Asynchronous Distributed Monitoring for Multiparty Session Enforcement

Tzu-Chun Chen[1], Laura Bocchi[2], Pierre-Malo Deniélou[3], Kohei Honda[1], and Nobuko Yoshida[3]

[1] Queen Mary, University of London
[2] University of Leicester
[3] Imperial College London

Abstract. We propose a formal model of runtime safety enforcement for large-scale, cross-language distributed applications with possibly untrusted endpoints. The underlying theory is based on multiparty session types with logical assertions (*MPSA*), an expressive protocol specification language that supports runtime validation through monitoring. Our method starts from global specifications based on *MPSA*s which the participants should obey. Distributed monitors use local specifications, projected from global specifications, to detect whether the interactions are well-behaved, and take appropriate actions, such as suppressing illegal messages. We illustrate the design of our model with examples from real-world distributed applications. We prove monitor transparency, communication conformance, and global session fidelity in the presence of possibly unsafe endpoints.

1 Introduction

Communication among distributed components is becoming the norm for building large-scale software, for example in the backend of web services, financial protocols, enterprise applications and cyberinfrastructures. This change is sustained by an accelerating infrastructural support for portable distributed components through technologies such as clouds, messaging middleware and distributed stores. While distribution leads to such virtues as scalability, sharing and resilience [10,22], guaranteeing safety poses new technical challenges. First, endpoints for a distributed application are often managed under multiple administrative domains, making it hard to enforce the use of verified code. Thus, even in a single application, *safe and unsafe components can co-exist*. Secondly, many non-trivial correctness properties of distributed programs rely on reciprocal assurance through cooperating endpoints (for example, a sender multi-casts a message with a type expected by each receiver). Hence a safety property needs be specified as a *global invariant* involving multiple peers. Thirdly, in spite of its global nature, scalable distributed assurance is only feasible through *local validation*: a centralised dynamic validation (validating all distributed interactions in one place) is clearly unrealistic in large-scale distributed systems.

Against these backgrounds, this paper introduces a theory of *runtime* verification for distributed programs, based on distributed endpoint monitoring in which non-trivial global safety assurance results from local runtime verification and enforcement of possibly untrusted endpoints, rather than from static checking. We stipulate that an external

R. Bruni and V. Sassone (Eds.): TGC 2011, LNCS 7173, pp. 25–45, 2012.

monitor is associated with each endpoint participating in a distributed infrastructure (in actual implementations, one or more logical monitors may be realised by (a cluster of) one or more physical monitors). A monitor associated with an endpoint acts for that endpoint as its unique entry point to the infrastructure, and guarantees that its interactions with its environment, which are *globally observable*, never violates a given global specification. The framework hinges on the linkage between local validation and global correctness, and is characterised by asynchrony: each monitor can verify the behaviour of local process only by observing the outgoing or incoming messages and validating that they conform to a given local specification.

For linking the global invariants to local validation, we use the preceding work on *multiparty session types* (*MPST*s) [5,17], which provides a formally founded approach to the local validation of globally specified protocols, assuring communication safety, protocol fidelity and progress. We use its logical extension, called *multiparty session assertions* (*MPSA*s) [3], which further allow, by extending *MPST*s with logical formulae, fine-grained specification of interactional behaviour including message contents, choices of conversation paths and recursion invariants. By projecting a *global assertion* (global protocol specification) onto each *endpoint*, we obtain a local assertion for each endpoint. By all endpoints adhering to their respective local specifications, their interactions satisfy global correctness.

Our theory offers a formal framework to semantically link local behaviour of processes to their global behaviour and to global invariants, through the introduction of a notion of *global observables*. Since, in distributed processes, the sending events and the receiving events are decoupled, and because external monitors can only observe asynchronously exchanged messages, the semantic account of global invariants (hence correctness of runtime verification) should take into account temporary discrepancies of the global view. When a sender sends a message correctly, its local view is updated; however, as the receiver has not yet received the message, we cannot update neither the global view nor the receiver's local view. In other words, to prove the correctness of distributed runtime verification, we cannot simply use the projection from global invariants to endpoints: we need to take into account the time lag between the sending and receiving events. Asynchrony also poses a challenge in the treatment of out-of-order asynchronous message monitoring, which we capture through type-level permutations of actions.

This paper offers an overview of a theory of monitored networks illustrated through many examples. Its main contributions may be summarised as follows.

Contributions. The main contributions of the present paper include:

- A model of distributed monitoring featuring the following elements: (1) endpoint code is possibly ill-behaved; (2) global assertions [3] enable concise global specification of application-level multiparty protocols; and (3) conformance to stipulated global protocols is guaranteed at runtime through local monitoring.
- A multiparty session π-calculus with distributed external monitors, with a new capability passing primitive for fine-grained control of distributed session initialisation. The calculus presents an exact semantic account of monitors' behaviour, offering a foundation of architectural realisations of the proposed framework, with an efficient monitoring mechanism.

– An overview of the fundamental properties of monitored networks, including local and global safety, the latter built on a novel global observational framework. We discuss how local communication conformance leads to global conformance and session fidelity, as well as estalbishing the local and global monitor transparency.

As far as we know, the formal behavioural assurance of global properties for distributed applications with unsafe endpoints against non-trivial logical specifications, built on a rigorous semantic basis, is new. The general ideas of the proposed framework are illustrated in § 2 through a concrete example. An asynchronous distributed calculus with multiparty session primitives and the syntax of endpoint monitors are introduced in § 3. The monitored network and global observable environment (which constructs the global observables) are introduced in § 4. In § 4, we propose a formal framework for *runtime* monitoring, and describe how an endpoint monitor is capable to guide and protect local processes to exactly obey the global specification, and thus assure the safety and correctness of the whole monitored network during runtime. § 5 outlines the resulting properties and theorems, such as local and global safety, local and global transparency, and session fidelity. For further comparisons, see § 6. For full definitions and descriptions of the running example please refer to the online full version [20].

2 Basic Ideas through Examples

2.1 Assumptions and Background

As we discussed in Introduction, we assume that endpoint components of distributed applications may reside in geographically disparate and heterogeneous administrative domains, so that we cannot expect that all programs are pre-verified. This necessitates the use of *runtime* verification and enforcement through trusted monitors. Monitors act as gateways for participating endpoint code: first an endpoint sends a message and other requests to a monitor; then if the monitor finds the message to be valid, it will route it to its designation. If the monitor finds the message to be invalid, it will treat it as an error. Below we illustrate the basic idea of this distributed runtime verification through a concrete use case from a project to build a global distributed infrastructure for ocean science, the Ocean Observatories Initiative (OOI) [23]. The OOI aims to build a distributed network of environmental observatories for ocean sciences, with persistent and interactive capabilities [7]. Its key element is a comprehensive cyberinfrastructure (OOI CI), which offers services to its users through interactions among distributed endpoints in geographically distributed OOI observatories, from seafloor instruments to buoys and on-shore research stations, communicating through a uniform messaging infrastructure. Its usecase scenarios focus on structured conversations among distributed endpoints which may be a thousand miles apart, running under different administrative domains with different degrees of trust. For example, some components of a distributed application may be scripting programs in users' browsers, some may be located in the endpoint cloud of an academic or corporate institution, and some others may be infrastructural components residing in one of the central clouds of the OOI CI.

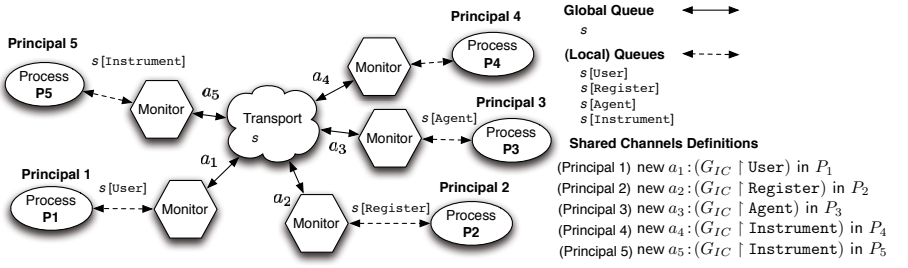

Fig. 1. Monitoring architecture

Because each distributed service in the OOI CI is realised by structured conversations among distributed components, it is essential for its development that protocols for interactions among the participating components are specified with clear semantics. We call these protocols, *application-level protocols*. For example, a scientist may wish to use a remote instrument, say a seabed camera, after being authorised by an agent: the interactions among these three will form a specific application-level protocol. In the OOI CI, the design choice has been made so that these application-level protocols are to be uniformly specified by a common protocol description language. Safety of interactions is validated against these stipulated protocol descriptions.

Since we *cannot* expect either each endpoint code or its environment to be (wholly) trusted, a primary way to ensure a global invariant for such an interactional application is through trusted monitors which enforce correct behaviour for participating programs, a design being considered in the OOI CI. Such monitors need in general to be *external* to these participants, i.e. they are not embedded in their code, because, as already discussed, we may not be able to trust an administrative domain where an endpoint code resides (an endpoint may as well be equipped with its internal monitor). These monitors will check and verify the incoming/outgoing messages to/from distributed components, so that the correctness of global interactions are ensured. Logically, we can stipulate that there is one monitor for each principal, guarding their behaviour.

A particular application-level protocol in the OOI CI is the *instrument command* we shall treat below. It allows a user to perform operations on a remote instrument. This protocol is specified as a global assertion G_{IC}, described later in this section (the formal definition is in § 3.2). G_{IC} models a session which involves roles `User`, `Register`, `Agent` and `Instrument`. Given a global assertion G, we denote its projection on role `p` as $G \upharpoonright \texttt{p}$, giving an endpoint assertion which describes the protocol from the perspective of a specific role. We call a participant in one or more distributed applications in the OOI CI, *principal*. A principal executes its local process and interacts with other principals through one or more applications. Its incoming and outgoing messages are validated by a monitor, which is part of the infrastructure. Figure 1 presents a monitored network involving five principals.

We now illustrate, using the instrument command example, some of the features of our formalism, including how the initialisation of conversation is done and how the associated protocol is specified.

2.2 Session Establishment through Distributed Invitations

For a monitor to listen to and check the communications of a principal, it should start by observing the initiation phase of conversations, called *session establishment*. Session establishments are done in two steps:

1. A principal initiates a session by creating a *session channel* for a specific application-level protocol (e.g., G_{IC}); and
2. The principal sends, through shared channels, invitations to other principals to participate in the session in a specified role.

The initiation of a conversation (1) equips the initiator with its *endpoint capabilities*, one for each role, each of which can be passed to another principal (who may in turn pass it to yet another principal). Thus the binding of principals to endpoint roles is incrementally established through capability passing. This fine-grained scheme can represent many real-world examples [6]. Further, the operation can be seen as a asynchronous linear capability passing system, leading to a clear monitor semantics.

As an example, consider the following process P_1 of $Principal_1$, from Figure 1, which creates a session s with specification G_{IC} and sends invitations to others.

$$\begin{aligned} P_1 &= \mathsf{new}\, s : G_{IC}\ \mathsf{in}\ (P_{JOIN} \mid P_{INV}) \\ P_{JOIN} &= \mathsf{join}\ s[\mathtt{User}]; P_{\mathtt{user}} \\ P_{INV} &= \overline{a_2}\langle s[\mathtt{Register}] : G_{IC} \upharpoonright \mathtt{Register}\rangle; \overline{a_3}\langle s[\mathtt{Agent}] : G_{IC} \upharpoonright \mathtt{Agent}\rangle; \\ &\quad\ \overline{a_4}\langle s[\mathtt{Instrument}] : G_{IC} \upharpoonright \mathtt{Instrument}\rangle \end{aligned}$$

where the capabilities $s[\mathtt{User}]$, $s[\mathtt{Register}]$, $s[\mathtt{Agent}]$ and $s[\mathtt{Instrument}]$, are either *activated* (i.e. the process has joined the session) or forwarded (i.e. used for inviting others). E.g., P_1 joins the session as `User` (P_{JOIN} above), and sends invitations to the other roles (P_{INV} above). Thus $Principal_1$ is inviting other four principals in Figure 1. *Shared channels* $(a_1, a_2, \ldots)$ determine which invitations each process is entitled to receive, e.g. only $Principal_4$ and $Principal_5$ can accept (by any other process) an invitation $s[\mathtt{Instrument}]$, through a_4 and a_5, respectively.

Below we show $Principal_4$, which receives the capability $s[\mathtt{Instrument}]$ and passes it to $Principal_5$ who joins the session.

$$P_4 = a_4(y_4[\mathtt{Instrument}] : G_{IC} \upharpoonright \mathtt{Instrument}).\overline{a_5}\langle y_4[\mathtt{Instrument}] : G_{IC} \upharpoonright \mathtt{Instrument}\rangle$$
$$P_5 = a_5(y_5[\mathtt{Instrument}] : G_{IC} \upharpoonright \mathtt{Instrument}).\mathsf{join}\ y_5[\mathtt{Instrument}]; P_{\mathtt{Instrument}}$$

Monitors observe the exchange of capabilities between their endpoint and the network. Once a process joins a session, the corresponding monitor is activated and enforces conformance to the conversation scenario prescribed for that session/role for the subsequent interactions. The asynchronous message in transit between a process (as a role in a session) and its monitor is represented as a message in a *local queue*: messages in transit among monitors for a session is called *global queue*. In Figure 1, $s[\mathtt{User}]$, $s[\mathtt{Register}]$, etc., represent local queues, while s is a global queue.

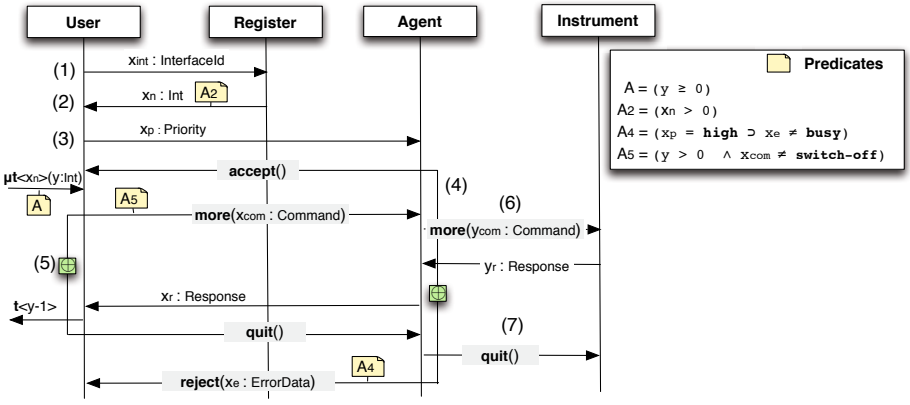

Fig. 2. Illustration of the global assertion for 'OOI Instrument Command'

2.3 Specifying Application-Level Protocols

Monitors dynamically learn about which protocols to enforce during session establishment. Application-level protocols are specified as *MPSA*s [3]. A global *MPSA*, sometimes called global assertion, is an abstract description of the interaction steps taken by the roles in the session. Basic interaction steps are asynchronous message exchanges. Branching and recursion describe potential choices and repeated interactions. Each message exchange/branching is annotated with a predicate specifying a constraint on the message value or the choice of a branch. Recursion is annotated with an invariant.

Figure 2 gives an illustration of global assertion G_{IC} as a message sequence chart, in which `User` performs one or more commands on a remote `Instrument`. `Register` is used to retrieve information on the instrument's usage, e.g., to determine the maximum number of commands allowed in the current session depending on the system load. `Agent` interfaces the communications with the actual `Instrument`.

Full arrows represent interactions where one party sends a branch label and a message value (or just one of the two) to another party. The labels carry information on the branch to follow. Arrows linked by $\oplus$ represent alternative branches.

The conversation proceeds as follows:

(1) `User` sends `Register` a message x_{int} of type InterfaceId.
(2) `Register` replies with an integer x_n which determines the number of commands that `User` will be allowed to perform on the instrument. The predicate annotating this interaction specifies an obligation for `Register` to send a value for x_n satisfying $x_n > 0$; dually `User` can rely on this fact.
(3) `User` sends `Agent` a priority x_p (e.g., *low*, *high*).
(4) `Agent` sends `User` a label which is either `accept` or `reject`. In case of `reject`, `Agent` sends also an error message x_e of type errData and the protocol terminates. The predicate for this branch ensures that a request will not be rejected due to the fact that the instrument is busy if the priority is high. In case of `accept` the protocol continues with a recursion $\mu \mathsf{t}\langle x_n \rangle (y : \mathsf{Int})$ where y is a parameter initialised to x_n,

$y \geq 0$ is an invariant and y is used to enforce `User` to perform at most x_n commands on the instrument.

(5) `User` selects either branch more and sends a new command to `Agent`, or quit and terminates. The predicate $y > 0 \wedge x_{com} \neq$ switch-off is a guard to the branch more: a new command can be sent only if `User` has not performed already x_n commands in this session and anyway the command must not ask to switch off the instrument.

(6, 7) Finally, either the command (6) or the quit notification (7) is forwarded by `Agent` to `Instrument`. In the former case, `Instrument` responds to `Agent` (who forwards the message to `User`).

2.4 Processes for Instrument Command

In the following we present the processes for instrument command used in our later discussions. $Principal_1$ in Figure 1 runs P_1 in § 2.2. P_1 refers to $P_{\texttt{user}}$ which is given as:

$$\begin{aligned} P_{\texttt{user}} &= s[\texttt{User},\texttt{Register}]!\langle v_{int}\rangle; s[\texttt{Register},\texttt{User}]?(x_n).\\ &\quad s[\texttt{User},\texttt{Agent}]!\langle \mathsf{High}\rangle; s[\texttt{Agent},\texttt{User}]?\{\texttt{accept}().P^u_{acc},\ \texttt{reject}(x_e)\}\\ P^u_{acc} &= \mu X\langle x_n\rangle(y).\mathsf{if}\ needmore() \wedge y > 0\ \mathsf{then}\ s[\texttt{User},\texttt{Agent}]!\texttt{more}\langle next()\rangle;\\ &\quad s[\texttt{Agent},\texttt{User}]?(x_r).\mathsf{t}\langle y-1\rangle; \mathsf{else}\ s[\texttt{User},\texttt{Agent}]!\texttt{quit}\langle\rangle \end{aligned}$$

where $needmore()$ and $next()$ are functions local to $Principal_1$: $needmore()$ returns a boolean (i.e., whether more commands are needed), and $next()$ returns the next command. $Principal_2$ in Figure 1 uses a_2 to receive an invitation, then joins the session as `Register`. We assume $Principal_2$ returns always 10 allowing all participants to perform at most 10 commands on the instrument. Note any positive value for x_n would satisfy the predicate $x_n > 0$ in **(2)** of §2.2.

$$\begin{aligned} P_2 &= a_2(y_2[\texttt{Register}] : G_{IC} \upharpoonright \texttt{Register}).\mathsf{join}\ y_2[\texttt{Register}]; P_{register}\\ P_{register} &= y_2[\texttt{User},\texttt{Register}]?(x_{int}).y_2[\texttt{Register},\texttt{User}]!\langle 10\rangle \end{aligned}$$

The next process is for Agent ($Principal_3$ in Figure 1), which, through channel a_3, receives an invitation and joins the session as `Agent`. We assume $Principal_3$ relies on a local function $error()$ returning an error data. Noticeably, the agent simply forwards the command and the response between the user and the instrument.

$$\begin{aligned} P_3 &= a_3(y_3[\texttt{Agent}] : G_{IC} \upharpoonright \texttt{Agent}).\mathsf{join}\ y_3[\texttt{Agent}]; P_{agent}\\ P_{agent} &= y_3[\texttt{User},\texttt{Agent}]?(x_p).\mathsf{if}\ x_p = \mathsf{high}\ \mathsf{then}\ y_3[\texttt{Agent},\texttt{User}]!\texttt{accept}\langle\rangle.P^a_{acc},\\ &\quad \mathsf{else}\ y_3[\texttt{Agent},\texttt{User}]!\texttt{reject}\langle error()\rangle\\ P^a_{acc} &= \mu\langle x_n\rangle X(y).y_3[\texttt{User},\texttt{Agent}]?\{\texttt{more}(x_{com}).P^a_{com},\\ &\quad \texttt{quit}().y_3[\texttt{Agent},\texttt{Instrument}]!\texttt{quit}\langle\rangle\}\\ P^a_{com} &= y_3[\texttt{Agent},\texttt{Instrument}]!\texttt{more}\langle x_{com}\rangle; y_3[\texttt{Instrument},\texttt{Agent}]?(y_r).\\ &\quad y_3[\texttt{Agent},\texttt{User}]!\langle y_r\rangle.\mathsf{t}\langle y-1\rangle \end{aligned}$$

Finally two instruments, $Principal_4$ and $Principal_5$ in Figure 1, are given below. The first, $Principal_4$, forwards the invitation to $Principal_5$ and terminates. The second, $Principal_5$, joins the session and relies on a local function *response* that takes a command and returns a response.

$u ::= a,b \mid x,y$		shared channel	$v ::= a \mid \mathsf{true} \mid \mathsf{false} \mid \mathtt{n}$		value
$k ::= s,s' \mid x,y$		sessions	$e ::= v \mid x \mid e+e' \mid e \wedge e' \mid \cdots$		expression
$P ::= \overline{u}\langle k[\mathtt{p}]:T\rangle;P$		request		$\mid k[\mathtt{p}_1,\mathtt{p}_2]!l\langle e\rangle;P$	selection
	$\mid u(y[\mathtt{p}]:T).P$	accept		$\mid k[\mathtt{p}_1,\mathtt{p}_2]?\{l_i(x_i).P_i\}_{i\in I}$	branching
	$\mid \mathsf{new}\, s:G\ \mathsf{in}\ P$	session creation		$\mid P \mid Q$	parallel
	$\mid \mathsf{new}\, a:T[\mathtt{p}]\ \mathsf{in}\ P$	name creation		$\mid \mathbf{0}$	inact
	$\mid \mathsf{join}\ s[\mathtt{p}];P$	join		$\mid \mu\mathsf{t}\langle e\rangle(x).P \quad \mid \mathsf{t}\langle e\rangle$	recursion
	$\mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional		$\mid P_{rt}$	runtime process
$P_{rt} ::= (\nu\, s:G)P \mid (\nu\, a:T[\mathtt{p}])P$		hiding	$h ::= \emptyset \mid \langle \mathtt{p},\mathtt{q},l\langle v\rangle\rangle \cdot h$		message queue
	$\mid \overline{a}\langle s[\mathtt{p}] : T\rangle$	invitation			
	$\mid s[\mathtt{p}] : h$	queue			

Fig. 3. Syntax of processes

$$
\begin{aligned}
P_4 &= a_4(y_4[\mathtt{Instrument}]:G_{IC} \upharpoonright \mathtt{Instrument}).\\
&\quad\ \overline{a_5}\langle y_4[\mathtt{Instrument}]:G_{IC} \upharpoonright \mathtt{Instrument}\rangle\\
P_5 &= a_5(y_5[\mathtt{Instrument}]:G_{IC} \upharpoonright \mathtt{Instrument}).\mathsf{join}\ y_5[\mathtt{Instrument}];P_{inst}\\
P_{inst} &= \mu\mathsf{t}.y_5[\mathtt{Agent},\mathtt{Instrument}]?\{\\
&\quad\ \mathtt{more}(y_{com}).y_5[\mathtt{Instrument},\mathtt{Agent}]!\langle response(y_{com})\rangle.\mathsf{t},\\
&\quad\ \mathtt{quit}()\}
\end{aligned}
$$

This concludes the introduction of processes for the Instrument Command use case.

3 A Calculus for Distributed Monitoring

In this section and the next, we introduce a calculus of distributed monitored processes. The syntax is divided into three parts:

(§ **3.1**) local untyped processes P, with a fine-grained distributed session initiation primitive called *invitation*;
(§ **3.2**) local monitors $\mathscr{M}$ which check the correctness of the incoming and outgoing messages (w.r.t. a set of endpoint assertions) and drop the wrong ones; and
(§ **4.1**) distributed networks N which consist of one or more monitored local processes $\mathscr{M}[P]$ and (global) queues containing pending messages sent by one monitored process but not yet received by another.

3.1 Multiparty Session π-calculus with Distributed Initialisation

This subsection presents the syntax of local processes, extending [5] with fine-grained primitives for session creation and invitation.

Invitations are exchanged through *shared channels* $(a,b,\ldots)$. The session interactions occur through *session channels* $(s,s',\ldots)$ whose names identify session execution

instances. Shared channel identifiers ($u, u', \ldots$) denote shared channels or variables; session identifiers ($k, k', \ldots$) denote session channels or variables. We let $\mathtt{p}, \mathtt{q}, \ldots$ range over *participant roles*, given as finite natural numbers. We use x, y for *variables*; v, v' for *values*; and e, e' for natural or boolean expressions.

We let P, P' denote processes. The *request* process asynchronously invites, through a shared channel u, another process to play $\mathtt{p}$ in a session (channel) k and continues as P (T denotes an endpoint assertion, defined later). The *accept* process receives a session invitation for role $\mathtt{p}$ and, after instantiating y with the received session channel, behaves as $\mathtt{p}$ (following endpoint assertion T). Process $\mathsf{new}\, s : G\ \mathsf{in}\ P$ creates a fresh session s whose protocol obeys a global assertion G, and behaves as P. Similarly for the shared name creation. By *join*, a process joins a session s as $\mathtt{p}$, creating a fresh local queue $s[\mathtt{p}]$.

The *selection* and *branching* represent communications through an established session k. We make the sender $\mathtt{p}_1$ and receiver $\mathtt{p}_2$ explicit by the notation $[\mathtt{p}_1, \mathtt{p}_2]$. The selection sends, in a session k, an expression e with label l from $\mathtt{p}_1$ to $\mathtt{p}_2$. Dually the branching is ready to receive each label l_i, and behaves as $P_i\{e/x\}$. Conditional, parallel composition and inaction are standard. The *recursion* $\mu \mathsf{t}\langle e \rangle(x).P$ defines t as P with recursion parameter x which is initialised to e; $\mathsf{t}\langle e \rangle$ is the corresponding recursion call.

Once a session is started, we use the runtime syntax P_{rt} (not accessible to the programmer) which includes hiding, asynchronous invitation messages and the queues representing messages in transit between a process and its monitor for a session instance.

For brevity, we often write $k[\mathtt{p}_1, \mathtt{p}_2]?l(x).P$ or $k[\mathtt{p}_1, \mathtt{p}_2]?(x).P$ for a single branch, and omit trailing $\mathbf{0}$ and type annotations. We call *initial* a process which does not contain free variables and runtime syntax.

Figure 4 lists the LTS for processes. Each output is performed in two steps: (1) a local action spawns the message (i.e., invitations remain local and linear messages are inserted in a local queue $s[\mathtt{p}]$) and (2) a visible action sends out the message. Inputs are dual. Whereas local actions are always allowed by monitors, visible actions have to be checked (as it will be clear in § 4.1). The first five rules are for local actions: $\lfloor$ARQ,AAC$\rfloor$ respectively spawn and receive an invitation message through a shared channel; $\lfloor$SEL$\rfloor$ puts a message in a local queue for a session s, after evaluating e to a value v; $\lfloor$BRA$\rfloor$ gets a message from a local queue with label l_j, so that the j-th process P_j receives a value v; $\lfloor$IF$\rfloor$ is standard. The rest models visible actions: $\lfloor$NEW,NEWS$\rfloor$ are for shared channel and session channel creation (it is through these actions that monitors learn about the session specifications to be enforced), $\lfloor$JOIN$\rfloor$ is for session joining using capability $s[\mathtt{p}]$ and creates the corresponding local queue, $\lfloor$OUT$\rfloor$ sends out an invitation, $\lfloor$OUT$\rfloor$ (resp. $\lfloor$IN$\rfloor$) sends out (resp. receives) a message from (resp. into) a local queue.

3.2 Distributed Monitors

To every process is associated a dedicated monitor that manages its interactions with the network (hence with other peers) by inspecting outgoing and incoming messages: the monitor protects the endpoint from bad messages from the environment and the environment from those from the endpoint. The syntax of monitors is presented in Figure 5.

Note that a monitor $\mathscr{M}$ should check two kinds of messages: (1) capability exchanges for invitations via shared channels, and (2) session messages via session channels. Thus

$$\lfloor \textsc{Arq,Aac} \rfloor \quad \overline{a}\langle s[\mathtt{p}]\rangle; P \xrightarrow{\tau} \overline{a}\langle s[\mathtt{p}]\rangle \mid P \qquad \overline{a}\langle s[\mathtt{p}]\rangle \mid a(y[\mathtt{p}]).Q \xrightarrow{\tau} Q\{s/y\}$$

$$\lfloor \textsc{Sel} \rfloor \quad s[\mathtt{p}_1,\mathtt{p}_2]!l\langle e\rangle; P \mid s[\mathtt{p}_1] : h \xrightarrow{\tau} P \mid s[\mathtt{p}_1] : h \cdot \langle \mathtt{p}_1,\mathtt{p}_2,l\langle v\rangle\rangle \qquad (e \downarrow v)$$

$$\lfloor \textsc{Bra} \rfloor \quad s[\mathtt{p}_1,\mathtt{p}_2]?\{l_i(x_i).P_i\}_{i\in I} \mid s[\mathtt{p}_2] : \langle \mathtt{p}_1,\mathtt{p}_2,l_j\langle v\rangle\rangle \cdot h \xrightarrow{\tau} P_j\{v/x_j\} \mid s[\mathtt{p}_2] : h$$

$$\lfloor \textsc{If} \rfloor \quad \mathsf{if\ true\ then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} P \qquad \mathsf{if\ false\ then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} Q$$

$$\lfloor \textsc{New,News} \rfloor \quad \mathsf{new}\, a{:}T[\mathtt{p}]\ \mathsf{in}\ P \xrightarrow{\mathsf{new}\ a:T[\mathtt{p}]} P \qquad \mathsf{new}\, s{:}G\ \mathsf{in}\ P \xrightarrow{\mathsf{new}\ s:G} P$$

$$\lfloor \textsc{Join,Out} \rfloor \quad \mathsf{join}\ s[\mathtt{p}]; P \xrightarrow{\mathsf{join}(s[\mathtt{p}])} P \mid s[\mathtt{p}] : \emptyset \qquad s[\mathtt{p}_1] : \langle \mathtt{p}_1,\mathtt{p}_2,l\langle v\rangle\rangle \cdot h \xrightarrow{s[\mathtt{p}_1,\mathtt{p}_2]!l\langle v\rangle} s[\mathtt{p}_1] : h$$

$$\lfloor \textsc{Req,In} \rfloor \quad \overline{a}\langle s[\mathtt{p}] \ : \ T\rangle \xrightarrow{\overline{a}\langle s[\mathtt{p}] \,:\, T\rangle} \mathbf{0} \qquad s[\mathtt{p}_2] : h \xrightarrow{s[\mathtt{p}_1,\mathtt{p}_2]?l\langle v\rangle} s[\mathtt{p}_2] : h \cdot \langle \mathtt{p}_1,\mathtt{p}_2,l\langle v\rangle\rangle$$

We omit standard context/structure congruence rules. Accept $\lfloor \textsc{Acc} \rfloor$ is defined as a dual of $\lfloor \textsc{Req} \rfloor$.

Fig. 4. Labelled transition system for processes

$$\begin{array}{lllll}
\mathscr{M} ::= \Gamma, \Delta & \text{monitor} & \Gamma ::= \emptyset \mid \Gamma, a : T[\mathtt{p}] & \text{shared env} \\
\Delta ::= \emptyset \mid \Delta, s[\mathtt{p}] : T \mid \Delta, s[\mathtt{p}]^\bullet : T & \text{session env} & S ::= \mathsf{nat} \mid \mathsf{bool} \mid T[\mathtt{p}] & \text{sorts} \\
G ::= \mathtt{p} \to \mathtt{q} : \{l_i(x_i : S_i)\{A_i\}.G_i\}_{i\in I} & \text{interaction} & T ::= \mathtt{p}!\{l_i(x_i : S_i)\{A_i\}.T_i\}_{i\in I} & \text{selection} \\
\quad \mid \ G_1 \mid G_2 & \text{parallel} & \quad \mid \ \mathtt{p}?\{l_i(x_i : S_i)\{A_i\}.T_i\}_{i\in I} & \text{branch} \\
\quad \mid \ \mu \mathtt{t}\langle e\rangle(x : S)\{A\}.G \mid \mathtt{t}\langle e\rangle \mid \mathsf{end} & \text{rec/end} & \quad \mid \ \mu \mathtt{t}\langle e\rangle(x : S)\{A\}.T \mid \mathtt{t}\langle e\rangle \mid \mathsf{end} & \text{rec/end}
\end{array}$$

Fig. 5. Monitors and global/endpoint assertions (resp. $\mathscr{M}$, G, and T)

the syntax for a monitor above consists of two typing environments, one for shared channels (Γ) and one for sessions (Δ). In Δ, we let $s[\mathtt{p}] : T$ represent a capability which is owned by the local process but is not active yet (i.e., it can still be sent to invite another process); $s[\mathtt{p}]^\bullet : T$ represents an active capability after the monitored process has joined the session as $\mathtt{p}$. The session environment Δ associates each *linear* capability to an endpoint assertion T which describes the behaviour of a specific role in a session.

When a principal creates a session instance $s : G$, the associated monitor learns about its specification from the *global assertion* G, which describes a global scenario among multiple participants annotated with logical formulae. The main construct is a labelled message exchange, where $\mathtt{p}$ sends $\mathtt{q}$ a label l_i (I is a finite set of integers) and a message x_i with sort S_i. Sorts include base types and shared channel types $T[\mathtt{p}]$. $T[\mathtt{p}]$ is an endpoint assertion T (described later) modelling role $\mathtt{p}$. The global assertion G_i describes the continuation of the session for the selected branch i, and A_i is a predicate on interaction variables specifying what $\mathtt{p}$ must guarantee and dually what $\mathtt{q}$ can rely on. A_i expresses a constraint on the choice of branch i (e.g., a guard that must hold when selecting label l_i) and on the value of the exchanged message x_i, which we call *interaction variable*. We do not fix a specific logic for A, we only assume it is decidable. Interactions bind each x_i in A_i and in G_i. Parallel composition is written as $G \mid G'$. A recursive assertion is guarded in the standard way and defines a recursion parameter with its initialisation and an invariant predicate A. end ends the session. Below we give an example of a global assertion from the Instrument Command use case.

Example 1 (OOI Instrument Command - Global Assertion). The following is the global assertion for our running example (Figures 1 and 2). A branch without predicate means its accompanying predicate is true. We sometimes omit labels when there is a single branch.

$$
\begin{aligned}
G_{IC} &= \mathtt{User} \to \mathtt{Register} : (x_{int} : \mathsf{InterfaceId}).\\
&\quad \mathtt{Register} \to \mathtt{User} : (x_n : \mathsf{Int})\{x_n > 0\}.\mathtt{User} \to \mathtt{Agent} : (x_p : \mathsf{Priority}).\\
&\quad \mathtt{Agent} \to \mathtt{User} : \{\mathtt{accept}().G_{acc},\ \ \mathtt{reject}(x_E : \mathsf{ErrData})\{x_P = \mathsf{high} \supset x_e \neq \mathsf{busy}\}\}\\
G_{acc} &= \mu \mathsf{t}\langle x_n\rangle(y)\{y \geq 0\}.\\
&\quad \mathtt{User} \to \mathtt{Agent} : \{\mathtt{more}(x_{com} : \mathsf{Command})\{y > 0 \wedge x_{com} \neq \mathsf{switch\text{-}off}\}.G_{com},\\
&\qquad\qquad \mathtt{quit}().\mathtt{Agent} \to \mathtt{Instrument} : \mathtt{quit}()\}\\
G_{com} &= \mathtt{Agent} \to \mathtt{Instrument} : (y_{com} : \mathsf{Command}).\\
&\quad \mathtt{Instrument} \to \mathtt{Agent} : (y_r : \mathsf{Response}).\mathtt{Agent} \to \mathtt{User} : (x_r : \mathsf{Response}).\ \mathsf{t}\langle y-1\rangle
\end{aligned}
$$

The scenario modelled by G_{IC} has already been described informally in § 2 (Figure 2). We here assume the consistency properties for assertions, the projectability and well-assertedness, from [3,17].[1]

The *endpoint assertions* T are local specification for endpoints, which are used by monitors. They specify which interactions are acceptable for an endpoint: in other words, an endpoint assertion specifies constraints on a session from the perspective of a specific role, rather than globally. In the grammar of Figure 5, selection expresses the transmission to $\mathtt{p}$ of a label l_i taken from a set $\{l_i\}_{i\in I}$, together with an interaction variable x_i of sort S_i and that the remaining interaction in the session is T_i. Branching is its dual. Others are similar to their global versions.

An *endpoint projection* or often simply *projection* $G \upharpoonright \mathtt{p}$ projects G onto $\mathtt{p}$ returning an endpoint assertion. Projection is defined as in [3] (please see online Appendix [20]). An example follows, projected from global assertions in Example 1.

Example 2 (OOI Instrument Command - Endpoint Assertion for User). Below we show the projections of respectively G_{IC}, G_{acc} and G_{com} onto $\mathtt{User}$.

$$
\begin{aligned}
T_{user} &= \mathtt{Register!}(x_{int} : \mathsf{InterfaceId}).\mathtt{Register?}(x_n : \mathsf{Int})\{x_n > 0\}.\mathtt{Agent!}(x_p : \mathsf{Priority}).\\
&\quad \mathtt{Agent?}\{\mathtt{accept}().T^u_{acc},\ \ \mathtt{reject}(x_e : \mathsf{ErrData})\{x_p = \mathsf{high} \supset x_e \neq \mathsf{busy}\}\}\\
T_{acc} &= \mu \mathsf{t}\langle x_n\rangle(y)\{y \geq 0\}.\\
&\quad \mathtt{Agent!}\{\mathtt{more}(x_{com} : \mathsf{Command})\{y > 0 \wedge x_{com} \neq \mathsf{switch\text{-}off}\}.T^u_{com},\ \ \mathtt{quit}()\}\\
T_{com} &= \mathtt{Agent?}(x_r : \mathsf{Response}).\mathsf{t}\langle y-1\rangle
\end{aligned}
$$

3.3 Semantics of Monitors

The semantics of a monitor is given as a LTS following the standard interpretation of typing environments in process calculi. Later, this relation is used to control the

[1] Projectability says that a global assertion is projectable to each endpoint [17] (defined in online Appendix [20]) while well-assertedness says that it is always possible for an endpoint to find a path which satisfies its own obligations [3] (defined in online Appendix [20]). In this paper we only treat global assertions satisfying these properties. See [3,17] for further explanations.

$$[\textsc{Tau,Join}] \qquad \mathscr{M} \xrightarrow{\tau} \mathscr{M} \qquad \mathscr{M}, s[\mathtt{p}]^\circ : T \xrightarrow{\mathsf{join}(s[\mathtt{p}])} \mathscr{M}, s[\mathtt{p}]^\bullet : T$$

$$[\textsc{New, NewS}] \qquad \frac{a \notin \mathsf{dom}(\mathscr{M})}{\mathscr{M} \xrightarrow{\mathsf{new}\ a:T[\mathtt{p}]} \mathscr{M}, a : T[\mathtt{p}]} \qquad \frac{s \notin \mathsf{dom}(\mathscr{M})}{\mathscr{M} \xrightarrow{\mathsf{new}\ s:G} \mathscr{M}, \{s[\mathtt{p}_i]^\circ : (G \upharpoonright \mathtt{p}_i)\}_{\mathtt{p}_i \in G}}$$

$$[\textsc{Req}] \qquad \mathscr{M}, a : T[\mathtt{p}], s[\mathtt{p}]^\circ : T \xrightarrow{\overline{a}\langle s[\mathtt{p}] : T\rangle} \mathscr{M}, a : T[\mathtt{p}]$$

$$[\textsc{Acc}] \qquad \frac{s \notin \mathsf{dom}(\mathscr{M})}{\mathscr{M}, a : T[\mathtt{p}] \xrightarrow{a\langle s[\mathtt{p}] : T\rangle} \mathscr{M}, a : T[\mathtt{p}], s[\mathtt{p}]^\circ : T}$$

$$[\textsc{Sel}] \qquad \frac{\mathscr{M} \vdash v : S_j,\ A_j\{v/x_j\} \downarrow \mathsf{true}, T \curvearrowright \mathtt{p}_2!\{(x_i : S_i)\{A_i\}.T_i\}_{i\in I}}{\mathscr{M}, s[\mathtt{p}_1]^\bullet : T \xrightarrow{s[\mathtt{p}_1,\mathtt{p}_2]!l_j\langle v\rangle} \mathscr{M}, s[\mathtt{p}_1]^\bullet : T_j\{v/x_j\}}$$

$$[\textsc{Bra}] \qquad \frac{\mathscr{M} \vdash v : S_j,\ A_j\{v/x_j\} \downarrow \mathsf{true}, T \curvearrowright \mathtt{p}_1?\{l_i(x_i : S_i)\{A_i\}.T_i\}_{1\in I}}{\mathscr{M}, s[\mathtt{p}_2]^\bullet : T \xrightarrow{s[\mathtt{p}_1,\mathtt{p}_2]?l_j\langle v\rangle} \mathscr{M}, s[\mathtt{p}_2]^\bullet : T_j\{v/x_j\}}$$

$$[\textsc{BraN}] \qquad \frac{S_j = T[\mathtt{p}],\ a \notin \mathsf{dom}(\mathscr{M}), \forall i\ a \notin A_i,\ A_i \downarrow \mathsf{true},\ T \curvearrowright \mathtt{p}_1?\{l_i(x_i : S_i)\{A_i\}.T_i\}_{1\in I}}{\mathscr{M}, s[\mathtt{p}_2]^\bullet : T \xrightarrow{s[\mathtt{p}_1,\mathtt{p}_2]?l_j\langle a\rangle} \mathscr{M}, a : S_j, s[\mathtt{p}_2]^\bullet : T_j}$$

Fig. 6. Labelled transition system for monitors

behaviour of processes. $\mathscr{M} \xrightarrow{\ell} \mathscr{M}'$ only if ℓ is a legal action (i.e., that a process should be allowed to perform by the monitor). We use the following labels:

$$\ell ::= \tau \mid \overline{a}\langle s[\mathtt{p}]\ :\ T\rangle \mid a\langle s[\mathtt{p}]\ :\ T\rangle \mid \mathsf{new}\ s : G \mid \mathsf{new}\ a : T[\mathtt{p}] \mid \mathsf{join}(s[\mathtt{p}]) \mid s[\mathtt{p}_1,\mathtt{p}_2]!l\langle v\rangle \mid s[\mathtt{p}_1,\mathtt{p}_2]?l\langle v\rangle$$

A label can be a τ-action, a session request or reception, the creation of a new session or shared channel, the join of a session, selection or branching. Request and selection (resp. reception and branching) are often collectively called *output* (resp. *input*).

The LTS for monitors is defined in Figure 6. Rule [New] allows a shared name creation with type $T[\mathtt{p}]$. Rule [NewS] is for a new session s with type G, which is always allowed as far as s is fresh; it adds the projections of G at each $\mathtt{p}_i$ (denoted by $G \upharpoonright \mathtt{p}_i$), endowing the local process with the capabilities $s[\mathtt{p}_i]$ to play behaviours of all roles in G. Rule [Join] activates session $s[\mathtt{p}]$. Rule [Req] represents that, for invitation at a, if the monitor includes the type of shared channel a as $T[\mathtt{p}]$, and the type of $\mathtt{p}$ in session s is exactly T, the monitor approves this invitation and, since this capability has been sent out, relinquishes $s[\mathtt{p}]$; rule [Acc] is its dual.

In [Sel], [Bra] and [BraN], we use *permutations*, denoted $T \curvearrowright T'$. A sound permutation (called *asynchronous subtyping* in [21]) changes the order of actions to capture the semantics of asynchronously arriving messages without affecting causally related actions. Consider the global assertion $\mathtt{p}_2 \to \mathtt{p}_1 : (x:S)\{A\}.\mathtt{p}_3 \to \mathtt{p}_1 : (x' : S')\{A'\}.G$ where $\mathtt{p}_1 \neq \mathtt{p}_3$. In an asynchronous network we cannot prevent message x' to reach $\mathtt{p}_1$ *earlier* than x. Hence $\mathtt{p}_1$'s monitor needs to accept the messages from $\mathtt{p}_2$ and $\mathtt{p}_3$ in any order. We define $T \curvearrowright T'$ when we can permute up an action in T to the top, and do nothing else, to reach T', via the axioms: $\mathtt{p}_1 \dagger_1 \{l_i(x_i : S_i)\{A_i\}.\mathtt{p}_2 \dagger_2 \{l'_j(x'_j : S'_j)\{A'_j\}.T_{ij}\}_{j\in I}\}_{i\in I} \curvearrowright$

$\mathtt{p}_2\dagger_2\{l'_j(x'_j:S'_j)\{A'_j\}.\mathtt{p}_1\dagger_1\{l_i(x_i:S_i)\{A_i\}.T_{ij}\}_{i\in I}\}_{j\in J}$ where $\dagger_1 = ?, \dagger_2 = ?$ or $\dagger_1 = !, \dagger_2 = !$ or $\dagger_1 = !, \dagger_2 = ?$. Note that $\dagger_1 = ?, \dagger_2 = !$ is *unsound* since the actions are causally related.

Thus [SEL] says that if the endpoint assertion of $s[\mathtt{p}_1]^\bullet$ at $\mathtt{p}_1$ can be permuted to $\mathtt{p}_2!\{l_i(x_i:S_i)\{A_i\}.T_i\}_{i\in I}$, then the outgoing message with label l_i sent from $\mathtt{p}_1$ to $\mathtt{p}_2$, as far as it satisfies formula $A_j\{v/x_j\}$, is approved by the monitor which prepares the next (incoming or outgoing) message with local specification T_j. Rule [BRA] is its symmetric (input) counterpart, while [BRAN] is one for fresh shared channel a (ensured by $a \notin \mathsf{dom}(\mathscr{M})$ and $a \notin A_i$), so that $a : S_j$ is added to the monitor.

Unmonitored networks are given by erasing the co-domain (types and assertions) from each monitor in a monitored network. The result of such erasure, the monitor which 'switches-off' the monitoring activity, acts simply as a gateway with information on local addresses, including a session endpoint $s[\mathtt{p}]$. We call this stripped-off gateway a *monitor-off* $(\mathscr{M}^\circ, \mathscr{M}_1^\circ, \ldots)$, used for routing session messages to the right destinations. The semantics of $\mathscr{M}^\circ$ is obtained by erasing the co-domain from each monitor in each rule in Figure 6. We write $\mathsf{erase}(\mathscr{M})$ for the monitor-off obtained through this erasure of $\mathscr{M}$. Hereafter we denote $\mathsf{erase}(\mathscr{M})$ as $\mathscr{M}^\circ$.

Example 3 ($\mathscr{M}$ vs $\mathscr{M}^\circ$). Let $\mathscr{M}_1 \stackrel{\text{def}}{=} s[\mathtt{p}_1]^\bullet : \mathtt{p}_2!(x : \mathsf{Int})\{x > 0\}.T$ and $\ell = s[\mathtt{p}_1, \mathtt{p}_2]!\langle -10\rangle$. Then $\mathscr{M}_1^\circ \xrightarrow{\ell}$ but $\mathscr{M}_1 \not\xrightarrow{\ell}$ since the value -10 does not satisfy the predicate $x > 0$ in the endpoint assertion for the session monitored by $\mathscr{M}_1$. Similarly, for type violations, if $\mathscr{M}_2 \stackrel{\text{def}}{=} s[\mathtt{p}_1]^\bullet : \mathtt{p}_2!(x{:}\mathsf{String})\{\mathsf{true}\}.T$ and $\ell = s[\mathtt{p}_1, \mathtt{p}_2]!\langle 10\rangle$ then $\mathscr{M}_2^\circ \xrightarrow{\ell}$ but $\mathscr{M}_2 \not\xrightarrow{\ell}$.

4 Monitored Network and Global Observables

4.1 Monitored Network

Syntax. We write $\mathscr{M}[P]$ for P monitored by $\mathscr{M}$, called *monitored process*. Then a *monitored network* or *network* $(N, N', \ldots)$ is given as:

$$N ::= \emptyset \mid \mathscr{M}[P] \mid N_1 \mid N_2 \mid (\nu a{:}T[\mathtt{p}])N \mid (\nu s{:}G)N \mid s : h \mid \overline{a}\langle s[\mathtt{p}] : T\rangle$$

A monitored network represents a network of processes and their monitors, together with messages in transit, which include global message queues for each session ($s : h$ where h is a partial sequence of messages) and an unordered collection of invitations $\overline{a}\langle s[\mathtt{p}] : T\rangle$.

Reduction and Transition. The dynamics of networks, in particular how messages travel from a local configuration through a network to another local configuration, is formalised by the reduction rules in Figure 7. In each rule except [PROC], the [-ERR] case is when the monitor detects a violation of the specification by the current action; we stipulate that $\mathscr{M}$ simply drops such message. Each rule corresponds to a rule for the monitor semantics defined in Figure 6. Rule [NEW] creates a fresh (bound) shared channel, while rule [NEWS] creates a new session channel, together with an empty (global)

$\lfloor$Proc$\rfloor$
$$\frac{P \xrightarrow{\tau} P'}{\mathscr{M}[P] \to \mathscr{M}[P']}$$

$\lfloor$New, New-err$\rfloor$
$$\frac{\mathscr{M} \xrightarrow{\mathsf{new}\ a:T[\mathtt{p}]} \mathscr{M}'}{\mathscr{M}[\mathsf{new}\, a:T[\mathtt{p}]\ \mathsf{in}\ P] \to (\nu a:T[\mathtt{p}])(\mathscr{M}'[P])} \qquad \frac{\mathscr{M} \not\xrightarrow{\mathsf{new}\ a:T[\mathtt{p}]}}{\mathscr{M}[\mathsf{new}\, a:T[\mathtt{p}]\ \mathsf{in}\ P] \to \mathscr{M}[\mathbf{0}]}$$

$\lfloor$NewS, NewS-err$\rfloor$
$$\frac{\mathscr{M} \xrightarrow{\mathsf{new}\ s:G} \mathscr{M}'}{\mathscr{M}[\mathsf{new}\, s:G\ \mathsf{in}\ P] \to (\nu s:G)(\mathscr{M}'[P] \mid s:\emptyset)} \qquad \frac{\mathscr{M} \not\xrightarrow{\mathsf{new}\ s:G}}{\mathscr{M}[\mathsf{new}\, s:G\ \mathsf{in}\ P] \to \mathscr{M}[\mathbf{0}]}$$

$\lfloor$Join, Join-err$\rfloor$
$$\frac{\mathscr{M} \xrightarrow{\mathsf{join}(s[\mathtt{p}])} \mathscr{M}'}{\mathscr{M}[\mathsf{join}\ s[\mathtt{p}];P] \to \mathscr{M}'[P \mid s[\mathtt{p}]:\emptyset]} \qquad \frac{\mathscr{M} \not\xrightarrow{\mathsf{join}(s[\mathtt{p}])} \mathscr{M}'}{\mathscr{M}[\mathsf{join}\ s[\mathtt{p}];P] \to \mathscr{M}[P]}$$

$\lfloor$Req, Req-err$\rfloor$
$$\frac{\mathscr{M} \xrightarrow{\overline{a}\langle s[\mathtt{p}] : T\rangle} \mathscr{M}'}{\mathscr{M}[\overline{a}\langle s[\mathtt{p}]\ :\ T\rangle] \to \mathscr{M}'[\mathbf{0}] \mid \overline{a}\langle s[\mathtt{p}]\ :\ T\rangle} \qquad \frac{\mathscr{M} \not\xrightarrow{\overline{a}\langle s[\mathtt{p}] : T\rangle} \mathscr{M}'}{\mathscr{M}[\overline{a}\langle s[\mathtt{p}]\ :\ T\rangle] \to \mathscr{M}[\mathbf{0}]}$$

$\lfloor$Acc, Acc-err$\rfloor$
$$\frac{\mathscr{M} \xrightarrow{a\langle s[\mathtt{p}] : T\rangle} \mathscr{M}'}{\mathscr{M}[\mathbf{0}] \mid \overline{a}\langle s[\mathtt{p}]\ :\ T\rangle \to \mathscr{M}'[\overline{a}\langle s[\mathtt{p}]\ :\ T\rangle]} \qquad \frac{\mathscr{M} \not\xrightarrow{a\langle s[\mathtt{p}] : T\rangle} \mathscr{M}'}{\mathscr{M}[\mathbf{0}] \mid \overline{a}\langle s[\mathtt{p}]\ :\ T\rangle \to \mathscr{M}[\mathbf{0}]}$$

$\lfloor$Out$\rfloor$
$$\frac{\mathscr{M} \xrightarrow{s[\mathtt{p}_1,\mathtt{p}_2]!l\langle v\rangle} \mathscr{M}'}{\mathscr{M}[s[\mathtt{p}_1] : \langle \mathtt{p}_1,\mathtt{p}_2,l\langle v\rangle\rangle \cdot h] \mid s:h' \to \mathscr{M}'[s[\mathtt{p}_1] : h] \mid s:h' \cdot \langle \mathtt{p}_1,\mathtt{p}_2,l\langle v\rangle\rangle}$$

$\lfloor$Out-err$\rfloor$
$$\frac{\mathscr{M} \not\xrightarrow{s[\mathtt{p}_1,\mathtt{p}_2]!l\langle v\rangle} \mathscr{M}'}{\mathscr{M}[s[\mathtt{p}_1] : \langle \mathtt{p}_1,\mathtt{p}_2,l\langle v\rangle\rangle \cdot h] \mid s:h' \to \mathscr{M}[s[\mathtt{p}_1] : h] \mid s:h'}$$

$\lfloor$In$\rfloor$
$$\frac{\mathscr{M} \xrightarrow{s[\mathtt{p}_1,\mathtt{p}_2]?l\langle v\rangle} \mathscr{M}'}{\mathscr{M}[s[\mathtt{p}_2] : h] \mid s:\langle \mathtt{p}_1,\mathtt{p}_2,l\langle v\rangle\rangle \cdot h' \to \mathscr{M}'[s[\mathtt{p}_2] : h \cdot \langle \mathtt{p}_1,\mathtt{p}_2,l\langle v\rangle\rangle] \mid s:h'}$$

$\lfloor$In-err$\rfloor$
$$\frac{\mathscr{M} \not\xrightarrow{s[\mathtt{p}_1,\mathtt{p}_2]?l\langle v\rangle} \mathscr{M}'}{\mathscr{M}[s[\mathtt{p}_2] : h] \mid s:\langle \mathtt{p}_1,\mathtt{p}_2,l\langle v\rangle\rangle \cdot h' \to \mathscr{M}[s[\mathtt{p}_2] : h] \mid s:h'}$$

We omit the standard context rules and structural congruence rules.

Fig. 7. Reduction for monitored network

queue used by all processes joining session s. Rule $\lfloor$Join$\rfloor$ creates a local queue for session s and role $\mathtt{p}$. In rules $\lfloor$Req,Acc$\rfloor$, monitor $\mathscr{M}$ checks outgoing and incoming invitations. Ideally, $\lfloor$Req$\rfloor$ forwards an outgoing invitation, which is still local to a process, to the external environment (dually for $\lfloor$Acc$\rfloor$ with an incoming invitation). In $\lfloor$Out$\rfloor$, $\mathscr{M}$ forwards a session message from a local queue $s[\mathtt{p}]$ into the global queue s (dually for $\lfloor$In$\rfloor$). Other rules are standard.

Example 4. Let $P = s[\mathtt{p}_1,\mathtt{p}_2]!\langle 100\rangle;P' \mid s[\mathtt{p}_1] : \emptyset$ be a process with an empty queue playing role $\mathtt{p}_1$ in s. Let also $\mathscr{M}_1 = s[\mathtt{p}_1]^\bullet : \mathtt{p}_2!(x:\mathsf{Int})\{x>0\}.T$ be its local monitor. The communication happens in two steps. First, the message is spawn into the local queue as $P \xrightarrow{\tau} P_2$ with $P_2 = P' \mid s[\mathtt{p}_1] : s[\mathtt{p}_1,\mathtt{p}_2]!\langle 100\rangle$ hence, since $\mathscr{M}_1 \xrightarrow{\tau} \mathscr{M}_1$, then $\mathscr{M}_1[P] \to \mathscr{M}_1[P_2]$. Second, the message is forwarded to the global queue as $P_2 \xrightarrow{\ell} P' \mid s[\mathtt{p}_1] : \emptyset$ with $\ell = s[\mathtt{p}_1,\mathtt{p}_2]!\langle 100\rangle$; hence since $\mathscr{M}_1 \xrightarrow{\ell} s[\mathtt{p}_1]^\bullet : T$ then $\mathscr{M}_1[P] \to s[\mathtt{p}_1]^\bullet : T[P' \mid s[\mathtt{p}_1] : \emptyset]$.

4.2 The Global Observables

As Figure 7 shows, from the perspective of the global network, all *global-behaviours* of monitored processes become unobservable. To analyse and state properties of monitored networks, the global-behaviours need be observed by linking global assertions to global interactions among monitored networks, neglecting local interactions inside endpoints. We therefore propose a notion of global observables and we define it through two labelled transitions systems: one for networks and one for environments (they represent the abstraction of global specifications and message flows).

First, we formalise the notion of the global observables of *networks* as follows: $N \overset{\ell}{\rightarrow}_{\mathrm{g}} N'$ for N without hiding, if any of its monitors of monitored processes has the transition ℓ (by the LTS given Figure 6). Thus global observability is the aggregate of what all monitors observe.

Second, we formalise a global observable environment, ranged over by $\mathscr{E}, \mathscr{E}', \ldots$, in order to witness the legality of all messages in transit. A global observable environment includes pending messages together with global assertions (for ease of defining its LTS, we also include local assertions). The use of pending messages is motivated as follows.

In a monitored network, we expect that all endpoint processes follow exactly what global assertions G define. However, at runtime, monitors at receiver-side can only observe the behaviours of processes through the passing messages, in order to capture whether the incoming message has been sent or not. If this message has not been sent, there should exist a monitor at sender-side specifying this sending action. In this case, these two monitors are coherent. On the other hand, if this message has been sent (so there is no any monitor contains the specification of this sending action), the monitor at receiver-side needs to be compensated not by another monitor specification but by a message in a global queue. This situation is illustrated by the following example.

Example 5. Assume a simple global protocol specifies that:

$$\mathrm{p}_1 \rightarrow \mathrm{p}_2 : (x : \mathsf{int})\{x > 5\}.G_2$$

At the beginning, the specification of sender-side monitor is

$$s[\mathrm{p}_1]^\bullet : \mathrm{p}_2!(x : \mathsf{int})\{x > 5\}.G_2 \upharpoonright \mathrm{p}_1$$

and the one of receiver-side is

$$s[\mathrm{p}_2]^\bullet : \mathrm{p}_1?(x : \mathsf{int})\{x > 5\}.G_2 \upharpoonright \mathrm{p}_2$$

When the sender-side monitor permits an outgoing message $s : \langle \mathrm{p}_1, \mathrm{p}_2, 10 \rangle$, (i.e., a legal sending action), its specification immediately changes to $s[\mathrm{p}_1]^\bullet : G_2 \upharpoonright \mathrm{p}_1$ for the next action; however, the specification of receiver-side monitor may not change because it is waiting for the message that is travelling in the global queue $s : h \cdot \langle \mathrm{p}_1, \mathrm{p}_2, 10 \rangle$; as long as the message does not arrive, the receiver-side monitor cannot change its specification. In this case, the receiver-side monitor knows the messages will certainly come by looking at the global queue.

Keeping in mind that a global queue (containing assertions of run-time pending messages) is needed as a part of the global observable environment, we define the syntax of global observable environment $\mathscr{E}$ as follows:

$$\begin{array}{lll} \mathscr{E} ::= \Gamma, \Delta, \Theta & \Gamma ::= \emptyset \mid \Gamma, a : T[\mathtt{p}] & \Theta ::= \emptyset \mid \Theta, s : G \\ \Delta ::= \emptyset \mid \Delta, s[\mathtt{p}] : T \mid \Delta, s[\mathtt{p}]^{\bullet} : T \mid \Delta, s : \vec{mv} & & mv ::= \langle \mathtt{p}, \mathtt{q}, l\langle v \rangle \rangle \end{array}$$

where Γ is a typing environment, Δ is a session environment as the one in Figure 5, to which we add message assertions $s : \vec{mv}$ to model a global queue environment. Each assignment in $s : \vec{mv}$ is of the form $s : mv_1..mv_n$, $n \geq 0$, where mv_i is a message assertion of shape $\langle \mathtt{p}, \mathtt{p}', l\langle v \rangle \rangle$. Θ is a global environment associating sessions to global assertions. Thus $\mathscr{E}$ can use both global specification and pending messages to help monitors and thus ensure that the whole network is in a correct state.

4.3 Labelled Transition Rules for $\mathscr{E}$

In Figure 8 we define $\mathscr{E} \xrightarrow{\ell}_{\mathrm{g}} \mathscr{E}'$, which says: environment $\mathscr{E}$ allows a global observation of ℓ as a valid interaction, and, after the corresponding changes in the assertions and global queues, becomes ready to observe a possible next action as $\mathscr{E}'$.

$$\mathscr{E} \xrightarrow{\tau}_{\mathrm{g}} \mathscr{E} \qquad [\mathscr{E}\text{-TAU}]$$

$$\frac{\mathscr{E} \vdash v : S_j \quad A_j\{v/x_j\} \downarrow \mathsf{true} \quad G \curvearrowright \mathtt{p}_1 \to \mathtt{p}_2 : \{l_i(x_i : S_i)\{A_i\}.G_i\}_{i \in I}}{\mathscr{E}, s : G, s[\mathtt{p}_2]^{\bullet} : T, s : \langle \mathtt{p}_1, \mathtt{p}_2, l_j\langle v \rangle \rangle \cdot \vec{mv} \xrightarrow{s[\mathtt{p}_1, \mathtt{p}_2]?l_j\langle v \rangle}_{\mathrm{g}} \mathscr{E}, s : G_j\{v/x_j\}, s[\mathtt{p}_2]^{\bullet} : T_j, s : \vec{mv}} \qquad [\mathscr{E}\text{-BCH}]$$

$$\frac{\mathscr{E} \vdash v : S_j \quad A_j\{v/x_j\} \downarrow \mathsf{true} \quad T \curvearrowright \mathtt{p}_2!\{l_i(x_i : S_j)\{A_i\}.T_i\}_{i \in I}}{\mathscr{E}, s[\mathtt{p}_1]^{\bullet} : T, s : \vec{mv} \xrightarrow{s[\mathtt{p}_1, \mathtt{p}_2]!l_j\langle v \rangle}_{\mathrm{g}} \mathscr{E}, s[\mathtt{p}_1]^{\bullet} : T_j\{v/x_j\}, s : \vec{mv} \cdot \langle \mathtt{p}_1, \mathtt{p}_2, l_j\langle v \rangle \rangle} \qquad [\mathscr{E}\text{-SEL}]$$

$$\frac{s \notin \mathsf{dom}(\mathscr{E})}{\mathscr{E} \xrightarrow{\mathsf{new}\ s:G}_{\mathrm{g}} \mathscr{E}, s : G, \{s[\mathtt{p}_i] : (G \upharpoonright \mathtt{p}_i)\}_{\mathtt{p}_i \in G}, s : \emptyset} \qquad [\mathscr{E}\text{-NEW S}]$$

$$\frac{a \notin \mathsf{dom}(\mathscr{E})}{\mathscr{E} \xrightarrow{\mathsf{new}\ a:T[\mathtt{p}]}_{\mathrm{g}} \mathscr{E}, a : T[\mathtt{p}]} \qquad [\mathscr{E}\text{-NEW A}]$$

$$\mathscr{E}, s[\mathtt{p}] : T \xrightarrow{\mathsf{join}(s[\mathtt{p}])}_{\mathrm{g}} \mathscr{E}, s[\mathtt{p}]^{\bullet} : T \qquad [\mathscr{E}\text{-JOIN}]$$

$$\mathscr{E}, a : T[\mathtt{p}], s[\mathtt{p}] : T \xrightarrow{\overline{a}\langle s[\mathtt{p}] : T \rangle}_{\mathrm{g}} \mathscr{E}, a : T[\mathtt{p}], s[\mathtt{p}] : T \qquad [\mathscr{E}\text{-REQ}]$$

$$\mathscr{E}, a : T[\mathtt{p}], s[\mathtt{p}] : T \xrightarrow{a\langle s[\mathtt{p}] : T \rangle}_{\mathrm{g}} \mathscr{E}, a : T[\mathtt{p}], s[\mathtt{p}] : T \qquad [\mathscr{E}\text{-ACC}]$$

Fig. 8. Labelled transition system for environments

In [$\mathscr{E}$-BCH], T is obtained by removing all *outputted actions* of $\mathtt{p}_2$ (i.e., $\vec{mv} \upharpoonright \mathtt{p}_2$) from all actions of $\mathtt{p}_2$ (i.e., $G \upharpoonright \mathtt{p}_2$); T_j is, similarly, obtained by removing outputted actions of $\mathtt{p}_2$ from all actions in G_j of $\mathtt{p}_2$ and replacing x_j by v in $G_j \upharpoonright \mathtt{p}_2$. The reason why this is needed, instead of simply applying $G \upharpoonright \mathtt{p}_2$, is to obtain the appropriate endpoint specification considering the asynchrony nature of interactions. We use the following example to show this situation.

Example 6. Given a simple global assertion

$$\mathtt{p}_1 \to \mathtt{q}_1 : (x_1 : \mathsf{int})\{x_1 > 0\}.\mathtt{p}_1 \to \mathtt{q}_2 : (x_2 : \mathsf{int})\{x_2 > 1\}.\mathtt{q}_3 \to \mathtt{p}_1 : (x_3 : \mathsf{int})\{x_3 > 2\}.\mathsf{end} \tag{1}$$

Obviously, as $\mathtt{p}_1$ is active, the local specification of monitor at $\mathtt{p}_1$ is

$$s[\mathtt{p}_1]^\bullet : \mathtt{q}_1!(x_1 : \mathsf{int})\{x_1 > 0\}.\mathtt{q}_2!(x_2 : \mathsf{int})\{x_2 > 1\}.\mathtt{q}_3?(x_3 : \mathsf{int})\{x_3 > 2\}.\mathsf{end} \tag{2}$$

It is possible that participants interact in the following order: (I) $\mathtt{q}_3$ firstly sends message $\langle \mathtt{q}_3, \mathtt{p}_1, \langle 3 \rangle \rangle$ to $\mathtt{p}_1$, (II) then $\mathtt{p}_1$ sends messages to $\mathtt{q}_1$ and $\mathtt{q}_2$ with $\langle \mathtt{p}_1, \mathtt{q}_1, \langle 1 \rangle \rangle$ and $\langle \mathtt{p}_1, \mathtt{q}_2, \langle 2 \rangle \rangle$ respectively. Note that, as the first interaction happens, $\mathtt{p}_1$ may not receive this messages immediately. Assume before this message arrives, $\mathtt{p}_1$ has sent out messages $\langle \mathtt{p}_1, \mathtt{q}_1, \langle 1 \rangle \rangle$ and $\langle \mathtt{p}_1, \mathtt{q}_2, \langle 2 \rangle \rangle$. Therefore, the global queue is

$$\langle \mathtt{q}_3, \mathtt{p}_1, \langle 1 \rangle \rangle \cdot \langle \mathtt{p}_1, \mathtt{q}_1, \langle 1 \rangle \rangle \cdot \langle \mathtt{p}_1, \mathtt{q}_2, \langle 2 \rangle \rangle$$

and the global assertion *maintains* as Equation (1) because no message has been received. However, the current specification of monitor at $\mathtt{p}_1$ is $s[\mathtt{p}_1]^\bullet : \mathtt{q}_3?(x_3 : \mathsf{int})\{x_3 > 2\}.\mathsf{end}$ since it has done two output actions. In such a case, $G \upharpoonright \mathtt{p}_1$, which is still Equation (2), cannot reflect the reality of $\mathtt{p}_1$ that is going to receive a message because of the asynchrony nature.

Continue with rule [$\mathscr{E}$-BCH]. It says that if a value is typed S_j and satisfies A_j, and G has a corresponding interaction up to permutations, it allows $\langle \mathtt{p}_1, \mathtt{p}_2, l_j\langle v \rangle \rangle$ to be received, resulting in the new local/global assertions. The permutation relation $G \curvearrowright G'$ (defined in online Appendix [20]) means that the specification G can be permuted to G', so that what is not *apparently* an active action in G becomes active in G' modulo permutation. For example, if both `Buyer` and `Broker` are sending a message to `Seller`:

$$G \stackrel{\text{def}}{=} \mathtt{Buyer} \to \mathtt{Seller} : (x : \mathsf{integer}).\mathtt{Broker} \to \mathtt{Seller} :: (x : \mathsf{string}).\mathsf{end}$$

then `Broker`'s message may as well arrive at `Seller` first, hence we permute this to

$$G' \stackrel{\text{def}}{=} \mathtt{Broker} \to \mathtt{Seller} : (x : \mathsf{string}).\mathtt{Buyer} \to \mathtt{Seller} : (x : \mathsf{integer}).\mathsf{end}$$

where $G \curvearrowright G'$ and G' is ready to check an appropriate message from `Broker` to `Seller`. Rule [$\mathscr{E}$-SEL] states that when $\mathscr{E}$ approves an output, its global assertion is unchanged but put a message to the global queue, indicating that an interaction has *partially* happened by an output, but not completed.

[$\mathscr{E}$-NEW S] says if a session s is new to $\mathscr{E}$, $\mathscr{E}$ adds the global assertion of s and the session environments $\{s[\mathtt{p}_i] : (G \upharpoonright \mathtt{p}_i)\}_{\mathtt{p}_i \in G}$ corresponding to s, and the queue $s : \emptyset$ at the

same time. [$\mathscr{E}$-NEW A] says if a shared name a is new to $\mathscr{E}$, then $\mathscr{E}$ adds this new shared channel. [$\mathscr{E}$-JOIN] makes the specified session-role $s[\mathrm{p}]$ become active. Finally, since $\mathscr{E}$ watches the global environment (i.e., by watching all endpoint monitors), as request and accept happens at endpoint, it does not affect the global environment. [$\mathscr{E}$-REQ] and [$\mathscr{E}$-ACC] state this fact. Note that, for [$\mathscr{E}$-REQ] (or [$\mathscr{E}$-ACC]), if the left-hand side environment violates the rule, then the rule [$\mathscr{E}$-TAU] is applied; which means that, globally, there is no such a request (or accept) action observed.

5 Transparency, Conformance and Session Fidelity

This section informally outlines the key local and global safety properties that our monitoring mechanism can enforce.

5.1 Local Safety and Transparency

We first list the properties that monitors guarantee for local configurations. Hereafter, we let $\mathscr{L}$, a *located process*, stand for either a monitored process $\mathscr{M}[P]$ or monitor-off process $\mathscr{M}^\circ[P]$. *Conformance of a located process to a monitor's specification* means this located process only sends "good" messages (which implies that its monitor always permits those messages) and receive "good" messages (which have been approved by its monitor). In this case we say this process *conforms* to $\mathscr{M}$, (i.e. it behaves well w.r.t. $\mathscr{M}$), represented as $\mathscr{M} \models \mathscr{L}$.

We can then show every monitored process conforms to the specification given by its monitor. Locally speaking, a monitored-process and a monitor-off process behave precisely in the same way if the latter already conforms to its monitor's specification. Therefore $\mathscr{M} \models \mathscr{M}^\circ[P]$ implies $\mathscr{M} \models \mathscr{M}^\circ[P] \sim \mathscr{M}[P]$, denoting that $\mathscr{M}^\circ[P]$ and $\mathscr{M}[P]$ are bisimilar under $\mathscr{M}$ [2] Note that a process P that can be validated by $\Gamma \vdash P \triangleright \Delta$ (in the sense of [3]) is guaranteed to behave correctly and thus satisfies $\mathscr{M} \models \mathscr{M}^\circ[P]$ for $\mathscr{M} = \Gamma, \Delta$.

5.2 Global Safety, Fidelity and Transparency

Global Safety and Transparency. Let us say a network is *open* if it has no name restrictions, and *receivable* if all pending messages in global queue can be received by their destinations. Here we state that, *given N is open and receivable, $N \xrightarrow{\ell}_g N'$ implies, for an input ℓ, N' is receivable.*

To describe each party in a network behaves consistently with other parties, *coherence* is defined: an open network N is *coherent* if all of its pending messages are *receivable* up to *permutation* of actions $\curvearrowright$ and, after these messages have been received, the resulting monitors, say $\{\mathscr{M}_i\}_{i\in I}$ with $\mathscr{M}_i = \Gamma_i, \Delta_i$, satisfy the following conditions: **(1)** shared channels carry the same specification; **(2)** no two locations share the same role

[2] Monitored strong bisimulation is defined by: (1) for output or silent actions, if $\mathscr{M}^\circ[P]$ allows an action then also the monitor must allow that action; (2) for input actions, if the monitor allows an action then also $\mathscr{M}^\circ[P]$ must allow that action.

for a same session and each assertion has its dual; **(3)** for each session channel, the local assertion for each of its roles is the result of projecting from a common global assertion.

Coherence should be preserved by the interactions that can happen in a monitored network. Thus $N \xrightarrow{\ell}_{\mathrm{g}} N'$ with N coherent implies N' is coherent. When N is open, N is *locally conformant* if, for each monitored process $\mathscr{M}_i[P_i]$ in N, we have $\mathscr{M}_i \models \mathsf{erase}(\mathscr{M}_i)[P_i]$. If N is coherent and locally conformant and such that $N \xrightarrow{\ell_1}_{\mathrm{g}} \cdots \xrightarrow{\ell_n}_{\mathrm{g}} N'$, then N' is also coherent and locally conformant. Let $\sim$ be the standard strong bisimilarity defined by $\xrightarrow{\ell}_{\mathrm{g}}$. By the global invariance, we have: *N, which is coherent and locally conformant, implies* $N \sim \mathsf{erase}(N)$.

Session Fidelity. When a new session is generated, it is associated with a global assertion G. This notion is called *session fidelity* [17].

The coherence of $\mathscr{E}$ is defined as the one for network (for the full definition, please see the online Appendix [20]). Given $\mathscr{E}$ and N open, we write $\mathscr{E} \vdash N$ when: $\mathscr{E}$ is coherent, its shared and linear environments come from the *monitors* in N, and its global queue environment comes from the pending messages in N. Whenever N is coherent, we can construct a (coherent) $\mathscr{E}$ such that $\mathscr{E} \vdash N$. Using these results and the exact correspondence between the permutation rules over assertions, we can establish the following result: $\mathscr{E} \vdash N$ *and* $N \xrightarrow{\ell}_g N'$ *implies* $\mathscr{E} \xrightarrow{\ell}_g \mathscr{E}'$ *such that* $\mathscr{E}' \vdash N'$. It states that global interactions in a coherent network never violate the expected network-wide global specifications: the former follows the latter step by step.

6 Related Work and Conclusion

Related work. Our previous work [3] uses the static validation of an endpoint process against local (endpoint) assertions for guaranteeing global safety of their behaviour. In other words, the postulated global properties are guaranteed only if the local code of *every* endpoint is statically verified. As discussed in Introduction, this assumption is often not practicable in heterogeneous distributed applications where endpoints are located in multiple administrative domains thus we cannot trust all of the participants to well-behave. The present paper also gives, for the first time, a direct formalisation of the notion of global invariants through global transitions and the induced bisimilarity.

The monitoring mechanism presented in this paper can be seen as a distributed variant of *runtime verification* in the sense of [1,14,15]. We monitor program executions to detect violation of properties and to enforce correct behaviours. The proposed approach is light-weight, concerned only with the observable executions, and does not aim to give a conclusive analysis about all their possible behaviours. Hence it is applicable to real-world systems, where off-line formal verification is intractable or impossible due to incomplete information on system components or the dynamic change in requirements.

In the classification of [8,18], our monitor is *online* (program execution is checked as it takes place) rather than *offline* (only the history of the execution is analysed), and is *outline* (or *external*) (the monitors run as separate processes and analyse the program via its observable events) rather than *inline* (the monitors are embedded in the target programs). This combination is the most effective for specifying and maintaining global correctness and protecting endpoints from illegal interactions (cf. § 2).

Our monitor mechanism respects two important principles: *soundness*, which states that enforcement results in a correct behaviour, and *transparency*, which requires the behaviour of a correct program not to be modified (a principle studied for example in an automata framework [11,25]). Our monitor *suppresses* [19] the illegal actions (as seen in (else) case of $\mathscr{M} \xrightarrow{\ell} \mathscr{M}'$) but neither immediately halts nor inserts the correct actions. Incorporating more elaborate reactions as in edit automata [19,25] is interesting future topics. None of the above work can ensure the fidelity to application-level multiparty global protocols for distributed applications with logical properties.

Our work uses a distributed process calculus to model networks of monitored (possibly unsafe) processes. From this viewpoint, the most closely related formalism is safeDpi [16] (a precursor of [24]) which models filtering for migrating processes, preserving type safety, using channel dependent types. Recently dynamic joining mechanisms are studied in the conversation-calculus [4] (which uses conversation contexts) and [13] (based on roles). Our primitive invites other parties via capability passing through shared channels, while [4,13] formalise joining to the existing session and maintain progress by advanced type checking. The main difference is that our work is about runtime enforcement of global protocol properties rather than static type checking, and that we enforce properties by local runtime checking. Our framework also incorporates logical assertions, offering more fine-grained logical specifications than bare protocols representable by types.

Another recent work [2,9] presents a secure implementation of sequential multiparty sessions. Through the mechanised use of cryptography, the integrity of session executions is protected in the presence of an active attacker controlling the network and some peers. The main differences from [2,9] are our support of a more general class of global specifications based on *MPSA*s and our choice of a framework based on trusted external monitors. We indeed advocate a framework independent from any particular programming language, enabling a greater applicability for large-scale distributed infrastructures. Unifying these two approaches by including active, un-monitored, attackers to the network offers an interesting research opportunity.

Conclusion and On-Going Work. Our formalism aims at providing a reference semantics for efficient and interoperable monitors in order to enforce safe multiparty session executions. A preliminary prototype is running over an advanced messaging middleware, and consists of: a protocol specification language called Scribble [26], the interfaces and runtimes for Scala, Java and Ocaml, and distributed monitors written in Ocaml. Following the monitoring rules in Figure 6, a monitor can be implemented efficiently (incl. the projection algorithm for generating monitors, which is polynomial against the size of global types [12]). Collaborations with OOI and Scribble projects to develop a *MPSA*-based monitor architecture of industrial strength are on-going.

References

1. Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.): RV 2010. LNCS, vol. 6418. Springer, Heidelberg (2010)
2. Bhargavan, K., Corin, R., Deniélou, P.-M., Fournet, C., Leifer, J.: Cryptographic protocol synthesis and verification for multiparty sessions. In: CSF, pp. 124–140 (2009)

3. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A Theory of Design-by-Contract for Distributed Multiparty Interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 162–176. Springer, Heidelberg (2010)
4. Caires, L., Vieira, H.T.: Conversation Types. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 285–300. Springer, Heidelberg (2009)
5. Carbone, M., Honda, K., Yoshida, N.: Structured Interactional Exceptions in Session Types. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 402–417. Springer, Heidelberg (2008)
6. W3C WS-CDL, http://www.w3.org/2002/ws/chor/
7. Chave, M., Arrott, A., Farcas, C., Farcas, E., Krueger, I., Meisinger, M., Orcutt, J., Vernon, F., Peach, C., Schofield, O., Kleinert, J.: Cyberinfrastructure for the US Ocean Observatories Initiative. In: Proc. IEEE OCEANS 2009. IEEE (2009)
8. Chen, F., Rosu, G.: MOP:An Efficient and Generic Runtime Verification Framework. In: OOPSLA, pp. 569–588 (2007)
9. Corin, R., Denielou, P.-M., Fournet, C., Bhargavan, K., Leifer, J.: Secure Implementations for Typed Session Abstractions. In: CSF, pp. 170–186. IEEE Computer Society (2007)
10. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed Systems, Concepts and Design. Addison-Wesley (2001)
11. Dam, M., Jacobs, B., Lundblad, A., Piessens, F.: Security Monitor Inlining for Multithreaded Java. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 546–569. Springer, Heidelberg (2009)
12. Deniélou, P.-M., Yoshida, N.: Buffered Communication Analysis in Distributed Multiparty Sessions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 343–357. Springer, Heidelberg (2010)
13. Deniélou, P.-M., Yoshida, N.: Dynamic multirole session types. In: POPL, pp. 435–446 (2011)
14. Falcone, Y.: You Should Better Enforce Than Verify. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 89–105. Springer, Heidelberg (2010)
15. Havelund, K., Goldberg, A.: Verify Your Runs. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 374–383. Springer, Heidelberg (2008)
16. Hennessy, M., Rathke, J., Yoshida, N.: SafeDpi: a language for controlling mobile code. Acta Inf. 42(4-5), 227–290 (2005)
17. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL 2008, pp. 273–284. ACM (2008)
18. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)
19. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Trans. Inf. Syst. Secur. 12, 19:1–19:41 (2009)
20. Online Appendix of this paper, http://www.eecs.qmul.ac.uk/~tcchen/TGC11/
21. Mostrous, D., Yoshida, N., Honda, K.: Global Principal Typing in Partially Commutative Asynchronous Sessions. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 316–332. Springer, Heidelberg (2009)
22. Mullender, S. (ed.): Distributed Systems. Addison-Wesley (1993)
23. Ocean Observatories Initiative (OOI), http://www.oceanleadership.org/programs-and-partnerships/ocean-observing/ooi/
24. Riely, J., Hennessy, M.: Trust and partial typing in open systems of mobile agents. In: Proc. POPL 1999 (1999)
25. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3, 30–50 (2000)
26. Scribble Project homepage, http://www.scribble.org