Tuosto. Three algorithms and a methodology for amending contracts for choreographies. *Scientific Annals of Computer Science*, 22(1):61–104, 2012.
7. L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR'10*, volume 6269 of *LNCS*, pages 222–236. Springer-Verlag, 2010.
8. L. Caires and H. T. Vieira. Conversation types. In *ESOP*, volume 5502 of *LNCS*, pages 285–300. Springer, 2009.
9. S. Capecchi, I. Castellani, and M. Dezani-Ciancaglini. Information flow safety in multiparty sessions. In *EXPRESS*, volume 64 of *EPTCS*, pages 16–30, 2011.
10. M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions in session types. In *CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.
11. G. Castagna and L. Padovani. Contracts for mobile processes. In *CONCUR 2009*, number 5710 in LNCS, pages 211–228, 2009.
12. T.-C. Chen and K. Honda. Specifying stateful asynchronous properties for distributed programs. (to appear in CONCUR), 2012.

13. M. Coppo and M. Dezani-Ciancaglini. Structured communications with concurrent constraints. In *TGC*, pages 104–125, 2008.
14. P.-M. Deniélou and N. Yoshida. Dynamic multirole session types. In *POPL*, pages 435–446, 2011.
15. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, August 1975.
16. T. Freeman and F. Pfenning. Refinement types for ML. *SIGPLAN Not.*, 26(6):268–277, 1991.
17. M. K. Ganai. Efficient decision procedure for bounded integer non-linear operations. In *HVC '08*, pages 68–83. LNCS, 2009.
18. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
19. C. Jones. Abstraction for concurrency. In *SEFM*, LNCS, 2012. to appear.
20. G. Nelson and D. C. Oppen. A simplifier based on efficient decision algorithms. In *POPL'78*, pages 141–150. ACM, 1978.
21. Ocean Observatories Initiative (OOI). `http://www.oceanleadership.org/programs-and-partnerships/ocean-observing/ooi/`.
22. W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, pages 4–13, New York, NY, USA, 1991. ACM.