DOI 10.1007/s00165-016-0377-z

Formal Aspects of Computing (2016) 28: 531–566

Formal Aspects of Computing

Event-based run-time adaptation in communication-centric systems

Cinzia Di Giusto[1] and Jorge A. Pérez[2]

[1] Université de Nice Sophia-Antipolis, CNRS, I3S, UMR 7271, Sophia Antipolis, France
[2] University of Groningen, Groningen, The Netherlands

Abstract. *Communication-centric systems* are software systems built as assemblies of distributed artifacts that interact following predefined communication protocols. *Session-based concurrency* is a type-based approach to ensure the conformance of communication-centric systems to such protocols. This paper presents a model of session-based concurrency with mechanisms for *run-time adaptation*. Our model allows us to specify communication-centric systems whose session behavior can be dynamically updated at run-time. We improve on previous work by proposing an *event-based* approach: adaptation requests, issued by the system itself or by its context, are assimilated to events which may trigger adaptation routines. These routines exploit type-directed checks to enable the reconfiguration of processes with active protocols. We equip our model with a type system that ensures *communication safety* and *consistency* properties: while safety guarantees absence of run-time communication errors, consistency ensures that update actions do not disrupt already established session protocols. We provide soundness results for binary and multiparty protocols.

Keywords: Concurrency, Behavioral types, Session types, Run-time adaptation, Process calculi.

1. Introduction

Context and motivation. Modern software systems are increasingly built as assemblies of distributed artifacts that interact by following predefined communication protocols. Correctness in these *communication-centric* systems, therefore, depends on ensuring that such dialogues conform to their protocols. *Session-based concurrency* is a type-based approach to ensure conformance to prescribed protocols: dialogues are organized into units called *sessions*; interaction patterns are abstracted as *session types* [HVK98], against which specifications may be checked. As these specifications are often given in the π-calculus [MPW92, HVK98], protocol conformance can be analyzed at the level of concurrent processes.

As communication-centric systems typically operate on open and dynamic infrastructures, *run-time adaptation* appears as an indispensable feature to ensure their flexible and uninterrupted operation. Here we understand run-time adaptation as the dynamic reconfiguration of a system's behavior in response to an exceptional event. Examples of exceptional events are, e.g., a varying requirement or a local failure. These events are not necessarily

Correspondence and offprint requests to: Jorge A. Pérez, e-mail: j.a.perez@rug.nl

catastrophic but are hard to predict. In this context, conformance to structured protocols and run-time adaptation appear as intertwined concerns: although the specification of run-time adaptation may not be strictly tied to protocol reconfiguration, steps of run-time adaptation have a direct influence in a system's communicating behavior and overall correctness.

We are interested in enhancing models of session-based concurrency with forms of run-time adaptation. This entails extending models of session-based concurrency with constructs for representing adaptation, but also developing appropriate analysis and verification techniques for such extended models.

As a first answer to this challenge, in previous work [DP13a, DP15] we extended a typed process framework for binary sessions with constructs from the model of *adaptable processes* [BDPZ12]. In adaptable processes there are two key constructs, namely *located processes* and *update processes*:

- Given a process P, the located process $\mathtt{loc}\lfloor P \rceil$ denotes the fact that P resides in *location* $\mathtt{loc}$. Locations are explicit delimiters for process behavior: they denote possibly nested computation sites. Since locations are *transparent*, process P in $\mathtt{loc}\lfloor P \rceil$ may interact with processes outside $\mathtt{loc}$.
- An update process $\mathtt{loc}\{U\}$ specifies that the behavior currently enclosed by location $\mathtt{loc}$ should be modified by the *adaptation routine* U—intuitively, a function from processes to processes located at $\mathtt{loc}$.

At the operational level, located and update processes are complementary: processes $\mathtt{loc}\lfloor P \rceil$ and $\mathtt{loc}\{U\}$ are meant to synchronize in order to realize an adaptation step. As we will see, U is applied to P by "moving" to its enclosing context. The exact operational semantics that justifies this adaptation step will be illustrated shortly. Observe that since processes within locations (such as P in $\mathtt{loc}\lfloor P \rceil$) may implement one or several session protocols, the interaction with update processes allow us to specify adaptation routines that concern several running sessions.

We would like to guarantee that adaptation steps resulting from the synchronization of located and update processes preserve overall system correctness. In a session-typed setting this entails ensuring that such synchronizations do not jeopardize the session protocols enclosed by locations. As a simple example of a dangerous adaptation routine, consider the update process $\mathtt{loc}\{\mathbf{0}\}$, which says that the new behavior at $\mathtt{loc}$ should be the inactive process $\mathbf{0}$, without any considerations on the current behavior at $\mathtt{loc}$. Indeed, a synchronization between $\mathtt{loc}\lfloor P \rceil$ and $\mathtt{loc}\{\mathbf{0}\}$ would result into process $\mathtt{loc}\lfloor \mathbf{0} \rceil$, therefore discarding running session protocols in P. Processes depending on the sessions implemented by P would lack complementary partners, thus compromising any conceivable form of protocol conformance.

To rule out adaptation steps that jeopardize established protocols, communication and adaptation actions should be harmonized. In [DP13a, DP15] we defined the notion of *consistency*, according to which a located process can be updated only if it does not contain active (established) session protocols. Adaptation actions are enabled only for locations that do not enclose running sessions; an associated type system ensures protocol conformance but also that session protocols are never disrupted as a result of an adaptation step. This is a simple solution; since adaptation is enabled only when sessions in a location are in a quiescent state, it is an alternative that privileges communication over adaptation. In applications, however, it may be necessary to give communication and adaptation a similar status, enabling adaptation steps also on locations that contain already established sessions. The *main contribution* of this paper is a session-typed framework that enables this kind of adaptation steps, thus overcoming the limitations of the framework introduced in [DP13a, DP15]. Next we elaborate on our approach.

Our approach. To enable adaptation steps on locations that contain already established sessions, we propose update processes $\mathtt{loc}\{U\}$ in which the adaptation routine U (i) Dynamically checks the current state of the protocols running in $\mathtt{loc}$ and (ii) Determines a new process behavior for $\mathtt{loc}$ based on that state.

In their simplest form, our novel update processes concern located processes which implement only one session channel. They are of the form:

$$\mathtt{loc}\big\{\mathsf{case}\ x\ \mathsf{of}\ \{(\beta_1 : P_1), (\beta_2 : P_2), \ldots, (\beta_m : P_m)\}\big\} \quad (m \geq 1) \tag{1}$$

where x is a channel variable, $\beta_1, \ldots, \beta_m$ are session types (cf. Sect. 2.1), and $P_1, \ldots, P_m$ are processes in which x occurs free. Intuitively, each pair $(\beta_i : P_i)$ represents an *alternative* for updating location $\mathtt{loc}$: session type β_i denotes the protocol state under which process P_i will be used as the new implementation at $\mathtt{loc}$.

Suppose a process $\mathtt{loc}\lfloor Q \rceil$ in which Q implements a session of type α along session channel κ. The interaction between $\mathtt{loc}\lfloor Q \rceil$ and the update process in (1) intuitively proceeds as follows: If there is a β_j (with $j \in \{1, \ldots, m\}$) that "matches" with α, then the located process is updated to $\mathtt{loc}\lfloor P_j[\kappa/x] \rceil$. That is, the (current) protocol state

α guides the selection of the j-th adaptation alternative. Otherwise, if no β_j validates a match, then the located process is kept unchanged as $\texttt{loc}\big[Q\big]$ and the update process is consumed.

In their general form, our novel update processes may act on located processes which enclose more than one running session, thus adding considerable expressiveness. Such an update process is denoted:

$$\texttt{loc}\big\{\mathsf{case}\ x_1,\ldots,x_n\ \mathsf{of}\ \{(\beta_i^1;\ \cdots;\ \beta_i^n) : P_i\}_{i\in I}\big\} \tag{2}$$

where n is the number of running sessions at location $\texttt{loc}$. Intuitively, $(\beta_j^1;\ \cdots;\ \beta_j^n) : P_j$ denotes the j-th alternative for updating $\texttt{loc}$: session types $\beta_j^1,\ldots,\beta_j^n$ denote the protocol states under which process P_j will be used as the new implementation at $\texttt{loc}$. That is, the update process in (2) is meant to interact with a located process $\texttt{loc}\big[Q\big]$ in which Q implements session types $\alpha_1,\ldots,\alpha_n$; to enable an adaptation step, each α_i must "match" with β_j^i for all $i \in \{1,\ldots,n\}$.

The semantics for our *typeful* update processes exploits *session monitors*—processes that maintain the (current) protocol state at a given channel—and constructs for *dynamic type inspection*, as put forward in [Kou12, KYHH16].

As just described, our update processes use types to select an alternative for adaptation—that is, types say *how* to adapt. However, update processes by themselves they do not specify *when* adaptation should be triggered. A simple solution is to "embed" update processes within session communication prefixes. Unfortunately, such a specification style would only allow to handle exceptional conditions which can be fully characterized in advance. Other important exceptional conditions, in particular contextual and/or unsolicited run-time conditions, cannot always be anticipated/predicated nor have a clear causal dependency with steps related to structured communications.

To offer a uniform solution to the issue of when to trigger update processes, we propose an *event-based* approach. Our process syntax includes *adaptation signals* that are similar to communication prefixes and that can be used to issue an adaptation request for a given location. Each location is endowed with a *queue* that stores its pending adaptation signals/requests. To detect a request r in the queue of $\texttt{loc}$ and react accordingly, we use the *arrival predicate* $\mathsf{arrive}(\texttt{loc}, r)$, proposed in [Kou12]. The use of adaptation requests and the arrival predicate allows us to maintain a separation between structured communications (disciplined by session types) and the events that ultimately trigger update processes.

As an example, let $\texttt{upd}_E$ denote an adaptation request. In our model we may specify process

$$\mathsf{if}\ \mathsf{arrive}(\texttt{loc}, \texttt{upd}_E)\ \mathsf{then}\ \texttt{loc}\big\{\mathsf{case}\ x_1,\ldots,x_m\ \mathsf{of}\ \{(\beta_1^i;\ \cdots;\ \beta_m^i) : P_i\}_{i\in I}\big\}\ \mathsf{else}\ Q$$

which expresses an adaptation policy that checks for the arrival of a request $\texttt{upd}_E$ for location $\texttt{loc}$. If such a request is detected in the queue of $\texttt{loc}$ then the update process in (2) is triggered; otherwise, process Q is executed. Let us write $\mu\,\mathcal{X}.P$ to denote a recursive process. The previous update process can be easily modified to express a *persistent* adaptation policy that uses recursion to continuously check for request $\texttt{upd}_E$:

$$\mu\,\mathcal{X}.\mathsf{if}\ \mathsf{arrive}(\texttt{loc}, \texttt{upd}_E)\ \mathsf{then}\ \texttt{loc}\big\{\mathsf{case}\ x_1,\ldots,x_m\ \mathsf{of}\ \{(\beta_1^i;\ \cdots;\ \beta_m^i) : P_i\}_{i\in I}\big\}\ \mathsf{else}\ \mathcal{X}$$

Contributions. We present a process model for session-based concurrency and run-time adaptation based on five constructs: located processes, typeful update processes, session monitors, adaptation signals, and location queues. While located processes are inherited from our previous work [DP15], the use of the other constructs for run-time adaptation is new to this presentation. We equip this model with a session type system that ensures the following key properties:

- *Safety* Well-typed programs do not exhibit communication errors (e.g., mismatched messages).
- *Consistency* Well-typed programs do not allow adaptation actions that disrupt already established session protocols.

Safety is the usual guarantee expected from any session type discipline. In contrast, consistency is a guarantee peculiar to our model: it relates the behavior of the adaptation mechanisms with the preservation of prescribed typed interfaces. An example of inconsistent behavior is the interaction between $\texttt{loc}\big[P\big]$ and $\texttt{loc}\{\mathbf{0}\}$ which, as motivated above, discards the running sessions in P. We prove that well-typed programs are safe and consistent: this ensures that specified session protocols are respected, while forbidding incautious adaptation steps that could disrupt the session behavior of interacting partners.

In the *first part* of the paper we consider a typed process model for *binary sessions*, as our goal is to develop our approach to event-based adaptation in a representative and well-established setting. Nevertheless, the key ingredients of our approach are sufficiently simple to be expressed in more general scenarios. Hence, in the *second part* of the paper, as a proof of concept, we describe a generalization of our approach to the case of *multiparty, asynchronous* communication. This generalization builds upon the framework developed in [CDV15], which we briefly describe next.

The framework in [CDV15] uses global types (or *choreographies*) and *monitored processes* as main ingredients. A global type abstracts a sequence of communications between two or more participants. By projecting a global type onto each participant, one obtains a series of *local types*. In [CDV15], such local types are used as *monitors* for the processes that implement each partner: a monitor enables the visible actions of the process. The association of a monitor and a process is called a *monitored process*. The whole choreography is implemented as a collection of monitored processes, or *network*. By considering a network together with a *global state* (data/values used to plan adaptation steps), one then obtains a *system*.

The approach in [CDV15] defines a separation between the global type, the monitors (local types), and the processes. Adaptation is specified at the level of global types, which include an explicit signal (called *flag*) that defines a form of *anticipated adaptation* between all participants of the choreography. As a result of this coordinated signal, an adaptation function determines a new choreography for the system, relying on the global state. Such a choreography determines new implementations for all partners, using a centralized repository of typed processes called *collection*.

We extend the framework of [CDV15] with typeful update processes and event-based constructs. As a result, we obtain an alternative framework in which adaptation is not specified at level of global types (unlike [CDV15]); instead, we have events that handle *internal* and *external* adaptation requests:

- Internal adaptation requests are initiated by a participant that announces all other participants that the current global protocol should be abandoned, and that a new global protocol should be adopted instead.
- External adaptation requests are meant to dynamically update or upgrade the local implementation of a single participant.

To express internal adaptation requests, processes may explicitly output a message whose communication object denotes a choreography. This message can be accessed by all protocol participants (using the shared queue) so to determine their updated behavior. Networks in our model are distributed collections of locations, each containing a monitored process, a local collection of processes, and a queue. This queue is useful to model external adaptation requests which, as described above, are targeted to a particular participant. As a result, we obtain a setting in which internal adaptation is no longer anticipated at the level of types (therefore improving [CDV15]) and concerns all participants of the choreography, while external adaptation requests focus on a single participant.

Organization. The rest of the paper is organized as follows:

- Section 2 illustrates session-based concurrency, our approach and contributions by means of examples.
- Section 3 presents our process model of binary sessions with constructs for run-time adaptation. Then, Sect. 4 presents our session type system, which ensures safety and consistency for processes with adaptation mechanisms (Theorem 4.11).
- In Sect. 5 we generalize our approach to multiparty communications, defined as a variant of the framework given in [CDV15]. We show communication safety and consistency of adaptation steps (Theorem 5.19).
- In Sect. 6 we discuss related works and in Sect. 7 we collect some concluding remarks.

The appendix collects omitted definitions and proofs. This work is an extended version of the paper [DP16]. Here we offer additional explanations and proofs of technical results. Moreover, the approach to adaptation for multiparty sessions (Sect. 5), not presented in [DP16], is a new contribution.

2. Our approach, by example

In this section we illustrate our approach and contributions by means of an example, the *buyer-seller protocol*.

2.1. Binary sessions

Background. In Sects. 3 and 4, we consider binary session types, as defined in [HVK98]:

$\alpha, \beta ::=$	$?(T).\beta$	input value of type T, continue as β
$\mid$	$!(T).\beta$	output value of type T, continue as β
$\mid$	$\&\{n_1{:}\alpha_1, \ldots, n_m{:}\alpha_m\}$	branching (external choice)
$\mid$	$\oplus\{n_1{:}\alpha_1, \ldots, n_m{:}\alpha_m\}$	selection (internal choice)
$\mid$	$\mu\, t.\alpha$	recursive session
$\mid$	t	type variable
$\mid$	$\mathbf{end}$	terminated session

where T stands for basic types (e.g., booleans, integers) and session types α. Also, $n_1, \ldots, n_m$ denote *labels*.

To illustrate session types, consider two participants, a buyer and a seller, which interact as follows. First, buyer sends to seller the name of an item and seller replies back with its price. Then, depending on the amount, buyer may choose to add the item to her shopping cart or to close the transaction. In the latter case the protocol ends. In the former case, buyer must further choose a paying method. From buyer's perspective, this protocol may be described by the session type

$$\alpha_{\mathtt{buy}} = !\mathsf{item}.?\mathsf{amnt}.\alpha_{\mathtt{pay}}$$

where item and amnt are base types and

$$\alpha_{\mathtt{pay}} = \oplus\{addItem : \oplus\{ccard : \alpha_{\mathtt{cc}}\,,\ payp : \alpha_{\mathtt{pp}}\}\,,\ cancel : \mathbf{end}\}.$$

Thus, session type $\alpha_{\mathtt{buy}}$ says that protocol $\alpha_{\mathtt{pay}}$ may only be enabled after sending a value of type item and receiving a value of type amnt. Also, $addItem$, $ccard$, $payp$, and $cancel$ denote labels in the internal choice. Types $\alpha_{\mathtt{cc}}$ and $\alpha_{\mathtt{pp}}$ denote the behavior of each payment method. Following the protocol abstracted by $\alpha_{\mathtt{buy}}$, code for buyer may be specified as a π-calculus process. Processes P and R below give two different specifications for buyer:

$$P = \overline{x}(\mathsf{book}).x(a).\mathsf{if}\ a < 50\ \mathsf{then}\ x \triangleleft addItem;\ x \triangleleft ccard;\ P_1\ \mathsf{else}\ x \triangleleft cancel;\ \mathbf{0}$$
$$R = \overline{x}(\mathsf{game}).x(b).\mathsf{if}\ b < 80\ \mathsf{then}\ x \triangleleft addItem;\ x \triangleleft payp;\ R_1\ \mathsf{else}\ x \triangleleft cancel;\ \mathbf{0}$$

Thus, although both P and R implement $\alpha_{\mathtt{buy}}$, their behavior is rather different, for they purchase different items using different payment methods (which are abstracted by processes P_1 and R_1).

Let us now analyze the situation for the seller. To ensure protocol compatibility and absence of communication errors, the session type for seller, denoted $\beta_{\mathtt{sel}}$, should be *dual* to $\alpha_{\mathtt{buy}}$. Intuitively, duality decrees that every action from buyer must be matched by a complementary action from seller, e.g., every output in $\alpha_{\mathtt{buy}}$ is matched by an input in $\beta_{\mathtt{sel}}$. Formally, this duality is denoted $\alpha_{\mathtt{buy}} \perp_{\mathtt{c}} \beta_{\mathtt{sel}}$ (cf. Definition A.4). Thus, in our example, we let $\beta_{\mathtt{sel}} = ?\mathsf{item}.!\mathsf{amnt}.\beta_{\mathtt{pay}}$, where

$$\beta_{\mathtt{pay}} = \&\{addItem : \&\{ccard : \beta_{\mathtt{cc}}\,,\ payp : \beta_{\mathtt{pp}}\}\,,\ cancel : \mathbf{end}\}$$

and $\beta_{\mathtt{cc}}$ and $\beta_{\mathtt{pp}}$ denote the duals of $\alpha_{\mathtt{cc}}$ and $\alpha_{\mathtt{pp}}$, respectively. A process implementation Q for $\beta_{\mathtt{sel}}$ is the following:

$$Q = y(i).\overline{y}(price(i)).y \triangleright \{addItem : y \triangleright \{ccard : Q_1 \,[]\, ppal : Q_2\} \,[]\, cancel : \mathbf{0}\}$$

where $price$ stands for an auxiliary function.

The interaction of P and Q is defined using *session initialization* constructs: process $\overline{u}(x{:}\alpha).P$ denotes the *request* of a session of type α; dually, $u(x{:}\alpha).P$ denotes the *acceptance* of a session of type α. In both cases, u denotes a *(shared) name* used for synchronization. In our example, we may have

$$Sys = \overline{u}(x : \alpha_{\mathtt{buy}}).P \mid u(y : \beta_{\mathtt{sel}}).Q \longrightarrow (\nu\kappa)(P[\kappa^+/x] \mid Q[\kappa^-/y]) = S_0 \tag{3}$$

Thus, upon synchronization on u, a new session κ is established. Intuitively, in process S_0 session κ is "split" into two *session channels* (or *endpoints*) κ^+ and κ^-: we write + and − to denote their opposing *polarities* [GH05], which make their complementarity manifest. The restriction $(\nu\kappa)$ covers both channels, thus ensuring an interference-free medium for executing the session protocols described by α and β.

Session types may exploit a *subtyping* relation [GH05] to express useful relationships between protocol specifications. For instance, the type below defines a new payment method for seller:

$$\beta_{\texttt{gift}} = \&\{addItem : \&\{giftc : \beta_{\texttt{gc}}, ccard : \beta_{\texttt{cc}}, payp : \beta_{\texttt{pp}}\}, cancel : \mathbf{end}\} \tag{4}$$

Intuitively, $\beta_{\texttt{gift}}$ *extends* $\beta_{\texttt{pay}}$ with an alternative on label $giftc$. It is safe to use an implementation for $\beta_{\texttt{gift}}$ wherever an implementation for $\beta_{\texttt{pay}}$ is required. The *safe substitution* principle that connects $\beta_{\texttt{gift}}$ and $\beta_{\texttt{pay}}$ is formalized by a subtyping, denoted $\leq_{\texttt{C}}$ (cf. Definition A.4). This way, e.g., we have $\beta_{\texttt{pay}} \leq_{\texttt{C}} \beta_{\texttt{gift}}$.

Our process model and approach. We are interested in expressing and reasoning about the run-time modification of session-typed processes such as P and Q above. Such modifications may be desirable as a reaction to exceptional run-time conditions (say, an error) or to implement unanticipated requirements.

Continuing our example, consider again process Sys (cf. (3) above). In our process model, detailed in Sect. 3, we have the following reduction step:

$$Sys \longrightarrow (\nu\kappa)(P[\kappa^+/x] \mid \kappa^+\lfloor\alpha_{\texttt{buy}}\rfloor \mid Q[\kappa^-/y] \mid \kappa^-\lfloor\beta_{\texttt{buy}}\rfloor) = S$$

Session establishment creates *session monitors* $\kappa^+\lfloor\alpha_{\texttt{buy}}\rfloor$ and $\kappa^-\lfloor\beta_{\texttt{buy}}\rfloor$, processes that maintain the protocol state for the session at κ^+ and κ^-, respectively. These monitors are essential to implement run-time adaptation policies that handle processes with running sessions. Reduction may proceed as follows:

$$\begin{aligned} S \longrightarrow (\nu\kappa)(\kappa^+(a).\mathsf{if}\ a < 50\ \mathsf{then}\ \kappa^+ \triangleleft addItem;\ \kappa^+ \triangleleft ccard;\ P_1\ \mathsf{else}\ \kappa^+ \triangleleft cancel;\ \mathbf{0} \mid \kappa^+\lfloor ?\mathsf{amnt}.\,\alpha_{\texttt{pay}}\rfloor \mid \\ \overline{\kappa^-}(price(\mathsf{book})).\kappa^- \triangleright \{addItem : \kappa^- \triangleright \{ccard : Q_1 \,[\!]\, ppal : Q_2\} \,[\!]\, cancel : \mathbf{0}\} \mid \kappa^-\lfloor !\mathsf{amnt}.\,\beta_{\texttt{pay}}\rfloor) = S' \end{aligned}$$

Observe how the session monitors evolve together with the process, capturing the evolution of the protocol state. In this example, they are useful to record the fact that the buyer part of S' realizes the type $?\mathsf{amnt}.\,\alpha_{\texttt{pay}}$, whereas the seller part realizes the type $!\mathsf{amnt}.\,\beta_{\texttt{pay}}$. Suppose that we wish to modify at run-time the part of S' realizing the buyer behavior. To preserve protocol correctness, a candidate new implementation must conform, up to $\leq_{\texttt{C}}$, to the type $?\mathsf{amnt}.\,\alpha_{\texttt{pay}}$; a process realizing any other type will fail to safely interact with the part of S' implementing the seller.

We now illustrate how located and update processes, distinctive of our model, could be used in the buyer-seller scenario. As a simple example, process W below

$$W = \texttt{sys}\big[\,\texttt{buyer}\big[\overline{u}(x{:}\alpha_{\texttt{buy}}).P\big] \mid \texttt{seller}\big[u(y{:}\beta_{\texttt{sel}}).Q\big]\,\big] \tag{5}$$

represents an explicitly distributed variant of process Sys given in (3): the two partners now reside in different locations, namely `buyer` and `seller`; location `sys` encloses the whole system. Our process model allows us to define an update process that depends on the current protocol state of the two channels at location `sys`:

$$\texttt{sys}\left\{\mathsf{case}\ x, y\ \mathsf{of}\ \begin{cases}(\alpha_{\texttt{buy}};\ \beta_{\texttt{sel}}) & : \texttt{buyer}\big[R\big] \mid \texttt{seller}\big[Q\big] \\ (\alpha_{\texttt{pay}};\ \beta_{\texttt{pay}}) & : \texttt{buyer}\big[P^*\big] \mid \texttt{seller}\big[Q^*\big]\end{cases}\right\} \tag{6}$$

This update process defines two alternatives for adaptation:

1. If the current types for the two channels at `sys` are $\alpha_{\texttt{buy}}$ and $\beta_{\texttt{sel}}$ (i.e., the protocol has just been established) then only the buyer is updated—its new behavior will be given by process R.
2. If the current types for the two channels at `sys` are $\alpha_{\texttt{pay}}$ and $\beta_{\texttt{pay}}$ (i.e., both item and price information have been already exchanged) then new implementations P^* and Q^* are installed in the respective locations.

Session types are used to guide update processes, but also to ensure that the ensuing process alternatives are sound. The type system we develop in Sect. 4 ensures that process $\texttt{buyer}\big[R\big] \mid \texttt{seller}\big[Q\big]$ (the first alternative) has exactly two free session channels x and y, with session types $\alpha_{\texttt{buy}}$ and $\beta_{\texttt{sel}}$, respectively. A similar check applies to process $\texttt{buyer}\big[P^*\big] \mid \texttt{seller}\big[Q^*\big]$ and $\alpha_{\texttt{pay}}$ and $\beta_{\texttt{pay}}$.

As a simple example of the event-based constructs, we may consider a variant of the process given in Sect. 1, which defines a conditional (and persistent) adaptation policy:

$$U = \mu\,\mathcal{X}.\mathsf{if}\ \mathsf{arrive}(\texttt{sys}, \texttt{upd}_E)\ \mathsf{then}\ U_{\texttt{bs}}\ \mathsf{else}\ \mathcal{X} \tag{7}$$

Above, $U_{\mathtt{bs}}$ denotes the update process in (6). Intuitively, the process in (7) persistently checks if an adaptation message $\mathtt{upd}_E$ has arrived to the queue of location $\mathtt{sys}$, which we denote $\mathtt{sys}\lfloor\widetilde{r}\rfloor$. (We shall write r, r_0, r_1, ... to range over adaptation messages.) Notice that such a message may be issued by process W itself or by its context, using an adaptation signal, denoted $\overline{\mathtt{loc}}(\mathtt{upd}_E)$. We therefore envision systems in which communication actions, adaptation policies, and event-based constructs concurrently interact. In our example, we could have the process

$$W \mid U \mid \mathtt{sys}\lfloor r_0 \cdot \widetilde{r}\rfloor$$

which triggers the adaptation of location $\mathtt{sys}$ if $r_0 = \mathtt{upd}_E$; otherwise, the communication behavior of W would take place.

2.2. Multyparty sessions

Background. In Sect. 5 we consider multiparty sessions following the approach in [CDV15], which relies on *global types*, *monitors*, *processes*, *networks*, and *systems*.

In [CDV15], global types describe communications and explicit adaptation signals; monitors, derived from global types, describe the protocol from the perspective of a single participant. Processes only specify communicating behavior; they are associated to monitors that regulate their behavior and contain information on senders/receivers. Each protocol participant is then defined as a monitored process, a process tied with a monitor. Networks are collections of monitored processes, together with constructs for starting new global protocols and queues for handling asynchronous communication. A system is the composition of a network with a global state, a collection of data that may influence adaptation.

We introduce some notational conventions. Below, $\mathsf{p}, \mathsf{q}, \ldots$ denote participants and Π represents a set of participants. Also, $l_1, l_2, \ldots$ denote *labels*; $S_1, S_2, \ldots$ denote carried types (e.g., $\mathtt{Bool}$, $\mathtt{Int}$); and $\lambda_1, \lambda_2, \ldots$ denote *adaptation flags*. Global types abstract sequences of broadcast-like communications from a single participant to a set of participants; such communications include usual labeled messages and also explicit adaptation signals (flags).

$$\begin{array}{lll} \mathsf{G}, \mathsf{G}' ::= & \mathsf{p} \to \Pi : \{l_i(S_i).\mathsf{G}_i\}_{i\in I} & \text{directed communication with labeled alternatives} \\ \mid & \mathsf{p} \to \Pi : \{\lambda_i\}_{i\in I} & \text{adaptation flag} \\ \mid & \mathsf{end} & \text{terminated protocol} \end{array}$$

The set of *monitors* is defined as follows:

$$\begin{array}{lll} \mathsf{M} ::= & \mathsf{p}?\{l_i(S_i).\mathsf{M}_i\}_{i\in I} \mid \Pi!\{l_i(S_i).\mathsf{M}_i\}_{i\in I} & \text{communication} \\ \mid & \mathsf{p}?\{\lambda_i\}_{i\in I} \mid \Pi!\{\lambda_i\}_{i\in I} & \text{adaptation} \\ \mid & \mathsf{end} & \text{terminated choreography} \end{array}$$

Formally, global types and monitors are related via a *projection* function that defines how a global type can be decomposed into monitors. Projection also determines valid global types: a global type is said to be *well-formed* if (i) its projections are defined for all participants, and if (ii) exchanges of adaptation flags involve all participants declared in the global type. The syntax of processes is as follows:

$$\begin{array}{lll} P ::= & c?l(x).P \mid c!l(e).P \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid P + P \mid \mu\,\mathcal{X}.P \mid \mathcal{X} \mid \mathbf{0} \\ \mid & \mathsf{op}.P \mid c?l(\lambda, \mathtt{T}).P \mid c!l(\lambda(F), \mathtt{T}).P \end{array}$$

Each process owns a unique *channel* c. At run-time, channels specify a session channel and the identity of the participant. Most constructs are standard; processes $\mathsf{op}.P$, $c?l(\lambda, \mathtt{T}).P$, and $c!l(\lambda(F), \mathtt{T}).P$ deserve additional explanations:

- $\mathsf{op}.P$ denotes an operation on the global state; as such, it influences the behavior of future adaptation actions.
- process $c?l(\lambda, \mathtt{T}).P$ receives an adaptation flag λ and has a continuation of type $\mathtt{T}$.
- process $c!l(\lambda(F), \mathtt{T}).P$ specifies an *adaptation function* F which, in combination with the global state, will determine a new global type.

Given a monitor M and a process P, a monitored process is denoted $\mathsf{M}[\![P]\!]$. Adaptation actions manifest at the level of networks; they are realized directly over monitors but indirectly over processes: the modification of the global type of the system results into a new set of monitors associated to processes; if a current process does not "fit" its new monitor, then it must be replaced with a different process, obtained from a centralized collection of pairs $(P, \mathtt{T})$, denoted $\mathcal{P}$, a parameter of the framework. In [CDV15], this collection is assumed to be complete:

it contains processes for all conceivable monitors. Formally, the fit between processes and monitors is defined as an adequacy relation, denoted $\propto$, which uses subtyping to relate the type of processes and monitors.

The semantics of processes is given in terms of a labeled transition system (LTS) in which actions describe a session and the current participant. These actions are used to enforce the consistency between a process and its associated monitor, as formally captured by the semantics of networks. This semantics describes how each participant reacts to an adaptation request (i.e., a new global type) issued by another participant. Depending on whether or not the current process is adequate to the new monitor then the process implementation is kept or replaced, as described before.

The semantics of systems specifies how a single participant communicates a new global type for the whole system. Indeed, as the selection of the new global type depends on the adaptation function F and the global state, the new global type can only be described at the level of systems, rather than at the level of networks. This selection is communicated to participants using the shared queue, and locally handled at the level of networks, as just explained. Notice the new global type may determine a new set of participants, and so some of the current participants may be excluded as a result of adaptation. The new global type may also determine a new implementation for the participant invoking global reconfiguration; here again adequacy is used to determine whether a process reconfiguration is required or not.

Our process model and approach. Our example in Sect. 2.1 can be extended to the multyparty scenario by adding a third participant (*postal service*) to the conversation. The extended protocol is as follows: the buyer b asks for an item to the seller s, who replies with its price. Then, b confirms its order and pays it by sending to s her credit card number and to the postal service p her address. Finally, s finalizes the order by sending the name of the item to p. As a global type, this protocol may be expressed as follows:

$$\begin{aligned} \mathsf{G} ::= \mathsf{b} \to \mathsf{s} : it(item).\, \mathsf{s} \to \mathsf{b} : price(int).\\ \mathsf{b} \to \mathsf{s} : ok(bool).\, \mathsf{b} \to \mathsf{s} : card(int).\\ \mathsf{b} \to \mathsf{p} : ad(string).\mathsf{s} \to \mathsf{p} : itname(item) \end{aligned}$$

In the multiparty scenario, our located process adopt a slightly different meaning than in the binary setting. Each location is associated to only one participant in the conversation and contains a local collection of typed processes $\mathcal{P}$ that represents possible programs to be run upon session establishment. That is, the collection assumed global in [CDV15] is now distributed to the local participants. The idea behind this choice is that a location may correspond to an autonomous "computing entity" possibly running distinct programs, as in, e.g., a smartphone where applications are installed and can be executed depending on the need. Thus the three participants above can be implemented in the distinct locations $\mathtt{loc}_\mathsf{b}$, $\mathtt{loc}_\mathsf{s}$, and $\mathtt{loc}_\mathsf{p}$:

$$\text{Buyer: } \mathtt{loc}_\mathsf{b}\left[\mathcal{P}_\mathsf{b};\ \mathtt{loc}_\mathsf{b}\lfloor\epsilon\rfloor\right] \quad \text{Seller: } \mathtt{loc}_\mathsf{s}\left[\mathcal{P}_\mathsf{s};\ \mathtt{loc}_\mathsf{s}\lfloor\epsilon\rfloor\right] \quad \text{Postal Service: } \mathtt{loc}_\mathsf{p}\left[\mathcal{P}_\mathsf{p};\ \mathtt{loc}_\mathsf{p}\lfloor\epsilon\rfloor\right]$$

where, for each $\mathsf{q} \in \{\mathsf{b}, \mathsf{s}, \mathsf{p}\}$, $\mathtt{loc}_\mathsf{q}\lfloor\epsilon\rfloor$ is an event queue used to signal the presence of an adaption routine; it has the same behavior as in the binary case. The collection $\mathcal{P}_\mathsf{q}$ contains an implementation of the local protocol of each participant (obtained projecting the global protocol into each participant). We therefore have the following processes in the respective local collections:

$$\begin{aligned} P_\mathsf{b} &::= y!it(item).y?price(x).y!ok(true).y!card(number).y!ad(address)\\ P_\mathsf{s} &::= y?it(x_1).(\mathsf{if}\ (x_1 = item1)\ \mathsf{then}\ y!price(1).y?ok(x_2).y?card(x_3).y!itname(x_1)\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ y!price(2).y?ok(x_2).y?card(x_3).y!itname(x_1)\\ P_\mathsf{p} &::= y?ad(x_1).y?itname(x_2) \end{aligned}$$

In our proof of concept we define both external and internal adaptation. An internal adaptation routine is started by one of the participants who warns the others that a new protocol should be followed. External adaptation requests can be of two forms: they may either *upgrade* a process in the local collection or *update* the participant's current implementation using a type-directed check, following the update processes introduced for binary sessions.

Table 1. Syntax of expressions and processes for binary communications. Annotation α denotes a session type. Standard constructs are aligned to the left; novel constructs are aligned to the right

$e ::=$	$v \mid x, y, z \mid k = k \mid a = a$	expressions	$\mid$	$\mathsf{arrive}(\mathtt{loc}, r)$	arrival predicate
$P ::=$	$\overline{u}(x : \alpha).P$	session request	$\mid$	$\mathtt{loc}[P]$	located process
$\mid$	$u(x : \alpha).P$	session accept	$\mid$	$\mathtt{loc}\{\mathsf{case}\ x_1, \ldots, x_n\ \mathsf{of}\ \{(\beta_i^1; \cdots; \beta_i^n) : P_i\}_{i \in I}\}$	update process
$\mid$	$\overline{k}(e).P$	session output	$\mid$	$k\lfloor\alpha\rfloor$	session monitor
$\mid$	$k(x).P$	session input	$\mid$	$\mathtt{loc}\lfloor\widetilde{r}\rfloor$	location queue
$\mid$	$k \lhd n;\ P$	session selection	$\mid$	$\overline{\mathtt{loc}}(r)$	adaptation signal
$\mid$	$k \rhd \{n_i : P_i\}_{i \in I}$	session branching			
$\mid$	$\mathsf{close}(k).P$	session closure			
$\mid$	$\mu\, \mathcal{X}.P$	recursion			
$\mid$	$\mathcal{X}$	recursion variable			
$\mid$	$P \mid \mathrm{P}$	composition			
$\mid$	$(\nu\kappa)P$	channel hiding			
$\mid$	$(\nu u)P$	name hiding			
$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$	conditional			
$\mid$	$\mathbf{0}$	inaction			

As an example in the buyer-seller-postal service scenario, we can imagine that after having chosen an item, the buyer asks the other participants to change the protocol, as she wishes to change the payment method from credit card to bank transfer. This adaptation routine will cause the buyer and the postal service to change their implementations, while the postal service will remain unchanged. Moreover, a new participant (the bank k) will be added to the conversation. The new global protocol will then be:

$$\begin{aligned}\mathsf{G}' ::= {} & \mathsf{s} \to \mathsf{b} : \mathit{iban}(\mathit{string}).\mathsf{b} \to \mathsf{k} : \mathit{iban}(\mathit{string}).\\ & \mathsf{b} \to \mathsf{p} : \mathit{ad}(\mathit{string}).\mathsf{s} \to \mathsf{p} : \mathit{itname}(\mathit{item})\end{aligned}$$

and the idea is that current communications are stopped, and the current active process is substituted with another one that implements the new protocol. The semantics for networks would lead to the following adapted process:

$$\begin{aligned}&(\nu\kappa)(\mathtt{loc}_\mathsf{b}[y!\mathit{card}(\mathit{number}).y!\mathit{ad}(\mathit{address});\ \mathcal{P}_\mathsf{b};\ \mathtt{loc}_\mathsf{b}\lfloor\epsilon\rfloor] \mid\\ &\qquad \mathtt{loc}_\mathsf{s}[y?\mathit{card}(x).y!\mathit{itname}(x_1);\ \mathcal{P}_\mathsf{s};\ \mathtt{loc}_\mathsf{s}\lfloor\epsilon\rfloor] \mid \mathtt{loc}_\mathsf{p}[y?\mathit{ad}(x_1).y?\mathit{itname}(x_2);\ \mathcal{P}_\mathsf{p};\ \mathtt{loc}_\mathsf{p}\lfloor\epsilon\rfloor])\\ &\longrightarrow\\ &(\nu\kappa)(\mathtt{loc}_\mathsf{b}[y?\mathit{iban}(x).y!\mathit{iban}(x).y!\mathit{ad}(\mathit{address});\ \mathcal{P}_\mathsf{b};\ \mathtt{loc}_\mathsf{b}\lfloor\epsilon\rfloor] \mid \mathtt{loc}_\mathsf{k}[y?\mathit{iban}(x);\ \mathcal{P}_\mathsf{k};\ \mathtt{loc}_\mathsf{k}\lfloor\epsilon\rfloor]\\ &\qquad \mathtt{loc}_\mathsf{s}[y!\mathit{iban}(\mathit{ibstr}).y!\mathit{itname}(x_1);\ \mathcal{P}_\mathsf{s};\ \mathtt{loc}_\mathsf{s}\lfloor\epsilon\rfloor] \mid \mathtt{loc}_\mathsf{p}[y?\mathit{ad}(x_1).y?\mathit{itname}(x_2);\ \mathcal{P}_\mathsf{p};\ \mathtt{loc}_\mathsf{p}\lfloor\epsilon\rfloor])\end{aligned}$$

We shall revisit this scenario in Sect. 5—see Example 5.8.

3. Event-based adaptation for binary communications

Syntax. We rely on base sets for *names*, ranged over by $u, a, b \ldots$; *(session) channels*, ranged over by $k, \kappa^p, \ldots$, with *polarity* $p \in \{+, -\}$; *labels*, ranged over by $n, n', \ldots$; and *variables*, ranged over by $x, y, \ldots$. *Values*, ranged over $v, v', \ldots$, may include booleans (written `false` and `true`), integers, names, and channels. We use r to range over *adaptation messages*, such as $\mathtt{upd}_E$ in (7) above. We use $\widetilde{\cdot}$ to denote finite sequences. Thus, e.g., $\widetilde{x}$ is a sequence of variables $x_1, \ldots, x_n$. We use ϵ to denote the empty sequence.

Table 1 reports the syntax of expressions and processes. Processes include usual constructs for input, output, and labeled choice. Common forms of recursion, parallel composition, conditionals, and restriction are also included. Constructs for session establishment are annotated with a session type α, which is useful in derived static analyses. A prefix for closing a session, inherited from [DP15], is convenient to better structure specifications. Variable x is bound in processes $\overline{u}(x{:}\alpha).P$, $u(x{:}\alpha).P$, and $k(x).P$. Binding for name and channel restriction is as usual. Also, recursion variable $\mathcal{X}$ is bound in process $\mu\, \mathcal{X}.P$. Given a process P, its sets of free/bound channels, names, variables, and recursion variables—noted $\mathsf{fc}(P)$, $\mathsf{fn}(P)$, $\mathsf{fv}(P)$, $\mathsf{fpv}(P)$, $\mathsf{bc}(P)$, $\mathsf{bn}(P)$, $\mathsf{bv}(P)$, and $\mathsf{bpv}(P)$, respectively—are as expected. We always rely on usual notions of α-conversion and (capture-avoiding) substitution, denoted $[k/x]$ (for channels) and $[P/\mathcal{X}]$ (for processes). We write $[k_1, \ldots, k_n/x_1, \ldots, x_n]$ to stand for an n-ary simultaneous substitution. Processes without free variables or free channels are called *programs*.

Up to here, the language is essentially a synchronous π-calculus with sessions [HVK98]. As motivated in the Introduction, building upon *locations* $\mathtt{loc}, \mathtt{l}_1, \mathtt{l}_2, \ldots$, we consider the following novel process constructs for run-time adaptation:

- *Located processes* denoted $\mathtt{loc}\big[P\big]$;
- *Update processes* denoted $\mathtt{loc}\big\{\mathsf{case}\ x_1, \ldots, x_n\ \mathsf{of}\ \{(\beta_i^1;\ \cdots;\ \beta_i^n) : Q_i\}_{i\in I}\big\}$;
- *Session monitors* denoted $k\lfloor\alpha\rfloor$, for a session type α;
- *Location queues* denoted $\mathtt{loc}\lfloor\widetilde{r}\rfloor$; and
- *Adaptation signals* denoted $\overline{\mathtt{loc}}(r)$.

Moreover, expressions now include the *arrival predicate* $\mathsf{arrive}(\mathtt{loc}, r)$.

We now comment on these elements. *Located processes* and *update processes* have been already discussed. Here we just remark that update processes are assumed to refer to at least one variable x_i and to offer at least one alternative Q_i. Also, variables $x_1, \ldots, x_n$ are bound in $\mathtt{loc}\big\{\mathsf{case}\ x_1, \ldots, x_n\ \mathsf{of}\ \{(\beta_i^1;\ \cdots;\ \beta_i^n) : Q_i\}_{i\in I}\big\}$; this process is often abbreviated as $\mathtt{loc}\big\{\mathsf{case}\ \widetilde{x}\ \mathsf{of}\ \{(\beta_i^1;\ \cdots;\ \beta_i^n) : Q_i\}_{i\in I}\big\}$. Update processes generalize the `typecase` introduced in [Kou12], which defines a case-like choice based on a single channel; in contrast, to specify adaptation for locations with multiple open sessions, our update processes define type-directed checks over one or more channels.

Update processes go hand-in-hand with *monitors*, run-time entities which keep the current protocol state at a given channel. We write $\kappa^p\lfloor\alpha\rfloor$ to denote the monitor that stores the protocol state α for channel κ^p. In [Kou12], a similar construct is used to store in-transit messages in asynchronous communications. For simplicity, here we consider synchronous communication; monitors store only the current protocol state. This choice is aligned with our goal of identifying the elements from the eventful session framework that are essential to run-time adaptation (cf. Remark 4.12).

Location queues, not present in [Kou12], handle adaptation requests, modeled as a possibly empty sequence of messages $\widetilde{r}$. Location queues enable us to give a unified treatment to adaptation requests. Given $\mathtt{loc}\lfloor\widetilde{r}\rfloor$, it is worth observing that messages $\widetilde{r}$ are not related to communication as abstracted by session types. This represents the fact that we handle adaptation requests and structured session exchanges as orthogonal issues. An *adaptation signal* $\overline{\mathtt{loc}}(r)$ enqueues request r into the location queue of `loc`. To this end, as detailed below, the operational semantics defines synchronizations between adaptation signals and location queues. To connect run-time adaptation and communication, our language allows the coupling of update processes with the *arrival predicate on locations*, denoted $\mathsf{arrive}(\mathtt{loc}, r)$. Inspired by the `arrive` predicate in [Kou12], this predicate detects if a message r has been placed in the queue of `loc`.

Our process language embodies several concerns related to run-time adaptation: using adaptation signals and location queues we may specify *how* an adaptation request is issued; arrival predicates enable us to specify *when* adaptation will be handled; using update processes and monitors we may specify *what* is the goal of an adaptation event.

Semantics. The semantics of our language is given by a *reduction semantics*, the smallest relation generated by the rules in Table 2. We write $P \longrightarrow P'$ for the reduction from P to P'. Reduction relies on a standard notion of structural congruence, denoted $\equiv$ (see Appendix A). It also relies on *evaluation* and *location* contexts:

$$\mathsf{E} ::= - \mid \overline{k}(-).P \mid \mathsf{if}\ -\ \mathsf{then}\ P\ \mathsf{else}\ Q \qquad C, D ::= - \mid \mathtt{loc}\big[C \mid P\big]$$

Given $C\{-\}$ (resp. $\mathsf{E}[-]$), we write $C\{P\}$ (resp. $\mathsf{E}[e]$) to denote the process (resp. expression) obtained by filling in occurrences of hole $-$ in C with P (resp. in E with e).

Table 2. Reduction semantics for binary processes. Above, α and β denote session types

$$\langle \text{R:OPEN} \rangle \quad \frac{\alpha \perp_{\mathsf{C}} \beta}{C\{u(x:\alpha).P\} \mid D\{\overline{u}(y:\beta).Q\} \longrightarrow (\nu\kappa)\big(C\{P[\kappa^p/x] \mid \kappa^p\lfloor\alpha\rfloor\} \mid D\{Q[\kappa^{\overline{p}}/y] \mid \kappa^{\overline{p}}\lfloor\beta\rfloor\}\big)}$$

$$\langle \text{R:COM} \rangle \quad \frac{}{C\{\overline{\kappa}^p(v).P \mid \kappa^p\lfloor!(T).\alpha\rfloor\} \mid D\{\kappa^{\overline{p}}(x).Q \mid \kappa^{\overline{p}}\lfloor?(T).\beta\rfloor\} \longrightarrow C\{P \mid \kappa^p\lfloor\alpha\rfloor\} \mid D\{Q[v/x] \mid \kappa^{\overline{p}}\lfloor\beta\rfloor\}}$$

$$\langle \text{R:SEL} \rangle \quad \frac{i \in J}{C\{\kappa^p \triangleright \{n_j : P_j\}_{j\in J} \mid \kappa^p\lfloor\&\{n_j:\alpha_j\}_{j\in J}\rfloor\} \mid D\{\kappa^{\overline{p}} \triangleleft n_i;\ Q \mid \kappa^{\overline{p}}\lfloor\oplus\{n_j:\beta_j\}_{j\in J}\rfloor\} \longrightarrow C\{P_i \mid \kappa^p\lfloor\alpha_i\rfloor\} \mid D\{Q \mid \kappa^{\overline{p}}\lfloor\beta_i\rfloor\}}$$

$$\langle \text{R:CLO} \rangle \quad \frac{}{C\{\mathsf{close}(\kappa^p).P \mid \kappa^p\lfloor\mathbf{end}\rfloor\} \mid D\{\mathsf{close}(\kappa^{\overline{p}}).Q \mid \kappa^{\overline{p}}\lfloor\mathbf{end}\rfloor\} \longrightarrow C\{P\} \mid D\{Q\}}$$

$$\langle \text{R:UREQ} \rangle \quad \frac{}{C\{\mathtt{loc}\lfloor\widetilde{r}_1\rfloor\} \mid D\{\overline{\mathtt{loc}}(r)\} \longrightarrow C\{\mathtt{loc}\lfloor\widetilde{r}_1 \cdot r\rfloor\} \mid D\{\mathbf{0}\}}$$

$$\langle \text{R:ARR1} \rangle \quad \frac{\widetilde{r} = r_1 \cdot \widetilde{r}_0}{C\{\mathsf{E}[\mathsf{arrive}(\mathtt{loc}, r_1)]\} \mid D\{\mathtt{loc}\lfloor\widetilde{r}\rfloor\} \longrightarrow C\{\mathsf{E}[\mathtt{true}]\} \mid D\{\mathtt{loc}\lfloor\widetilde{r}_0\rfloor\}}$$

$$\langle \text{R:ARR2} \rangle \quad \frac{(\widetilde{r} = r_2 \cdot \widetilde{r}_0 \wedge r_1 \neq r_2) \vee \widetilde{r} = \epsilon}{C\{\mathsf{E}[\mathsf{arrive}(\mathtt{loc}, r_1)]\} \mid D\{\mathtt{loc}\lfloor\widetilde{r}\rfloor\} \longrightarrow C\{\mathsf{E}[\mathtt{false}]\} \mid D\{\mathtt{loc}\lfloor\widetilde{r}\rfloor\}}$$

$$\langle \text{R:UPD1} \rangle \quad \frac{\mathsf{match}_I(\{\alpha_1,\ldots,\alpha_m\},\ \{\beta_i^1,\ldots,\beta_i^m\}_{i\in I}) = \uparrow}{C\{\mathtt{loc}[P]\} \mid D\Big\{\mathtt{loc}\big\{\mathsf{case}\ x_1,\ldots,x_m\ \mathsf{of}\ \{(\beta_i^1;\ \cdots;\ \beta_i^m) : Q_i\}_{i\in I}\big\}\Big\} \longrightarrow C\{\mathtt{loc}[P]\} \mid D\{\mathbf{0}\}}$$

$$\langle \text{R:UPD2} \rangle \quad \frac{\begin{array}{c}\mathsf{match}_I(\{\alpha_1,\ldots,\alpha_m\},\ \{\beta_i^1,\ldots,\beta_i^m\}_{i\in I}) = l \\ \mathsf{fc}(P) = \{\kappa_1^p,\ldots,\kappa_m^p\} \qquad \forall j \in [1,\ldots,m].(\kappa_j^p\lfloor\alpha_j\rfloor \in P) \qquad \mathsf{barbs}(P;\ Q_l[\kappa_1^p,\ldots,\kappa_m^p/x_1,\ldots,x_m])\end{array}}{C\{\mathtt{loc}[P]\} \mid D\Big\{\mathtt{loc}\big\{\mathsf{case}\ x_1,\ldots,x_m\ \mathsf{of}\ \{(\beta_i^1;\ \cdots;\ \beta_i^m) : Q_i\}_{i\in I}\big\}\Big\} \longrightarrow C\{\mathtt{loc}[Q_l[\kappa_1^p,\ldots,\kappa_m^p/x_1,\ldots,x_m]]\} \mid D\{\mathbf{0}\}}$$

$$\langle \text{R:EVA} \rangle \quad \frac{e \longrightarrow e}{\mathsf{E}[e] \longrightarrow \mathsf{E}[e']} \qquad \langle \text{R:PAR} \rangle \quad \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \langle \text{R:RESN} \rangle \quad \frac{P \longrightarrow P'}{(\nu a)P \longrightarrow (\nu a)P'} \qquad \langle \text{R:RESC} \rangle \quad \frac{P \longrightarrow P'}{(\nu\kappa)P \longrightarrow (\nu\kappa)P'}$$

$$\langle \text{R:STR} \rangle \quad \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q} \qquad \langle \text{R:REC} \rangle \quad \frac{}{\mathsf{rec}\,\mathcal{X}.P \longrightarrow P[\mathsf{rec}\,\mathcal{X}.P/\mathcal{X}]}$$

$$\langle \text{R:IFTRUE} \rangle \quad \frac{}{\mathtt{if\ true\ then}\ P\ \mathtt{else}\ Q \longrightarrow P} \qquad \langle \text{R:IFFALSE} \rangle \quad \frac{}{\mathtt{if\ false\ then}\ P\ \mathtt{else}\ Q \longrightarrow Q}$$

We comment on the reduction rules below. The first four rules formalize session behavior within hierarchies of nested locations. Using duality for session types, denoted $\perp_{\mathsf{C}}$ (cf. Definition A.4), in Rule ⟨R:OPEN⟩ the synchronization on a name u leads to establish a session on fresh channels κ^p and $\kappa^{\overline{p}}$; also, two monitors with the declared session types are created. Duality for polarities p is as expected: $\overline{+} = -$ and $\overline{-} = +$. By construction, monitors are *local*, i.e., they are created in the same contexts in which the session is established. Rule ⟨R:COM⟩ represents communication of a value: we require both complementary prefixes and that the monitors support input and output actions. After reduction, prefixes in processes and monitors are consumed. Similarly, Rule ⟨R:SEL⟩ for labeled choice is standard, augmented with monitors. Rule ⟨R:CLO⟩ formalizes session termination, discarding involved monitors. The monitors in these three rules allow us to track the evolution of active session protocols. The following five rules in Table 2 define our event-based approach to run-time adaptation. Before commenting on them, we require two auxiliary definitions:

Definition 3.1 (*Matching*) Given an index set I of alternatives, session types $\alpha_1,\ldots,\alpha_m$, and an indexed sequence of session types $\{\beta_i^1,\ldots,\beta_i^m\}_{i\in I}$, we define

$$\mathsf{match}_I(\{\alpha_1,\ldots,\alpha_m\},\ \{\beta_i^1,\ldots,\beta_i^m\}_{i\in I}) = \begin{cases} l & \text{if} l \in I \wedge (\forall j \in [1,\ldots,m].\beta_l^j \leq_{\mathsf{C}} \alpha_j) \\ \uparrow & \forall n \in I.(\exists j \in [1,\ldots,m].\ \beta_n^j \not\leq_{\mathsf{C}} \alpha_j) \end{cases}$$

Intuitively, $\mathsf{match}_I(\{\alpha_1,\ldots,\alpha_m\},\ \{\beta_i^1,\ldots,\beta_i^m\}_{i\in I}) = l$ says that the l-th alternative ensures a match (up to subtyping) between types α_j and β_l^j, for all $j \in [1,\ldots,m]$. If there is no alternative satisfying the match (i.e., for every alternative n there is a pair α_j and β_n^j for which match does not hold) then the function is undefined, which is denoted by $\uparrow$.

Definition 3.2 (*Barbs*) Let P be a process. We write $P \downarrow_{\kappa^p}$ if P is structurally congruent to one of the following:

$$(\nu\widetilde{\kappa})(C\{\overline{\kappa}^p(v).P_1\} \mid D\{R\}) \qquad (\nu\widetilde{\kappa})(C\{\kappa^p(x).P_1\} \mid D\{R\})$$
$$(\nu\widetilde{\kappa})(C\{\kappa^p \triangleright \{n_1{:}P_1 \,[]\, \cdots \,[]\, n_m{:}P_m\}\} \mid D\{R\}) \qquad (\nu\widetilde{\kappa})(C\{\kappa^p \triangleleft n_i;\ P'\} \mid D\{R\})$$

We write $\mathsf{barbs}(P\,;\ Q)$ whenever $P \downarrow_{\kappa^p}$ if and only if $Q \downarrow_{\kappa^p}$.

We may now return to describing the reduction rules:

- Rule ⟨R:UREQ⟩ treats the issue of an adaptation request r as a synchronization between a location queue and an adaptation signal. The queue and the signal may be in different contexts; this enables "remote" requests.
- Rules ⟨R:ARR1⟩ and ⟨R:ARR2⟩ resolve arrival predicates by querying the (possibly remote) queue $\widetilde{r}$.
- Rules ⟨R:UPD1⟩ and ⟨R:UPD2⟩ define the update of the behavior at location `loc`. Given an index set I over the update process, suitability with respect to the behavior at `loc` is defined by the function $\mathtt{match}_I$ (cf. Definition 3.1). There are two possibilities. If there is no matching alternative then the current protocol state at `loc` is kept unchanged (Rule ⟨R:UPD1⟩). Otherwise, the predicate holds for an alternative that defines a new protocol state which preserves the barbs of the current state (Rule ⟨R:UPD2⟩). By an abuse of notation, we write $P_1 \in P$ to indicate that P_1 occurs in P, i.e., if $P = (\nu\widetilde{\kappa})C\{P_1\}$ for some C and $\widetilde{\kappa}$.

In addition, our semantics includes standard and/or self-explanatory treatments for reduction under evaluation contexts, parallel composition, located context, and restriction. Also, it accounts for applications of structural congruence, recursion and conditionals.

Example **3.3** Recall process W given in Sect. 2.1, Equation (5). According to our semantics:

$$W \longrightarrow (\nu\kappa)\big(\mathtt{sys}\big[\,\mathtt{buyer}\big[P[\kappa^p/x] \mid \kappa^p\lfloor\alpha_{\mathtt{buy}}\rfloor\big] \mid \mathtt{seller}\big[Q[\kappa^{\overline{p}}/y] \mid \kappa^{\overline{p}}\lfloor\beta_{\mathtt{sel}}\rfloor\big]\,\big]\big)$$
$$\longrightarrow^2 (\nu\kappa)\big(\mathtt{sys}\big[\,\mathtt{buyer}\big[P' \mid \kappa^p\lfloor\alpha_{\mathtt{pay}}\rfloor\big] \mid \mathtt{seller}\big[Q' \mid \kappa^{\overline{p}}\lfloor\beta_{\mathtt{pay}}\rfloor\big]\,\big]\big)$$

Suppose that following an external request the seller must offer a new payment method (a gift card). Precisely, we would like the seller to act according to the type $\beta_{\mathtt{gift}}$ given in (4). Let $\alpha_{\mathtt{gift}}$ be the dual of $\beta_{\mathtt{gift}}$. We then may define the following update process:

$$R^1_{xy} = \mathtt{sys}\big\{\mathtt{case}\ x, y\ \mathtt{of}\ \{(\alpha_{\mathtt{pay}}\,;\ \beta_{\mathtt{pay}}) : \mathtt{buyer}\big[P' \mid x\lfloor\alpha_{\mathtt{gift}}\rfloor\big] \mid \mathtt{seller}\big[Q'' \mid y\lfloor\beta_{\mathtt{gift}}\rfloor\big]\}\big\}$$

Thus, R^1_{xy} keeps the expected implementation for the buyer (P'), but updates its associated monitor. For the seller, both the implementation and monitor are updated; above, Q'' is a process that offers the three payment methods. We may then specify the whole system as: $W \mid \mu\mathcal{X}.\mathtt{if}\ \mathtt{arrive}(\mathtt{sys}, \mathtt{upd}_E)\ \mathtt{then}\ R^1_{xy}\ \mathtt{else}\ \mathcal{X}$. The type system introduced next ensures, among other things, that updates such as R^1_{xy} consider both a process and its associated monitors, ruling out the possibility of discarding the monitors that enable reduction.

4. Session types for eventful run-time adaptation

This section introduces a session type system for the process language of Sect. 3. Our main result (Theorem 4.11) is that well-typed programs enjoy both *safety* (absence of run-time communication errors) and *consistency* properties (update actions do not disrupt established sessions). Our development follows the lines of the typed framework in [DP15].

Syntax. The syntax of session types (ranged over by $\alpha, \beta, \ldots$) has been presented in Sect. 2.1. We consider *basic types* (ranged over by $\tau, \sigma, \ldots$) and write $T, S, \ldots$ to range over τ, α. Although our process language copes with run-time adaptation, our type syntax retains the intuitive meaning of standard session types [HVK98], which we now briefly recall.

Type $?(\tau).\alpha$ (resp. $?(\beta).\alpha$) abstracts the behavior of a channel which receives a value of type τ (resp. a channel of type β) and then continues as α. Dually, type $!(\tau).\alpha$ (resp. $!(\beta).\alpha$) represents the behavior of a channel which sends a value of type τ and then continues as α. Type $\&\{n_1 : \alpha_1, \ldots, n_m : \alpha_m\}$ describes a branching behavior: it offers m behaviors, and if the j-th alternative is selected then it behaves as describedby type α_j ($1 \le j \le m$).

In turn, type $\oplus\{n_1 : \alpha_1, \ldots, n_m : \alpha_m\}$ describes the behavior of a channel which may select a single behavior among $\alpha_1, \ldots, \alpha_m$ and then continues as α_j. We use **end** to type a channel with no communication behavior. Type $\mu\, t.\alpha$ describes recursive behavior; as usual, we consider recursive types under equi-recursive and contractive assumptions.

Along the paper we have informally appealed to *duality* and *subtyping* over session types (denoted $\perp_{\mathsf{c}}$ and $\leq_{\mathsf{c}}$, resp.). Since our session type structure is completely standard, we may rely on the (coinductive) definitions given by Gay and Hole [GH05]; see also Definition A.4.

Our typing judgments generalize usual notions with an *interface* $\mathcal{I}$ for each process. Based on the syntactic occurrences of session establishment prefixes $\overline{a}(x{:}\alpha)$, and $a(x{:}\alpha)$, the interface of a process describes the services appearing in it. We annotate services with a *qualification* q,which may be 'lin' (linear) or 'un' (unrestricted). Intuitively, a service is linear if it is offered a finite number of times, otherwise it is persistent or unrestricted. Thus, the interface of a process gives an "upper bound" on the services that it may execute. The typing system uses interfaces to control the behavior contained by locations after an update. We have:

Definition 4.1 (*Interfaces*) We define an *interface* as the multiset whose underlying set of elements is $\mathrm{I} = \{\mathsf{q}\ u{:}\alpha \mid \mathsf{q} \in \{\mathsf{lin}, \mathsf{un}\}\}$ (i.e., a set of assignments from names to qualified session types). We use $\mathcal{I}, \mathcal{I}', \ldots$ to range over interfaces. We write $dom(\mathcal{I})$ to denote the set $\{u \mid \mathsf{q}\ u : \alpha \in \mathcal{I}\}$ and $\#_{\mathcal{I}}(\mathsf{q}\ u : \alpha) = h$ to mean that u occurs h times in $\mathcal{I}$.

The union of two interfaces is essentially the union of their underlying multisets. We sometimes write $\mathcal{I} \uplus a : \alpha_{\mathsf{lin}}$ and $\mathcal{I} \uplus a : \alpha_{\mathsf{un}}$ to stand for $\mathcal{I} \uplus \{\mathsf{lin}\ a{:}\alpha\}$ and $\mathcal{I} \uplus \{\mathsf{un}\ a{:}\alpha\}$, respectively. Moreover, we write $\mathcal{I}_{\mathsf{lin}}$ (resp. $\mathcal{I}_{\mathsf{un}}$) to denote the subset of $\mathcal{I}$ involving only assignments qualified with lin (resp. un). We now define an ordering relation over interfaces, relying on subtyping:

Definition 4.2 (*Interface ordering*) Given interfaces $\mathcal{I}$ and $\mathcal{I}'$, we write $\mathcal{I} \sqsubseteq \mathcal{I}'$ iff

1. $\forall(\mathsf{lin}\ a{:}\alpha)$ such that $\#_{\mathcal{I}_{\mathsf{lin}}}(\mathsf{lin}\ a{:}\alpha) = h$ with $h > 0$, then one of the following holds:
   (a) There exist h distinct elements $(\mathsf{lin}\ a{:}\beta_i) \in \mathcal{I}'_{\mathsf{lin}}$ such that $\alpha \leq_{\mathsf{c}} \beta_i$ for $i \in [1 \ldots h]$;
   (b) There exists $(\mathsf{un}\ a{:}\beta) \in \mathcal{I}'_{\mathsf{un}}$ such that $\alpha \leq_{\mathsf{c}} \beta$.
2. $\forall(\mathsf{un}\ a{:}\alpha) \in \mathcal{I}_{\mathsf{un}}$ then $(\mathsf{un}\ a{:}\beta) \in \mathcal{I}'_{\mathsf{un}}$ and $\alpha \leq_{\mathsf{c}} \beta$, for some β.

Typing environments. We now define our typing environments, following [DP15]. Recall that we write q to range over qualifiers lin and un.

$$
\begin{array}{lll}
\Delta ::= \emptyset \mid \Delta, k : \alpha \mid \Delta, k : \lfloor \alpha \rfloor & & \text{typing with active sessions} \\
\Gamma ::= \emptyset \mid \Gamma, e : \tau \mid \Gamma, u : \langle \alpha_{\mathsf{q}}, \beta_{\mathsf{q}} \rangle & & \text{first-order environment (with } \alpha_{\mathsf{q}} \perp_{\mathsf{c}} \beta_{\mathsf{q}}) \\
\Theta ::= \emptyset \mid \Theta, \mathcal{X} : \Delta;\ \mathcal{I} \mid \Theta, \mathtt{loc} : \mathcal{I} & & \text{higher-order environment}
\end{array}
$$

We consider typings Δ and environments Γ and Θ. Typing Δ collects assignments from channels to session types; it describes currently active sessions. In our system, Δ also includes *bracketed assignments*, denoted $\kappa^p : \lfloor \alpha \rfloor$, which represent the type for monitors. Subtyping extends to these assignments ($\lfloor \alpha \rfloor \leq_{\mathsf{c}} \lfloor \beta \rfloor$ if $\alpha \leq_{\mathsf{c}} \beta$) and thus to typings. We write $dom(\Delta)$ to denote the set $\{k^p \mid k^p : \alpha \in \Delta \vee k^p : \lfloor \alpha \rfloor \in \Delta\}$. We write $\Delta, k : \alpha$ where $k \notin dom(\Delta)$. Furthermore, we write $\Delta, k : \langle\!\langle \alpha \rangle\!\rangle$ to abbreviate $\Delta, k : \alpha, k : \lfloor \alpha \rfloor$. That is, $k : \langle\!\langle \alpha \rangle\!\rangle$ describes both a session and its associated monitor.

Γ is a first-order environment which maps expressions to basic types and names to pairs of qualified session types. The higher-order environment Θ collects assignments of typings to process variables and interfaces to locations. While the former concerns recursive processes, the latter concerns located processes. As we explain next, by relying on the combination of these two pieces of information the type system ensures that run-time adaptation actions preserve the behavioral interfaces of a process. We write $vdom(\Theta) = \{\mathsf{X} \mid \mathsf{X} : \mathcal{I} \in \Theta\}$ to denote the variables in the domain of Θ. Given these environments, a *type judgment* is of form

$$\Gamma\,;\ \Theta \vdash P \triangleright \Delta;\ \mathcal{I}$$

meaning that, under environments Γ and Θ, process P has active sessions declared in Δ and interface $\mathcal{I}$.

Table 3. Well-typed processes: typing rules (part I)

$$\frac{\alpha \perp_{\mathsf{c}} \beta \quad \Gamma \vdash u \triangleright \langle \alpha_{\mathsf{lin}}, \beta_{\mathsf{lin}} \rangle \quad \gamma \leq_{\mathsf{c}} \alpha \quad \Gamma\,;\,\Theta \vdash P \triangleright \Delta, x : \gamma;\, \mathcal{I}}{\Gamma\,;\,\Theta \vdash u(x:\gamma).P \triangleright \Delta;\, \mathcal{I} \uplus u : \gamma_{\mathsf{lin}}}\ \langle \text{T:Acc} \rangle$$

$$\frac{\alpha \perp_{\mathsf{c}} \beta \quad \Gamma \vdash u \triangleright \langle \alpha_{\mathsf{q}}, \beta_{\mathsf{lin}} \rangle \quad \gamma \leq_{\mathsf{c}} \beta \quad \Gamma\,;\,\Theta \vdash P \triangleright \Delta, x : \gamma;\, \mathcal{I}}{\Gamma\,;\,\Theta \vdash \overline{u}(x:\gamma).P \triangleright \Delta;\, \mathcal{I} \uplus u : \gamma_{\mathsf{lin}}}\ \langle \text{T:Req} \rangle$$

$$\frac{\Gamma\,;\,\Theta \vdash P \triangleright \Delta;\mathcal{I} \quad k \notin \mathrm{dom}(\Delta)}{\Gamma\,;\,\Theta \vdash \mathsf{close}\,(k).P \triangleright \Delta, k : \mathbf{end};\mathcal{I}}\ \langle \text{T:Clo} \rangle$$

$$\frac{\Gamma\,;\,\Theta \vdash P \triangleright \Delta_1;\mathcal{I}_1 \quad \Gamma\,;\,\Theta \vdash Q \triangleright \Delta_2;\mathcal{I}_2}{\Gamma\,;\,\Theta \vdash P \mid Q \triangleright \Delta_1 \cup \Delta_2;\mathcal{I}_1 \uplus \mathcal{I}_2}\ \langle \text{T:Par} \rangle$$

$$\frac{\Gamma\,;\,\Theta \vdash \mu\mathcal{X}.P \triangleright \Delta;\mathcal{I}}{\Gamma\,;\,\Theta, \mathcal{X} : \Delta;\mathcal{I} \vdash P \triangleright \Delta;\mathcal{I}}\ \langle \text{T:Rec} \rangle$$

$$\frac{}{\Theta, \mathtt{loc} : \mathcal{I} \vdash \mathtt{loc} \triangleright \mathcal{I}}\ \langle \text{T:LocEnv} \rangle \qquad \frac{}{\Gamma \vdash r_1 \triangleright msg}\ \langle \text{T:Msg} \rangle \qquad \frac{\Gamma \vdash r_1;\widetilde{r} \triangleright msg}{\Gamma \vdash \widetilde{r} \triangleright msg \qquad \Gamma \vdash r_1 \triangleright msg}\ \langle \text{T:LocQ} \rangle$$

$$\frac{\Theta \vdash \mathtt{loc} \triangleright \mathcal{I} \qquad \Gamma \vdash r \triangleright msg}{\Gamma\,;\,\Theta \vdash \mathsf{arrive}(\mathtt{loc}, r) \triangleright \mathsf{bool}}\ \langle \text{T:Arrive} \rangle$$

$$\frac{\Gamma \vdash r \triangleright msg}{\Gamma\,;\,\Theta \vdash \overline{\mathtt{loc}}(r) \triangleright \emptyset;\emptyset}\ \langle \text{T:Sig} \rangle \qquad \frac{\Theta \vdash \mathtt{loc} \triangleright \mathcal{I} \qquad \Gamma\,;\,\Theta \vdash P \triangleright \Delta;\mathcal{I}' \qquad \mathcal{I}' \sqsubseteq \mathcal{I}}{\Gamma\,;\,\Theta \vdash \mathtt{loc}[P] \triangleright \Delta;\mathcal{I}'}\ \langle \text{T:Loc} \rangle$$

$$\frac{\Gamma \vdash \widetilde{r} \triangleright msg}{\Gamma\,;\,\Theta \vdash \mathtt{loc}\lfloor \widetilde{r} \rfloor \triangleright \emptyset;\emptyset}\ \langle \text{T:QLoc} \rangle$$

$$\frac{\Gamma\,;\,\Theta \vdash P \triangleright \Delta, \kappa^p : \langle\!\langle \alpha_1 \rangle\!\rangle, \kappa^{\overline{p}} : \langle\!\langle \alpha_2 \rangle\!\rangle;\mathcal{I} \quad \alpha_1 \perp_{\mathsf{c}} \alpha_2}{\Gamma\,;\,\Theta \vdash (\nu\kappa)P \triangleright \Delta;\mathcal{I}}\ \langle \text{T:CRes} \rangle \qquad \frac{\Gamma\,;\,\Theta \vdash P \triangleright \Delta;\mathcal{I} \cup \mathcal{I}_u \qquad u \notin \mathrm{dom}(\mathcal{I})}{\Gamma\,;\,\Theta \vdash (\nu u)P \triangleright \Delta;\mathcal{I}}\ \langle \text{T:NRes} \rangle$$

$$\frac{\Gamma\,;\,\Theta \vdash P \triangleright \Delta;\mathcal{I} \qquad \Delta \leq_{\mathsf{c}} \Delta' \qquad \mathcal{I} \sqsubseteq \mathcal{I}'}{\Gamma\,;\,\Theta \vdash P \triangleright \Delta';\mathcal{I}'}\ \langle \text{T:Sub} \rangle \qquad \frac{}{\Gamma\,;\,\Theta \vdash k\lfloor \alpha \rfloor \triangleright k : \lfloor \alpha \rfloor;\emptyset}\ \langle \text{T:Que} \rangle$$

$$\frac{\Theta \vdash \mathtt{loc} \triangleright \mathcal{I} \qquad \forall j \in J,\ \mathsf{fv}(Q_j) \setminus \{x_1, \ldots, x_m\} = \emptyset \qquad \Gamma\,;\,\Theta \vdash Q_j \triangleright x_1{:}\langle\!\langle \beta_1^j \rangle\!\rangle; \cdots ; x_m{:}\langle\!\langle \beta_m^j \rangle\!\rangle;\mathcal{I}_j \qquad \mathcal{I}_j \sqsubseteq \mathcal{I}}{\Gamma\,;\,\Theta \vdash \mathtt{loc}\{\mathsf{case}\ x_1, \ldots, x_m\ \mathsf{of}\ \{(\beta_1^j; \cdots ; \beta_m^j) : Q_j\}_{j \in J}\} \triangleright \emptyset;\emptyset}\ \langle \text{T:Adapt} \rangle$$

Typing rules are shown in Tables 3 and 4. Below we comment on some of the rules in Table 3: the rest are standard and/or self explanatory. Rule ⟨T:Adapt⟩ types update processes. Notice that the typing rule ensures that each process Q_i has exactly the same active sessions that those declared in the respective case. Also, we require that alternatives contain both processes and monitors. With $\mathcal{I}_j \sqsubseteq \mathcal{I}$ we guarantee that the process behavior does not "exceed" the expected behavior within the location. Rule ⟨T:Sub⟩ takes care of subtyping both for typings Δ and interfaces. Rule ⟨T:CRes⟩ types channel restriction; it ensures typing duality among partners of a session and their respective queues. Rule ⟨T:NRes⟩ types hiding of service names, by simply removing their declarations from the interface $\mathcal{I}$ of the process. In the rule, $\mathcal{I}_u$ contains only declarations for u, i.e., $\forall\, v \neq u,\ v \notin \mathrm{dom}\,\mathcal{I}_u$. Typing of queues is given by Rule ⟨T:Que⟩ that simply assigns type $k : \lfloor \alpha \rfloor$ to queue $k\lfloor \alpha \rfloor$.

Example **4.3** Consider the process from Example 3.3:

$$(\nu\kappa)\big(\mathtt{sys}\big[\,\mathtt{buyer}\big[P[\kappa^p/x] \mid \kappa^p\lfloor \alpha_{\mathtt{buy}} \rfloor\big] \mid \mathtt{seller}\big[Q[\kappa^{\overline{p}}/y] \mid \kappa^{\overline{p}}\lfloor \beta_{\mathtt{sel}} \rfloor\big]\,\big]\big)$$

The type of process R located in sys is $\Gamma\,;\ \Theta \vdash R \triangleright \Delta;\ \mathcal{I}$ with

$$\begin{array}{l}\Gamma \ := u : \langle \mathsf{lin} : \alpha_{\mathtt{buy}}, \mathsf{lin} : \beta_{\mathtt{sel}} \rangle \\ \Theta \ := \mathtt{buyer} : \{\mathsf{lin}\ \overline{u} : \alpha_{\mathtt{buy}}\},\ \mathtt{seller} : \{\mathsf{lin}\ u : \beta_{\mathtt{sel}}\},\ \mathtt{sys} : \{\mathsf{lin}\ \overline{u} : \alpha_{\mathtt{buy}}, \mathsf{lin}\ u : \beta_{\mathtt{sel}}\} \\ \Delta \ := \kappa^p : \langle\!\langle \alpha_{\mathtt{buy}} \rangle\!\rangle, \kappa^{\overline{p}} : \langle\!\langle \beta_{\mathtt{sel}} \rangle\!\rangle \\ \mathcal{I} \ := \emptyset\end{array}$$

Our type system enjoys the standard *subject reduction* property. We rely on *balanced* typings:

Definition 4.4 We say that typing Δ is *balanced* iff for all $\kappa^p : \alpha \in \Delta$ (resp. $\kappa^p : \lfloor \alpha \rfloor \in \Delta$) then also $\kappa^{\overline{p}} : \beta \in \Delta$ (resp. $\kappa^{\overline{p}} : \lfloor \beta \rfloor \in \Delta$), with $\alpha \perp_{\mathsf{c}} \beta$.

Similarly to what has been proposed in [HYC08], here we introduce a reduction over session typings:

Definition 4.5 (*Reduction for typings*) Reduction for typings, denoted $\Delta \longmapsto \Delta'$, is defined by the following rules:

1. $\Delta, \kappa^p : \langle\!\langle !(T).\alpha \rangle\!\rangle, \kappa^{\overline{p}} : \langle\!\langle ?(T).\beta \rangle\!\rangle \longmapsto \Delta, \kappa^p : \langle\!\langle \alpha \rangle\!\rangle, \kappa^{\overline{p}} : \langle\!\langle \beta \rangle\!\rangle$.
2. $\Delta, \kappa^p : \langle\!\langle \&\{n_1{:}\alpha_1, \ldots, n_m{:}\alpha_m\} \rangle\!\rangle, \kappa^{\overline{p}} : \langle\!\langle \oplus\{n_1{:}\beta_1, \ldots, n_m{:}\beta_m\} \rangle\!\rangle \longmapsto \Delta, \kappa^p : \langle\!\langle \alpha_i \rangle\!\rangle, \kappa^{\overline{p}} : \langle\!\langle \beta_i \rangle\!\rangle$, with $i \in \{1, \ldots, m\}$.

Table 4. Well-typed processes: typing rules (part II)

$$\langle \text{T:BOOL} \rangle \quad \frac{}{\Gamma \vdash \texttt{true}, \texttt{false} \triangleright \mathsf{bool}} \qquad \langle \text{T:NAME} \rangle \quad \frac{}{\Gamma \vdash u \triangleright \mathrm{name}} \qquad \langle \text{T:BVAR} \rangle \quad \frac{}{\Gamma, x : \mathsf{bool} \vdash x \triangleright \mathsf{bool}} \qquad \langle \text{T:NVAR} \rangle \quad \frac{}{\Gamma, x : \mathrm{name} \vdash x \triangleright \mathrm{name}}$$

$$\langle \text{T:EQ} \rangle \quad \frac{d = u \vee d = \kappa^p \vee d = x}{\Gamma \vdash d = d \triangleright \mathsf{bool}} \qquad \langle \text{T:SER} \rangle \quad \frac{\alpha \perp_{\mathsf{C}} \beta}{\Gamma, u : \langle \alpha_{\mathsf{q}}, \beta_{\mathsf{q}} \rangle \vdash u \triangleright \langle \alpha_{\mathsf{q}}, \beta_{\mathsf{q}} \rangle} \qquad \langle \text{T:RVAR} \rangle \quad \frac{}{\Gamma; \Theta, \mathcal{X} : \Delta, \mathcal{I} \vdash \mathcal{X} : \Delta; \mathcal{I}} \qquad \langle \text{T:NIL} \rangle \quad \frac{}{\Gamma\,;\, \Theta \vdash \mathbf{0} \triangleright \emptyset; \emptyset}$$

$$\langle \text{T:THR} \rangle \quad \frac{\Gamma\,;\, \Theta \vdash P \triangleright \Delta, k : \beta; \mathcal{I}}{\Gamma\,;\, \Theta \vdash \overline{k}(k').P \triangleright \Delta, k : !(\alpha).\beta, k' : \alpha; \mathcal{I}} \qquad \langle \text{T:CAT} \rangle \quad \frac{\Gamma\,;\, \Theta \vdash P \triangleright \Delta, k : \beta, x : \alpha; \mathcal{I}}{\Gamma\,;\, \Theta \vdash k(x).P \triangleright \Delta, k : ?(\alpha).\beta; \mathcal{I}}$$

$$\langle \text{T:IN} \rangle \quad \frac{\Gamma, x : \tau\,;\, \Theta \vdash P \triangleright \Delta, k : \alpha; \mathcal{I}}{\Gamma\,;\, \Theta \vdash k(x).P \triangleright \Delta, k : ?(\tau).\alpha; \mathcal{I}} \qquad \langle \text{T:OUT} \rangle \quad \frac{\Gamma\,;\, \Theta \vdash P \triangleright \Delta, k : \alpha; \mathcal{I} \quad \Gamma \vdash e \triangleright \tau}{\Gamma\,;\, \Theta \vdash \overline{k}(e).P \triangleright \Delta, k : !(\tau).\alpha; \mathcal{I}}$$

$$\langle \text{T:WEAKC} \rangle \quad \frac{\Gamma\,;\, \Theta \vdash P \triangleright \Delta; \mathcal{I} \quad \kappa^+, \kappa^- \notin dom(\Delta)}{\Gamma\,;\, \Theta \vdash (\nu\kappa)P \triangleright \Delta; \mathcal{I}} \qquad \langle \text{T:WEAKN} \rangle \quad \frac{\Gamma\,;\, \Theta \vdash P \triangleright \Delta; \mathcal{I} \quad u \notin dom(\mathcal{I})}{\Gamma\,;\, \Theta \vdash (\nu u)P \triangleright \Delta; \mathcal{I}}$$

$$\langle \text{T:IF} \rangle \quad \frac{\Gamma\,;\, \Theta \vdash e \triangleright \mathsf{bool} \qquad \Gamma\,;\, \Theta \vdash P \triangleright \Delta; \mathcal{I} \qquad \Gamma\,;\, \Theta \vdash Q \triangleright \Delta; \mathcal{I}}{\Gamma\,;\, \Theta \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright \Delta; \mathcal{I}} \qquad \langle \text{T:SEL} \rangle \quad \frac{\Gamma\,;\, \Theta \vdash P \triangleright \Delta, k : \alpha_i; \mathcal{I} \qquad 1 \leq i \leq m}{\Gamma\,;\, \Theta \vdash k \lhd n_i; P \triangleright \Delta, k : \oplus\{n_1 : \alpha_1, \ldots, n_m : \alpha_m\}; \mathcal{I}}$$

$$\langle \text{T:BRA} \rangle \quad \frac{\Gamma\,;\, \Theta \vdash P_1 \triangleright \Delta, k : \alpha_1; \mathcal{I}_1 \quad \cdots \quad \Gamma\,;\, \Theta \vdash P_m \triangleright \Delta, k : \alpha_m; \mathcal{I}_m \quad \mathcal{I} = \mathcal{I}_1 \uplus \ldots \uplus \mathcal{I}_m}{\Gamma\,;\, \Theta \vdash k \rhd \{n_1 : P_1 \,[\!]\, \cdots \,[\!]\, n_m : P_m\} \triangleright \Delta, k : \&\{n_1 : \alpha_1, \ldots, n_m : \alpha_m\}; \mathcal{I}}$$

Theorem 4.6 (Subject reduction) *If* $\Gamma\,;\, \Theta \vdash P \triangleright \Delta;\, \mathcal{I}$ *with* Δ *balanced and* $P \longrightarrow Q$ *then* $\Gamma\,;\, \Theta \vdash Q \triangleright \Delta';\, \mathcal{I}'$, *for some* $\mathcal{I}' \subseteq \mathcal{I}$ *and balanced* Δ' *such that either* $\Delta' \leq_{\mathsf{C}} \Delta$ *or* $\Delta \longmapsto \Delta'$.

Proof By induction on the last rule applied in the reduction. See Appendix B for details. □

We now define and state *safety* and *consistency* properties. While safety guarantees adherence to prescribed session types and absence of run-time *error*s, consistency ensures that sessions are not jeopardized by careless run-time adaptation actions. Defining both properties requires the following notions of *κ-processes* and *(located) κ-redexes*:

Definition 4.7 (*κ-processes, κ-redexes, errors*) A process P is a *κ-process* if it is a prefixed process with subject κ^p, i.e., P is one of the following:

$$\kappa^{\,p}(x).P' \qquad \overline{\kappa}^{\,p}(v).P' \qquad \mathsf{close}\,(\kappa^{\,p}).P' \qquad \kappa^{\,p} \rhd \{n_i : P_i\}_{i \in I} \qquad \kappa^{\,p} \lhd n.P'$$

Process P is a *κ-redex* if it contains the composition of exactly two κ-processes with *opposing polarities*, i.e., for some contexts C, D, and E, and processes $P_1, P_2, \ldots, P_m$, and P', process P is structurally congruent to one of the following:

$$\begin{array}{rcl} (\nu\widetilde{\kappa})(C\{\overline{\kappa}^{\,p}(v).P_1\} & | & D\{\kappa^{\,\overline{p}}(x).P_2\}) \\ (\nu\widetilde{\kappa})(C\{\kappa^{\,p} \rhd \{n_1 : P_1 \,[\!]\, \cdots \,[\!]\, n_m : P_m\}\} & | & D\{\kappa^{\,\overline{p}} \lhd n_i;\ P'\}) \\ (\nu\widetilde{\kappa})(C\{\mathsf{close}\,(\kappa^{\,p}).P_1\} & | & D\{\mathsf{close}\,(\kappa^{\,\overline{p}}).P_2\}) \end{array}$$

A *located κ-redex* is a κ-redex in which one or both of its constituent κ-processes are contained by least one located process. P is an *error* if $P \equiv (\nu\widetilde{\kappa})(Q \mid R)$ where, for some κ, Q contains either exactly two κ-processes that do not form a κ-redex or three or more κ-processes.

Example **4.8** The following processes

$$l_2\big[\, l_1\big[\kappa^{\,p}(\widetilde{x}).P_1\big] \mid \overline{\kappa^{\,\overline{p}}}(v).P_2\,\big] \qquad l_1\big[\kappa^{\,p}(\widetilde{x}).P_1\big] \mid l_2\big[\overline{\kappa^{\,\overline{p}}}(v).P_2\big] \qquad l_1\big[\kappa^{\,p}(\widetilde{x}).P_1 \mid \overline{\kappa^{\,\overline{p}}}(v).P_2\big]$$

are located κ-redexes, whereas process $\kappa^{\,p}(\widetilde{x}).P_1 \mid \overline{\kappa^{\,\overline{p}}}(v).P_2$ is not.

Informally, a process P is called *consistent* if whenever it has a located κ-redex then update actions do not destroy such a redex. Below, we formalize this intuition. Let us write $P \longrightarrow_{\text{upd}} P'$ for any reduction inferred using Rule ⟨R:UPD2⟩ (cf. Table 2). We then define:

Definition 4.9 (*Safety and consistency*) Let P be a process.

- We say P is *safe* if it never reduces into an error.
- We say P is *update-consistent* if and only if, for all P' and κ such that $P \longrightarrow^* P'$ and P' contains a κ-redex, if $P' \longrightarrow_{\text{upd}} P''$ then P'' contains a κ-redex.

Lemma 4.10 *If* $\Gamma\,;\ \Theta \vdash P \triangleright \Delta;\ \mathcal{I}$ *with* Δ *balanced then* P *is not an error process.*

Proof Follows from session type duality, Definition 4.4 (balanced typing), and Definition 4.7 (error). □

We now state our main result; it follows as a consequence of Theorem 4.6.

Theorem 4.11 (Typing ensures safety and consistency) *If* $\Gamma\,;\ \Theta \vdash P \triangleright \Delta;\ \mathcal{I}$ *with* Δ *balanced then program* P *is safe and update-consistent.*

Proof Safety means that P never reduces into an error. Since Δ is balanced, by Lemma 4.10 we know that P is not an error. Theorem 4.6 ensures that reduction preserves balanced typings; therefore, P never reduces to an error.

Proving update-consistency entails showing that for all P_0 and κ such that $P \longrightarrow^* P_0$ and P_0 contains a κ-redex, if $P_0 \longrightarrow_{\text{upd}} P_1$ then P_1 contains a κ-redex. That is, P_0 has both the κ-redex and an update action which preserves it. By Theorem 4.6 we know that P_0 is well-typed under a balanced typing. To show consistency we distinguish two cases. The first case is when the κ-redex is *not* located in P_0: then, the update action cannot affect it (i.e., the update concerns located process in P_0 which do not enclose the κ-processes that constitute the κ-redex) and the thesis follows. The second case is when the κ-redex is located in P_0: therefore, κ-processes that constitute the κ-redex could be affected by the reduction $P_0 \longrightarrow_{\text{upd}} P_1$. There are several sub-cases, depending on which κ-processes are located. We consider the sub-case in which one κ-process (say, $R_1 = \kappa\,^{p}(x).R'$) is located whereas the other (say, $R_2 = \overline{\kappa}\,^{\overline{p}}(v).R''$) is not; other cases are similar. The proof follows directly from Rule ⟨R:UPD2⟩ (cf. Table 2) which replaces R_1 with an alternative Q_l. The rule not only ensures that R_1 and Q_l have matching session types (up to subtyping) but also that they have the same barbs, i.e., the same top-level actions. This ensures that the κ-redex is preserved under the reduction $\longrightarrow_{\text{upd}}$, and so the thesis follows. □

Remark **4.12** (Asynchronous communication) In this section, we have focused on *synchronous* communication: this allows us to give a compact semantics, relying on a standard type structure. To account for asynchrony, we would require a run-time syntax for programs with queues for in-transit messages (values, sessions, labels). The type system should be extended to accommodate these new run-time processes. In our case, an extension with asynchrony would rely on the machinery defined in [Kou12]; we present such a machinery (for multiparty sessions) in the following section.

Remark **4.13** (Incremental adaptation) Adaptation in our framework is "incremental" in that modifications always preserve/extend active session protocols, exploiting subtyping. Our framework can be modified so that arbitrary protocols are installed as a result of an update. One would need to ensure that the two endpoints of a session are present in the same location: arbitrary updates are safe as long as both endpoints are simultaneously updated with dual protocols. This alternative requires modifying definitions for matching (Definition 3.1) and interface ordering (Definition 4.2).

5. Event-based adaptation for multiparty communications

Here we generalize our approach to the case of multiparty communications. As already explained, we present a proof of concept that integrates type-directed constructs for update and eventful constructs for adaptation requests into the framework put forward by Coppo et al. [CDV15].

As described in Sect. 2.2, in the framework of [CDV15] communication actions from adaptation mechanisms are tied together, as adaptation flags occur within global type specifications at the same level of protocol exchanges. Our event-based approach treats adaptation at the level of processes, defining a separation between communication and adaptation concerns, which in our opinion is beneficial for specification and analysis.

Integrating our event-based approach entails novelties with respect to the notions of global types and networks in [CDV15]. Monitored processes will be now embedded into distributed locations which contain local collections and queues that govern local and external adaptation. Furthermore, at the level of processes we replace processes $c?l(\lambda, \mathtt{T}).P$, and $c!l(\lambda(F), \mathtt{T}).P$ with a construct $c!(\mathsf{G}).P$ which enforces a locally-motivated adaptation based on global type G. Overall, we have a two-level framework of processes and systems (instead of processes, networks, and systems, as in [CDV15]).

5.1. Syntax of types

5.1.1. Global types

Global types specify structured interactions between two or more participants. We distinguish between communication and adaptation concerns; for this reason, unlike [CDV15], our global types do not mention synchronizations related to adaptation. We write $\mathsf{p}, \mathsf{q}, \ldots$ and $\Pi, \Pi', \ldots$ to denote participants and sets of participants, respectively. Exchange of values take place on *labels*, denoted $\lambda, \lambda', \ldots$ in the sequel, which are useful to express choices. We write $\mathtt{loc}, \mathtt{loc}', \ldots$ to range over locations. As in [CDV15], we assume that communicated values are extracted from a set of sorts S. Formally, we have:

Definition 5.1 (*Global types*) The set of *global types* is defined by

$$\mathsf{G} ::= \mathsf{p} \to \mathsf{q} : \{\lambda_i(S_i).\mathsf{G}_i\}_{i\in I} \mid \mathsf{end}$$

As we follow [CDV15], our syntax of global types is admittedly simple: a global type $\mathsf{p} \to \mathsf{q} : \{\lambda_i(S_i).\mathsf{G}_i\}_{i\in I}$ represents a labeled communication from participant p to participant q. Global type end denotes the terminated protocol. We write $\mathtt{part}(\mathsf{G})$ to denote the set of participants declared in G.

5.1.2. Monitors

Monitors capture the local view that participants have of a global type. They are used to enable actions of the process implementations; they also contain information on the participants and labels involved in the directed exchanges.

Definition 5.2 (*Monitors*) The set of *monitors* is defined by:

$$\mathcal{M} ::= \mathsf{p}?\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I} \mid \mathsf{p}!\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I} \mid \mathsf{end}$$

An input monitor $\mathsf{p}?\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I}$ can be associated to a process which can receive, for each $i \in I$, a value of sort S_i with label λ_i, with a continuation that is associated to $\mathcal{M}_i$ (external choice). The output monitor $\mathsf{p}!\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I}$ is dual: it can be associated to a process which sends a value of sort S_i, distinguished by label λ_i, for each $i \in I$, with a continuation that is associated to $\mathcal{M}_i$ (internal choice).

Following the approach of multiparty session types [HYC08], the monitor of a participant is obtained via a *projection function*, defined below.

Definition 5.3 (*Projection of a global type*) The projection of a global type G onto a monitor for a participant p, denoted $\mathsf{G} \downarrow_{\mathsf{p}}$, is defined as follows:

- $$(\mathsf{p} \to \mathsf{q} : \{\lambda_i(S_i).\mathsf{G}_i\}_{i\in I}) \downarrow_{\mathsf{r}} \;=\; \begin{cases} \mathsf{p}?\{\lambda_i(S_i).\mathsf{G}_i \downarrow_{\mathsf{r}}\}_{i\in I} & \text{if } \mathsf{r} = \mathsf{q} \\ \mathsf{q}!\{\lambda_i(S_i).\mathsf{G}_i \downarrow_{\mathsf{r}}\}_{i\in I} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{G}_1 \downarrow_{\mathsf{r}} & \text{if } \mathsf{r} \neq \mathsf{p} \text{ and } \mathsf{r} \neq \mathsf{q} \text{ and } \mathsf{G}_i \downarrow_{\mathsf{r}} = \mathsf{G}_j \downarrow_{\mathsf{r}} \text{ for all } i, j \in I \end{cases}$$
- $\mathsf{end} \downarrow_{\mathsf{r}} \;=\; \mathsf{end}$

We will say that a global type is *well-formed* if its projections are defined for all participants. We shall work only with well-formed global types. We assume that each well-formed G is associated with a mapping $\mathcal{L}_{\mathsf{G}}$ from participants to locations such that $\mathsf{p} \neq \mathsf{p}'$ implies $\mathcal{L}_{\mathsf{G}}(\mathsf{p}) \neq \mathcal{L}_{\mathsf{G}}(\mathsf{p}')$, for all $\mathsf{p}, \mathsf{p}' \in \mathtt{part}(\mathsf{G})$.

Table 5. Typing rules for multiparty processes

$$\frac{}{\Gamma, e : S \vdash e : S}\ \langle \text{M:Exp} \rangle \qquad \frac{}{\Gamma \vdash \mathbf{0} \triangleright c : \mathsf{end}}\ \langle \text{M:End} \rangle \qquad \frac{\Gamma \vdash P \triangleright c : T}{\Gamma \vdash c!(G).P \triangleright c : T}\ \langle \text{M:Adapt} \rangle$$

$$\frac{\Gamma, X : T \vdash P \triangleright c : T}{\Gamma \vdash \mu \mathcal{X}.P \triangleright c : T}\ \langle \text{M:Rec} \rangle \qquad \frac{\Gamma \vdash e : S \quad \Gamma \vdash P_1 \triangleright c : T \quad \Gamma \vdash P_2 \triangleright c : T}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \triangleright c : T}\ \langle \text{M:If} \rangle$$

$$\frac{\Gamma, x_1 : S_1 \vdash P_1 \triangleright c : T_1 \quad \cdots \quad \Gamma, x_n : S_n \vdash P_n \triangleright c : T_n}{\Gamma \vdash c?\{\lambda_i(x_i).P_i\}_{i \in I} \triangleright c : \&\{\lambda_i(S_i).T_i\}_{i \in I}}\ \langle \text{M:Rcv} \rangle$$

$$\frac{\Gamma \vdash P \triangleright c : T_k \quad \Gamma \vdash e : S \quad k \in I}{\Gamma \vdash c!\lambda(e).P \triangleright c : \oplus\{\lambda_i(S_i).T_i\}_{i \in I}}\ \langle \text{M:Send} \rangle$$

5.1.3. Process types

We now describe types for the process language that will be introduced in Definition 5.6. While in multiparty session types [HYC08], the process types correspond to *local types* (the projection of the global type onto every participant), the approach in [CDV15] uses a discipline of process types which is different from local types (i.e., monitors). This adds flexibility, and is useful to establish an independence between local implementations (as formalized by processes) and choreographic coordination (as formalized by global types and monitors). Clearly, process types cannot be completely independent from monitors; a basic coherence between the two should exist. This coherence is defined in terms of *adequacy* (cf. Definition 5.5), a relation that uses subtyping to determine which process types "fit" a given monitor.

Following [Pad10], the process types defined on [CDV15] exploit intersection and union types; this ensures an intuitive subtyping relation. Here we consider a syntax for process types that is certainly closer to the local types of [HYC08] than the syntax in [CDV15]. This is for the sake of presentation, as we wish to keep a simple conceptual relation with the binary session types introduced in Sect. 4. We then have:

Definition 5.4 (*Process types*) The syntax of process types is defined as follows:

$$T ::= \&\{\lambda_i(S_i).T_i\}_{i \in I} \mid \oplus\{\lambda_i(S_i).T_i\}_{i \in I} \mid \mu\, t.T \mid t \mid \mathsf{end}$$

We write $\mathcal{T}$ to denote the set of all types, and use $T, T', \ldots$ to range over process types.

Thus, intuitively, process types $\&\{\lambda_i(S_i).T_i\}_{i \in I}$ and $\oplus\{\lambda_i(S_i).T_i\}_{i \in I}$ are used to type labelled inputs and outputs, respectively. Recursive types are as expected, following usual considerations for type equality.

An *environment* Γ is a finite mapping from expression variables to sorts and from process variables to types:

$$\Gamma ::= \emptyset \mid \Gamma, x : S \mid \Gamma, X : T$$

where notation $\Gamma, x : S$ (resp. $\Gamma, X : \mathsf{T}$) means that x (resp. X) does not occur in the domain of Γ.

We now introduce *typing judgments* for expressions e and processes P:

$$\Gamma \vdash e : S \qquad \Gamma \vdash P \triangleright c : T$$

Expressions are typed by sorts; we assume standard typing rules. The judgments for processes which makes it explicit that processes with at most one channel can be typed.

Typing rules for processes are given in Table 5. Due to the absence of explicit flags for adaptation, they are a subset of the typing rules given in [CDV15]. We also consider Rule ⟨M:Adapt⟩ which represents the fact that internal adaptation requests are defined independently from communication protocols abstracted by types.

The relation between process types and monitors is formalized via *adequacy*. Adequacy relies on a subtyping relation, here denoted $\leq$; it corresponds to a finite variant of the subtyping $\leq_{\mathsf{C}}$ (formally given in Definition A.4). Our notion of adequacy is simple, and highlights how the difference between monitors and process types lies in the information about participant identities:

Definition 5.5 (*Adequacy*) Let $|\cdot|$ be a mapping from monitors to types defined as follows:

$$\begin{aligned} |\ \mathsf{p}?\{\lambda_i(S_i).\mathcal{M}_i\}_{i \in I}\ | &= \&\{\lambda_i(S_i).\ |\ \mathcal{M}_i\ |\}_{i \in I} \\ |\ \mathsf{p}!\{\lambda_i(S_i).\mathcal{M}_i\}_{i \in I}\ | &= \oplus\{\lambda_i(S_i).\ |\ \mathcal{M}_i\ |\}_{i \in I} \\ |\ \mathsf{end}\ | &= \mathsf{end} \end{aligned}$$

We then say that type T is *adequate* to monitor $\mathcal{M}$ (notation $T \propto \mathcal{M}$) if $T \leq |\ \mathcal{M}\ |$.

5.2. Processes and networks

5.2.1. Syntax

Processes abstract the protocol implementations for each participant. The association of a process P with a monitor $\mathcal{M}$ results into a *monitored process* $\mathcal{M}[\![P]\!]$, which realizes the behavior of a single participant. As hinted at above, the main differences of our framework with respect to [CDV15] appear at the level of monitored processes and networks (collections of monitored processes).

The definition of processes implementing multiparty services, given next, is rather standard. We have operators for sending and receiving messages, conditionals, and recursion. On top of these operators we add a primitive for broadcasting an *internal adaptation request*, denoted $c!(\mathsf{G}).P$. Intuitively, this primitive is an output action that mentions a new global type to be implemented by protocol participants.

Definition 5.6 (*Processes*) The syntax of processes is defined as:

$$\begin{aligned} P ::= {} & c?\{\lambda_i(x_i).P_i\}_{i\in I} \mid c!\,\lambda(e).P \mid \mu\,\mathcal{X}.P \mid \mathcal{X} \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid \mathbf{0} \\ \mid {} & c!(\mathsf{G}).P \end{aligned}$$

Each process owns a unique channel, denoted y. Given a session κ and a participant p, channel y in the user code is substituted at run-time by a session channel $\kappa[\mathsf{p}]$. In the following we write c to denote either y (the user channel) or $\kappa[\mathsf{p}]$ (the session channel). We assume that processes pass around expressions, defined as expected. Every expression e reduces to a value v; this is denoted $e \rightsquigarrow v$.

As an example of internal adaptation requests, suppose participants $\mathsf{p}_1, \ldots, \mathsf{p}_j, \ldots, \mathsf{p}_n$ establish session κ to communicate according to a specific protocol (say, Skype). To request a protocol change, participant p_j may issue an adaptation request to all its partners by including the process $\kappa[\mathsf{p}_j]!(\mathsf{GTalk}).P$ in its local implementation. As we will formalize later on, this request will arrive to the event queue of session κ, and then all participants will proceed to adapt their local behavior to the new global type GTalk.

Our notion of networks departs significantly from that in [CDV15] by considering an explicitly distributed setting based on (inactive) locations and event queues:

Definition 5.7 (*Networks*) The syntax of *networks* N and *adaptation requests* r is defined as

$N ::=$	$\mathtt{new}(\mathsf{G})$	initialize global type
	$\mid\ \mathtt{loc}\big[\mathcal{M}[\![P]\!];\ \mathcal{P};\ \mathtt{loc}\lfloor r\rfloor\big]$	active location
	$\mid\ \mathtt{loc}\big[\mathcal{P};\ \mathtt{loc}\lfloor r\rfloor\big]$	inactive location
	$\mid\ \kappa : h$	message queue
	$\mid\ \kappa^e\lfloor h\rfloor$	internal adaptation: event queue
	$\mid\ (\nu\kappa)N$	restriction
	$\mid\ N \mid N$	parallel composition
$r ::=$	ϵ	empty request
	$\mid\ (add : \langle Q, T\rangle)$	external adaptation: upgrade request
	$\mid\ (upd : \mathsf{case}\ \ x\ \ \mathsf{of}\ \{(T_i) : Q_i\}_{i\in I})$	external adaptation: update request

A location represents a place where several programs can be executed. We assume a preexisting set of locations which are "filled in" with appropriate monitored processes upon initialization of global types (see below). An active location $\mathtt{loc}\big[\mathcal{M}[\![P]\!];\ \mathcal{P};\ \mathtt{loc}\lfloor r\rfloor\big]$ is composed of a monitored process $\mathcal{M}[\![P]\!]$, a local collection of typed processes $\mathcal{P}$ (i.e., a set of pairs $(Q,\ T)$ where Q is a process and T is its corresponding type), and a queue $\mathtt{loc}\lfloor r\rfloor$ (where r stands for an *external* adaptation request). For simplicity, we assume that in each location only one session can be active. An inactive location is simply a location without a monitored process. The queue of a location connects it to an unspecified environment that can (i) add new processes to the collection (or upgrade the an existing process), or (ii) update the behavior of a monitored process. This is similar to the update processes for the binary framework.

As we shall see, initializing a global type G entails setting up monitored processes in appropriate locations, establishing a session channel κ, and creating a message queue $\kappa : h$ for supporting asynchronous communication. We write $\emptyset$ to denote the empty message queue; messages are of the form $(\mathsf{p}, \mathsf{q}, \lambda(v))$ thus denoting the fact that participant p sends value v and label λ to participant q. Message concatenation is denoted with '$\cdot$'.

The event queue $\kappa^e\lfloor h\rfloor$ handles internal adaptation requests, associated to process $c!(\mathsf{G}).P$ introduced earlier.

Table 6. LTS for multiparty processes

$$\frac{}{\kappa[\mathsf{p}]?\{\lambda_i(x_i).P_i\}_{i\in I} \xrightarrow{\kappa[\mathsf{p}]?\lambda_j(v_j)} P_j[v/x]}\ \langle \text{M:P:In}\rangle \qquad \frac{e \rightsquigarrow v}{\kappa[\mathsf{p}]!\lambda(e).P \xrightarrow{\kappa[\mathsf{p}]!\lambda(v)} P}\ \langle \text{M:P:Out}\rangle \qquad \frac{}{\kappa[\mathsf{p}]!\lambda(G).P \xrightarrow{\kappa[p]!\lambda(G)} P}\ \langle \text{M:P:AdapOut}\rangle$$

$$\frac{e \rightsquigarrow \texttt{true}}{\mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \xrightarrow{\tau} P_1}\ \langle \text{M:P:IfT}\rangle \qquad \frac{e \rightsquigarrow \texttt{false}}{\mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \xrightarrow{\tau} P_2}\ \langle \text{M:P:IfF}\rangle \qquad \frac{}{\mu\mathcal{X}.P \xrightarrow{\tau} P[\mu\mathcal{X}.P/X]}\ \langle \text{M:P:Rec}\rangle$$

5.2.2. Semantics

The semantics of networks is given in terms of a reduction relation, denoted $N \longrightarrow N'$. We require some auxiliary definitions, in order to connect the behavior of monitors and processes to that of whole networks; we also require a structural congruence to capture, among other things, asynchronous message communication as handled via queues. As in [CDV15], we assume that all local collections $\mathcal{P}$ are *complete*, which intuitively means that such collections contain all conceivable pairs of processes and types.

LTS for monitors and processes. Reduction for networks relies on LTSs for monitors and processes. The LTS for monitors is rather simple; it is used to enable the actions to be performed by processes:

$$\mathsf{p}?\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I} \xrightarrow{\mathsf{p}?\,\lambda_j} \mathcal{M}_j \quad (j \in I) \qquad\qquad \mathsf{q}!\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I} \xrightarrow{\mathsf{q}!\,\lambda_j} \mathcal{M}_j \quad (j \in I)$$

The LTS for processes relies on the following set of labels, with associated rules as in Table 6.

$$\alpha ::= \tau \mid \kappa[\mathsf{p}]?\,\lambda(v) \mid \kappa[\mathsf{p}]!\,\lambda(v) \mid \kappa[\mathsf{p}]!\,\lambda(G)$$

Structural congruence. Reduction for networks relies also on a *structural equivalence* and on *evaluation contexts*. The structural equivalence on networks, denoted $\equiv$, defines parallel composition as a commutative, associative operator, and which allows restriction to reduce and enlarge their scope without name clashes. Also, it decrees that any monitored process with the end monitor corresponds to the neutral element for parallel composition and absorbs restriction. More formally:

$$\mathsf{end}[P] \mid N \equiv N \quad (\nu\kappa)\mathsf{end}[P] \equiv \mathsf{end}[P] \quad (\nu\kappa)N_1 \mid N_2 \equiv (\nu\kappa)N_1 \mid N_2$$
$$(\nu\kappa)(\kappa : h) \mid N \equiv N \quad (\nu\kappa)(\nu\kappa')N \equiv (\nu\kappa')(\nu\kappa)N$$

We consider also a structural equivalence for message queues which allows us to commute messages involving different receivers or senders:

$$h \cdot (\mathsf{p}, \mathsf{q}, \lambda_i(v_i)) \cdot (\mathsf{p}', \mathsf{q}', \lambda_j(v_j)) \cdot h' \equiv h \cdot (\mathsf{p}', \mathsf{q}', \lambda_j(v_j)) \cdot (\mathsf{p}, \mathsf{q}, \lambda_i(v_i)) \cdot h' \quad \text{if } \mathsf{p} \neq \mathsf{p}' \text{ or } \mathsf{q} \neq \mathsf{q}'$$

This congruence is extended to networks in the expected way by letting: $h \equiv h'$ implies $\kappa : h \equiv \kappa : h'$.

Finally, we have the following definition for evaluation contexts, denoted $\mathcal{E}$:

$$\mathcal{E} ::= [\,] \mid \mathcal{E} \mid N \mid (\nu\kappa)\mathcal{E}$$

Reduction rules for networks The rules of the reduction semantics for networks are given in Table 7; we now describe them:

- Rule ⟨N:Open⟩ initializes a global type, denoted G in the rule. To that end, it considers all the inactive locations associated to the declared participants of G, denoted Π. For each $\mathsf{p} \in \Pi$, a process implementation is picked up from the local collection $\mathcal{P}_\mathsf{p}$. The process type of this implementation is expected to be adequate to the monitor $\mathsf{G} \downarrow_\mathsf{p}$. Observe that mapping $\mathcal{L}_\mathsf{G}$ ensures that each participant is assigned to a different (inactive) location. A fresh session name κ is created for protocol participants; associated message and event queues are also created.

Table 7. Reduction semantics for multiparty processes

$$\langle \text{N:Open} \rangle \quad \frac{\Pi = \mathtt{part}(\mathsf{G}) \qquad \forall \mathsf{p} \in \Pi.\big(\mathcal{M}_\mathsf{p} = \mathsf{G}\downarrow_\mathsf{p} \wedge (P_\mathsf{p}, T_\mathsf{p}) \in \mathcal{P}_\mathsf{p} \wedge T_\mathsf{p} \propto \mathcal{M}_\mathsf{p} \wedge \mathcal{L}_\mathsf{G}(\mathsf{p}) = \mathtt{loc}_\mathsf{p}\big)}{\mathtt{new}(\mathsf{G}) \mid \prod_{\mathsf{p}\in\Pi} \mathtt{loc}_\mathsf{p}\lceil \mathcal{P}_\mathsf{p};\, \mathtt{loc}_\mathsf{p}\lfloor\epsilon\rfloor\rceil \longrightarrow (\nu\kappa)\big(\prod_{\mathsf{p}\in\Pi} \mathtt{loc}_\mathsf{p}\lceil \mathcal{M}_\mathsf{p}[P_\mathsf{p}[\kappa[\mathsf{p}]/y]];\, \mathcal{P}_\mathsf{p};\, \mathtt{loc}_\mathsf{p}\lfloor\epsilon\rfloor\rceil \mid \kappa : \emptyset \mid \kappa^e\lfloor\epsilon\rfloor\big)}$$

$$\langle \text{N:Send} \rangle \quad \frac{\mathcal{M} \xrightarrow{\mathsf{q}!\lambda} \mathcal{M}' \qquad P \xrightarrow{\kappa[\mathsf{p}]!\lambda(v)} P'}{\mathtt{loc}_\mathsf{p}\lceil \mathcal{M}[P];\, \mathcal{P};\, \mathtt{loc}_\mathsf{p}\lfloor\epsilon\rfloor\rceil \mid \kappa : h \mid \kappa^e\lfloor\epsilon\rfloor \longrightarrow \mathtt{loc}_\mathsf{p}\lceil \mathcal{M}'[P'];\, \mathcal{P};\, \mathtt{loc}_\mathsf{p}\lfloor\epsilon\rfloor\rceil \mid \kappa : (\mathsf{p},\mathsf{q},\lambda(v))\cdot h \mid \kappa^e\lfloor\epsilon\rfloor}$$

$$\langle \text{N:Recv} \rangle \quad \frac{\mathcal{M} \xrightarrow{\mathsf{q}?\lambda} \mathcal{M}' \qquad P \xrightarrow{\kappa[\mathsf{p}]?\lambda(v)} P'}{\mathtt{loc}_\mathsf{p}\lceil \mathcal{M}[P];\, \mathcal{P};\, \mathtt{loc}_\mathsf{p}\lfloor\epsilon\rfloor\rceil \mid \kappa : (\mathsf{q},\mathsf{p},\lambda(v))\cdot h \mid \kappa^e\lfloor\epsilon\rfloor \longrightarrow \mathtt{loc}_\mathsf{p}\lceil \mathcal{M}'[P'];\, \mathcal{P};\, \mathtt{loc}_\mathsf{p}\lfloor\epsilon\rfloor\rceil \mid \kappa : h \mid \kappa^e\lfloor\epsilon\rfloor}$$

$$\langle \text{N:Tau} \rangle \quad \frac{P \xrightarrow{\tau} P'}{\mathtt{loc}\lceil \mathcal{M}[P];\, \mathcal{P};\, \mathtt{loc}\lfloor\epsilon\rfloor\rceil \mid \kappa : h \mid \kappa^e\lfloor\epsilon\rfloor \longrightarrow \mathtt{loc}\lceil \mathcal{M}[P'];\, \mathcal{P};\, \mathtt{loc}\lfloor\epsilon\rfloor\rceil \mid \kappa : h \mid \kappa^e\lfloor\epsilon\rfloor}$$

$$\langle \text{N:InMsg} \rangle \quad \frac{P \xrightarrow{\kappa!(\mathsf{G})} P'}{\mathtt{loc}\lceil \mathcal{M}[P];\mathcal{P};\mathtt{loc}\lfloor\epsilon\rfloor\rceil \mid \kappa^e\lfloor\epsilon\rfloor \mid \kappa : h \longrightarrow \mathtt{loc}\lceil \mathcal{M}[P'];\mathcal{P};\mathtt{loc}\lfloor\epsilon\rfloor\rceil \mid \kappa^e\lfloor\mathsf{G}\rfloor \mid \kappa : h}$$

$$\langle \text{N:InUpd} \rangle \quad \frac{\forall \mathsf{p}_i \in \mathtt{part}(\mathsf{G}').\, \mathsf{InUpd}(\mathsf{p}_i, \Pi, \mathsf{G}', \mathcal{P}_{\mathsf{p}_i}, \kappa, \mathcal{M}_{\mathsf{p}_i}, \mathcal{M}'_{\mathsf{p}_i}, P_{\mathsf{p}_i}, P'_{\mathsf{p}_i})}{(\nu\kappa)\big(\prod_{\mathsf{p}\in\Pi} \mathtt{loc}_\mathsf{p}\lceil \mathcal{M}_\mathsf{p}[P_\mathsf{p}];\mathcal{P}_\mathsf{p};\mathtt{loc}_\mathsf{p}\lfloor\epsilon\rfloor\rceil \mid \kappa : h \mid \kappa^e\lfloor\mathsf{G}'\rfloor\big) \mid \prod_{\mathsf{q}\in\mathtt{part}(\mathsf{G}')\setminus\Pi} \mathtt{loc}_\mathsf{q}\lceil \mathcal{P}_\mathsf{q};\mathtt{loc}_\mathsf{q}\lfloor\epsilon\rfloor\rceil \longrightarrow (\nu\kappa)\big(\prod_{\mathsf{p}\in\mathtt{part}(\mathsf{G}')} \mathtt{loc}_\mathsf{p}\lceil \mathcal{M}'_\mathsf{p}[P'_\mathsf{p}];\mathcal{P}_\mathsf{p};\mathtt{loc}_\mathsf{p}\lfloor\epsilon\rfloor\rceil \mid \kappa : \epsilon \mid \kappa^e\lfloor\epsilon\rfloor\big) \mid \prod_{\mathsf{q}\in\Pi\setminus\mathtt{part}(\mathsf{G}')} \mathtt{loc}_\mathsf{q}\lceil \mathcal{P}_\mathsf{q};\mathtt{loc}_\mathsf{q}\lfloor\epsilon\rfloor\rceil}$$

$$\langle \text{N:AddLoc1} \rangle \quad \frac{\exists (R, T_R) \in \mathcal{P} \qquad T_R \leq T_Q \qquad \mathcal{P}' = \mathcal{P} \cup \{(Q, T_Q)\} \setminus \{(R, T_R)\}}{\mathtt{loc}\lceil \mathcal{M}[P];\mathcal{P};\mathtt{loc}\lfloor(add : \langle Q, T_Q\rangle)\rfloor\rceil \longrightarrow \mathtt{loc}\lceil \mathcal{M}[P];\mathcal{P}';\mathtt{loc}\lfloor\epsilon\rfloor\rceil}$$

$$\langle \text{N:AddLoc2} \rangle \quad \frac{\exists (R, T_R) \in \mathcal{P} \qquad T_R \leq T_Q \qquad \mathcal{P}' = \mathcal{P} \cup \{(Q, T_Q)\} \setminus \{(R, T_R)\}}{\mathtt{loc}\lceil \mathcal{P};\mathtt{loc}\lfloor(add : \langle Q, T_Q\rangle)\rfloor\rceil \longrightarrow \mathtt{loc}\lceil \mathcal{P}';\mathtt{loc}\lfloor\epsilon\rfloor\rceil}$$

$$\langle \text{N:UpdLoc1} \rangle \quad \frac{\mathsf{match}^{\propto}_I(\{\mathcal{M}\}, \{T_i\}_{i\in I}) = l \wedge R = Q_l[\kappa[\mathsf{p}]/y]}{\mathtt{loc}_\mathsf{p}\lceil(\mathcal{M}[P];\mathcal{P};\mathtt{loc}_\mathsf{p}\lfloor(upd : \mathsf{case}\ x\ \mathsf{of}\ \{(T_i) : Q_i\}_{i\in I})\rfloor\rceil \longrightarrow \mathtt{loc}_\mathsf{p}\lceil\mathcal{M}[R];\mathcal{P};\mathtt{loc}_\mathsf{p}\lfloor\epsilon\rfloor\rceil}$$

$$\langle \text{N:UpdLoc2} \rangle \quad \frac{\mathsf{match}^{\propto}_I(\{\mathcal{M}\}, \{T_i\}_{i\in I}) = \uparrow}{\mathtt{loc}_\mathsf{p}\lceil(\mathcal{M}[P];\mathcal{P};\mathtt{loc}_\mathsf{p}\lfloor(upd : \mathsf{case}\ x\ \mathsf{of}\ \{(T_i) : Q_i\}_{i\in I})\rfloor\rceil \longrightarrow \mathtt{loc}_\mathsf{p}\lceil\mathcal{M}[P];\mathcal{P};\mathtt{loc}_\mathsf{p}\lfloor\epsilon\rfloor\rceil}$$

$$\langle \text{N:Equiv} \rangle \quad \frac{N_1 \equiv N'_1 \qquad N_1 \longrightarrow N_2 \qquad N_2 \equiv N'_2}{N'_1 \longrightarrow N'_2} \qquad \langle \text{N:Eval} \rangle \quad \frac{N \longrightarrow N'}{\mathcal{E}[N] \longrightarrow \mathcal{E}[N']}$$

- Rules ⟨N:Send⟩ and ⟨N:Recv⟩ regulate output and input actions from/to the message queue of an established session. The action must be enabled by the label of the monitor, which also describes information on the receiver/sender of the exchanged value. Both monitors and processes evolve as a result of the interaction with the queue. In both rules, a reduction is enabled provided the event queue is empty.
- Rule ⟨N:Tau⟩ formalizes the fact that evaluation of conditional expressions and recursion unfolding within a (monitored) process can occur without affecting its monitor.
- Rules ⟨N:InMsg⟩ and ⟨N:InUpd⟩ formalize *internal adaptation*. Similarly as Rule ⟨N:Tau⟩, issuing an internal adaptation request is a behavior specified by the process but independent from its monitor. Rule ⟨N:InMsg⟩ is enabled if the event queue is empty; it adds a global type to the event queue of the session.

Once a global type is added to the event queue, Rule ⟨N:InUpd⟩ can be triggered. This rule assumes that a fixed set of participants (denoted Π) is already executing a global protocol and should adapt to execute a new protocol G'. The rule sets up local implementations for all participants in $\mathtt{part}(\mathsf{G}')$, also considering the existing behaviors in the set of participants Π. There are three possibilities:

(a) Participant p belongs to both Π and $\mathtt{part}(\mathsf{G}')$ and location $\mathtt{loc}_\mathsf{p}$ encloses a monitored process that "fits" with the projection of G' onto p. In this case, the monitored process is kept as it is.
(b) Participant p belongs to both Π and $\mathtt{part}(\mathsf{G}')$, but the current monitored process enclosed in location $\mathtt{loc}_\mathsf{p}$ is not compatible with the projection of G' onto p. In this case, a new process implementation is selected from the local collection $\mathcal{P}_\mathsf{p}$ and instantiated with an appropriate session name.
(c) Participant p belongs to $\mathtt{part}(\mathsf{G}')$ but not to Π. In this case, the respective location is made active by setting up an appropriate monitored process extracted from the collection.

Consequently, the premise of Rule ⟨N:InUpd⟩ uses the following auxiliary predicate, which formalizes these three possibilities as a disjunction:

$$\begin{array}{l}\mathsf{InUpd}(\mathsf{p},\Pi,\mathsf{G}',\mathcal{P}_\mathsf{p},\kappa,\mathcal{M}_\mathsf{p},\mathcal{M}'_\mathsf{p},P_\mathsf{p},P'_\mathsf{p}) = \\ \quad (\mathsf{p}\in\Pi\cap\mathtt{part}(\mathsf{G}')\ \wedge\ \mathsf{G}'\downarrow_\mathsf{p}=\mathcal{M}_\mathsf{p}=\mathcal{M}'_\mathsf{p}\ \wedge\ P'_\mathsf{p}=P_\mathsf{p})\ \vee \\ \quad (\mathsf{p}\in\Pi\cap\mathtt{part}(\mathsf{G}')\ \wedge\ \mathsf{G}'\downarrow_\mathsf{p}=\mathcal{M}'_p\neq\mathcal{M}_\mathsf{p}\ \wedge\ (Q_\mathsf{p},T_\mathsf{p})\in\mathcal{P}_\mathsf{p}\ \wedge\ T_\mathsf{p}\propto\mathcal{M}'_\mathsf{p}\ \wedge\ P'_\mathsf{p}=Q_\mathsf{p}[^{\kappa[\mathsf{p}]}/_y])\ \vee \\ \quad (\mathsf{p}\in\mathtt{part}(\mathsf{G}')\setminus\Pi\ \wedge\ (Q_\mathsf{p},T_\mathsf{p})\in\mathcal{P}_\mathsf{p}\ \wedge\ \mathsf{G}'\downarrow_\mathsf{p}=\mathcal{M}'_\mathsf{p}\ \wedge\ T_\mathsf{p}\propto\mathcal{M}'_\mathsf{p}\ \wedge\ P'_\mathsf{p}=Q_\mathsf{p}[^{\kappa[\mathsf{p}]}/_y])\end{array}$$

Notice that the left-hand side of the conclusion of Rule ⟨N:InUpd⟩ considers a series of active locations (for participants in Π) together with a series of inactive locations, corresponding to participants in $\mathtt{part}(\mathsf{G}')$ but not in Π, i.e., the locations that will become active as a result of the reduction step (cf. item (c) above). Similarly, the right-hand side of the rule features a set of inactive locations, corresponding to participants present in Π but not in $\mathtt{part}(\mathsf{G}')$; these correspond to participants excluded as a result of adaptation. Also, observe that both message and event queues are emptied after reduction.

- Rules ⟨N:AddLoc1⟩, ⟨N:AddLoc2⟩, ⟨N:UpdLoc1⟩, and ⟨N:UpdLoc2⟩ formalize *external adaptation*. These rules are enabled by the addition of an adaptation request in the location queue. Such a request is meant to come from the context of a location, and contains typed processes.
  - Rule ⟨N:AddLoc1⟩ upgrades a process pair present in the local collection of an active location; this collection is denoted $\mathcal{P}$ in the rule. Rule ⟨N:AddLoc2⟩ is similar but applies to inactive locations. Since we assume that the collection is complete, these rules implement a form of upgrade on the local collection.
  - Rules ⟨N:UpdLoc1⟩ and ⟨N:UpdLoc2⟩ formalize type-directed adaptation for monitored processes, in the spirit of the update processes for binary sessions. The external adaptation message contains a finite series of alternatives to the current behavior of the monitored process. The rule tries to find a compatible update (Rule ⟨N:UpdLoc1⟩); if no suitable update is present then the external request is dismissed, and the monitored process is kept unchanged (Rule ⟨N:UpdLoc2⟩). Notice that compatibility depends on a match predicate, denoted $\mathsf{match}^{\propto}_I(\{\mathcal{M}\},\{T_i\}_{i\in I})$, defined as in Definition 3.1 but considering monitors and process types instead of binary session types, and adequacy rather than subtyping.
- Rules ⟨N:Equiv⟩ and ⟨N:Eval⟩ define self-explanatory treatments for incorporating structural equivalence and evaluation contexts into reduction.

Example **5.8** Consider the example of Sect. 2.2. Using Rule ⟨N:Open⟩ for session establishment we have:

$$\begin{array}{l}\mathtt{new}(\mathsf{G})\mid \mathtt{loc}_\mathsf{b}\big[\mathcal{P}_\mathsf{b};\ \mathtt{loc}_\mathsf{b}\lfloor\epsilon\rfloor\big]\mid \mathtt{loc}_\mathsf{s}\big[\mathcal{P}_\mathsf{s};\ \mathtt{loc}_\mathsf{s}\lfloor\epsilon\rfloor\big]\mid \mathtt{loc}_\mathsf{p}\big[\mathcal{P}_\mathsf{p};\ \mathtt{loc}_\mathsf{p}\lfloor\epsilon\rfloor\big] \\ \longrightarrow \\ (\nu\kappa)(\mathtt{loc}_\mathsf{b}\big[\mathcal{M}_\mathsf{b}[P_\mathsf{b}];\ \mathcal{P}_\mathsf{b};\ \mathtt{loc}_\mathsf{b}\lfloor\epsilon\rfloor\big]\mid \mathtt{loc}_\mathsf{s}\big[\mathcal{M}_\mathsf{s}[P_\mathsf{s}];\ \mathcal{P}_\mathsf{s};\ \mathtt{loc}_\mathsf{s}\lfloor\epsilon\rfloor\big]\mid \mathtt{loc}_\mathsf{p}\big[\mathcal{M}_\mathsf{p}[P_\mathsf{p}];\ \mathcal{P}_\mathsf{p};\ \mathtt{loc}_\mathsf{p}\lfloor\epsilon\rfloor\big]\mid \kappa{:}\emptyset\mid \kappa^e\lfloor\epsilon\rfloor)\end{array}$$

with

$$\begin{array}{l}\mathcal{M}_\mathsf{b} := \mathsf{s}!it(item).\mathsf{s}?price(int).\mathsf{s}!ok(bool).\mathsf{s}!card(int).\mathsf{p}!ad(string) \\ \mathcal{M}_\mathsf{s} := \mathsf{b}?it(item).\mathsf{b}!price(int).\mathsf{b}?ok(bool).\mathsf{b}?card(int).\mathsf{p}!itname(item) \\ \mathcal{M}_\mathsf{p} := \mathsf{b}?ad(string).\mathsf{s}?itname(item)\end{array}$$

Table 8. Projection of generalized types onto participants

$$(p!\lambda(S); m)\downarrow_{\mathsf{q}} = \begin{cases} !\lambda(S).m\downarrow_{\mathsf{q}} & \text{if } \mathsf{p}=\mathsf{q} \\ m\downarrow_{\mathsf{q}} & \text{otherwise} \end{cases} \qquad \epsilon\downarrow_{\mathsf{q}} = \epsilon \quad \mathsf{end}\downarrow_{\mathsf{q}} = \epsilon \quad \langle m, \mathcal{M}\rangle\downarrow_{\mathsf{q}} = (m\downarrow_{\mathsf{q}}).(\mathcal{M}\downarrow_{\mathsf{q}})$$

$$\mathsf{p}?\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I})\downarrow_{\mathsf{q}} = \begin{cases} ?\{\lambda_i(S_i).(\mathcal{M}_i\downarrow_{\mathsf{q}})\}_{i\in I} & \text{if } \mathsf{p}=\mathsf{q} \\ \mathcal{M}_k\downarrow_{\mathsf{q}} & \text{where } k\in I, \mathsf{p}\neq\mathsf{q} \text{ and } \forall i,j\in I\ \mathcal{M}_i=\mathcal{M}_j \end{cases}$$

$$\mathsf{p}!\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I})\downarrow_{\mathsf{q}} = \begin{cases} !\{\lambda_i(S_i).(\mathcal{M}_i\downarrow_{\mathsf{q}})\}_{i\in I} & \text{if } \mathsf{p}=\mathsf{q} \\ \mathcal{M}_k\downarrow_{\mathsf{q}} & \text{where } k\in I, \mathsf{p}\neq\mathsf{q} \text{ and } \forall i,j\in I\ \mathcal{M}_i=\mathcal{M}_j \end{cases}$$

Remark **5.9** (Adaptation functions) We have proposed using processes communicating global types (here denoted $c!\,\lambda(G).P$), rather than adaptation flags, to specify internal adaptation, i.e., adaptation requests issued by the system itself. This is a simple way of formalizing this important class of adaptation routines. As an alternative, a less direct mechanism would consist in communicating a value on which a future adaptation should depend on, rather than the expected new choreography. Such a value could be used as a parameter for the adaptation functions used in [CDV15]. This would add flexibility to the specification of internal adaptation requests.

Having introduced networks and their semantics, we now proceed to investigate type-based analysis techniques for them.

5.3. Safety and consistency

This section generalizes the results of Sect. 4 to the multiparty asynchronous case. In order to lighten the syntax, the types for networks do not include any information on interfaces and only describe an active session. Indeed, as interfaces provide information about the services the location may execute and as each location contains a complete collection of processes $\mathcal{P}$ then the interface type will be the same for each location. We have already described how to type a process, we now show how to type its monitor and relative queues and how to combine them into networks.

As communication is asynchronous the information on the type of a session participant p ($\kappa[\mathsf{p}]$) can be split between the process implementing the session and the associated session queue. We thus have *generalized types*, which can be either a message type, a monitor type or the combination of the two. More precisely:

Definition 5.10 Session types, generalized types, message and queues types are defined by:

$$\begin{array}{lll} \text{Session types} & \Delta & ::= \emptyset \mid \Delta, \kappa[\mathsf{p}] : \chi \\ \text{Generalized types} & \chi & ::= \mathcal{M} \mid \mathcal{Q} \mid \mathsf{m} \\ \text{Queue types} & \mathcal{Q} & ::= \langle \mathsf{m}, \mathcal{M}\rangle \\ \text{Message types} & \mathsf{m} & ::= \epsilon \mid q!\,\lambda(S) \mid \mathsf{m};\ \mathsf{m} \end{array}$$

The typing judgments for networks are of shape

$$\Gamma \vdash_{\Sigma} N \triangleright \Delta$$

where Σ lists the free session names in N. As for the binary case, type safety relies on a notion of balanced typing: communications are performed in the right order and with the proper type. The definition of duality $\bowtie$ is the same as for the binary case (see Definition A.2).

Definition 5.11 A session typing Δ is balanced for the session κ (denoted $\mathsf{bal}(\Delta, \kappa)$) if $\kappa[\mathsf{p}] : \chi \in \Delta$ and $\kappa[\mathsf{q}] : \chi' \in \Delta$ with $\mathsf{p}\neq\mathsf{q}$ imply $\chi\downarrow_{\mathsf{q}} \bowtie \chi'\downarrow_{\mathsf{p}}$ where the projection of generalized types onto participants is given in Table 8.

It is easy to see that session types obtained from the projection of global types are balanced.

Table 9. Typing rules for networks and queues

$$\langle \text{M:New} \rangle \quad \frac{}{\Gamma \vdash_\emptyset \texttt{new}(G) \triangleright \emptyset} \qquad \langle \text{M:End} \rangle \quad \frac{}{\Gamma \vdash_\emptyset \mathsf{end}[P] \triangleright \emptyset}$$

$$\langle \text{M:MP} \rangle \quad \frac{\Gamma \vdash P \triangleright \kappa[\mathsf{p}] : T \qquad \mathcal{M} \neq \mathsf{end} \qquad T \propto \mathcal{M}}{\Gamma \vdash_\emptyset \mathcal{M}[P] \triangleright \kappa[\mathsf{p}] : \mathcal{M}}$$

$$\langle \text{M:CQueue} \rangle \quad \frac{}{\Gamma \vdash_\emptyset \kappa^e \lfloor h \rfloor \triangleright \emptyset} \qquad \langle \text{M:QInit} \rangle \quad \frac{}{\Gamma \vdash_{\{\kappa\}} \kappa : \emptyset \triangleright \emptyset}$$

$$\langle \text{M:QSend} \rangle \quad \frac{\Gamma \vdash_{\{\kappa\}} \kappa : h \triangleright \Delta \qquad \Gamma \vdash v : S}{\Gamma \vdash_{\{\kappa\}} \kappa : (\mathsf{p}, \mathsf{q}, \lambda(v)) \cdot h \triangleright \Delta \# \{\kappa[\mathsf{p}] : \mathsf{q}!\lambda(S)\}}$$

$$\langle \text{M:Par} \rangle \quad \frac{\Gamma \vdash_{\Sigma_1} N_1 \triangleright \Delta_1 \qquad \Gamma \vdash_{\Sigma_2} N_2 \triangleright \Delta_2}{\Gamma \vdash_{\Sigma_1 \cup \Sigma_2} N_1 \mid N_2 \triangleright \Delta_1 \star \Delta_2}$$

$$\langle \text{M:Equiv} \rangle \quad \frac{\Gamma \vdash_\Sigma N \triangleright \Delta \qquad \Delta \approx \Delta'}{\Gamma \vdash_\Sigma N \triangleright \Delta'}$$

$$\langle \text{M:Res} \rangle \quad \frac{\Gamma \vdash_\Sigma N \triangleright \Delta \qquad \mathsf{bal}(\Delta, \kappa)}{\Gamma \vdash_{\Sigma \setminus \{\kappa\}} (\nu\kappa) N \triangleright \Delta \setminus \{\kappa\}}$$

$$\langle \text{M:Loc1} \rangle \quad \frac{\forall (P, T_P) \in \mathcal{P} \quad \Gamma \vdash P \triangleright T_P}{\Gamma \vdash_\emptyset \texttt{loc}\lceil \mathcal{P}; \texttt{loc}\lfloor r \rfloor \rceil \triangleright \emptyset}$$

$$\langle \text{M:Loc2} \rangle \quad \frac{\forall (P, T_P) \in \mathcal{P} \quad \Gamma \vdash P \triangleright T_P \qquad \Gamma \vdash_\emptyset \mathcal{M}[P] \triangleright \kappa[\mathsf{p}] : \mathcal{M}}{\Gamma \vdash_\emptyset \texttt{loc}\lceil \mathcal{M}[P]; \mathcal{P}; \texttt{loc}\lfloor r \rfloor \rceil \triangleright \kappa[\mathsf{p}] : \mathcal{M}}$$

Proposition 5.12 *Let* G *be a global type and* $\mathsf{p} \neq \mathsf{q}$*, then* $(\mathsf{G} \downarrow_\mathsf{p}) \downarrow_\mathsf{q} \bowtie (\mathsf{G} \downarrow_\mathsf{q}) \downarrow_\mathsf{p}$.

We are now ready to complete the typing system with typing rules for networks and queues, given in Table 9. We briefly comment on the rules. A session initiator is typed with the empty session typing (Rule ⟨M:New⟩). If $\mathcal{M} = \mathsf{end}$ then the monitored process $\mathcal{M}[P]$ is typed with the empty session type (Rule ⟨M:End⟩); otherwise, if the type of P is adequate with respect to the monitor then the type of the monitored process is $\kappa[\mathsf{p}] : \mathcal{M}$ (Rule ⟨M:MP⟩). The event queue κ^e is typed with the completed session type (Rule ⟨M:CQueue⟩). Rules ⟨M:QInit⟩ and ⟨QSend⟩ type the session message queue. In particular, Rule ⟨M:QSend⟩ makes use of the operator # that dispatches message types to proper session channels:

$$\begin{aligned} m \# m' &::= m;\ m' \\ \Delta \# \Delta' &::= \{\kappa[\mathsf{p}] : \chi \# \chi' \mid \kappa[\mathsf{p}] : \chi \in \Delta \wedge \kappa[\mathsf{p}] : \chi' \in \Delta'\} \cup \{\kappa[\mathsf{p}] : \chi \mid \kappa[\mathsf{p}] : \chi \in \Delta \cup \Delta' \wedge \kappa[\mathsf{p}] : \chi \notin \Delta \cap \Delta'\} \end{aligned}$$

Rule ⟨M:Par⟩ combines networks; it relies on an operator $\star$, which combines the typing information coming from queues and monitored processes:

$$\begin{aligned} m \star \mathcal{M} &::= \langle m, \mathcal{M} \rangle \\ \mathcal{M} \star m &::= \langle m, \mathcal{M} \rangle \\ \Delta \star \Delta' &::= \{\kappa[\mathsf{p}] : \chi \star \chi' \mid \kappa[\mathsf{p}] : \chi \in \Delta \wedge \kappa[\mathsf{p}] : \chi' \in \Delta'\} \cup \{\kappa[\mathsf{p}] : \chi \mid \kappa[\mathsf{p}] : \chi \in \Delta \cup \Delta' \wedge \kappa[\mathsf{p}] : \chi \notin \Delta \cap \Delta'\} \end{aligned}$$

In order to take into account the structural congruence on queues (Rule ⟨M:Equiv⟩) we build an equivalence relation ($\approx$) on types that is induced by the following equivalence rule on message types:

$$m;\ \mathsf{q}!\lambda(S);\ \mathsf{q}'!\lambda(S');\ m' \approx m;\ \mathsf{q}'!\lambda(S');\ \mathsf{q}!\lambda(S);\ m'$$

with $\mathsf{q} \neq \mathsf{q}'$, and is extended to generalized types by letting if $m \approx m'$ then $\langle m, \mathcal{M} \rangle \approx \langle m', \mathcal{M} \rangle$.

As expected, the type of a restricted network depends on the consistency of the session κ (Rule ⟨M:Res⟩). Finally, Rules ⟨M:Loc1⟩ and ⟨M:Loc2⟩ are used to type locations: as in the binary case, locations do not add any information to the session type.

Example **5.13** Consider Example 5.8, and the process

$$R = \texttt{loc}_\mathsf{b}\lceil \mathcal{M}_\mathsf{b}[P_\mathsf{b}];\ \mathcal{P}_\mathsf{b};\ \texttt{loc}_\mathsf{b}\lfloor \epsilon \rfloor \rceil \mid \texttt{loc}_\mathsf{s}\lceil \mathcal{M}_\mathsf{s}[P_\mathsf{s}];\ \mathcal{P}_\mathsf{s};\ \texttt{loc}_\mathsf{s}\lfloor \epsilon \rfloor \rceil \mid \texttt{loc}_\mathsf{p}\lceil \mathcal{M}_\mathsf{p}[P_\mathsf{p}];\ \mathcal{P}_\mathsf{p};\ \texttt{loc}_\mathsf{p}\lfloor \epsilon \rfloor \rceil \mid \kappa : \emptyset \mid \kappa^e \lfloor \epsilon \rfloor$$

obtained after the session establishment. According to the typing rules, we obtain the following typing judgment

$$\emptyset \vdash_{\{\kappa\}} R \triangleright \kappa[\mathsf{b}] : \langle \emptyset, \mathcal{M}_\mathsf{b} \rangle, \kappa[\mathsf{s}] : \langle \emptyset, \mathcal{M}_\mathsf{s} \rangle, \kappa[\mathsf{p}] : \langle \emptyset, \mathcal{M}_\mathsf{s} \rangle.$$

We now state the *subject reduction* theorem, which guarantees that communications are performed by (located) monitored process in a safe manner, following the protocols prescribed by global types. Similarly as for the case of binary sessions, we introduce a reduction over session types.

Definition 5.14 (*Reduction for typings*) Reduction for typings, denoted $\Delta \longmapsto \Delta'$, is defined by the following rules:

1. $\Delta, \kappa[\mathsf{p}] : \langle h\,,\ \mathsf{q}!\,\lambda(S).\mathcal{M}\rangle \longmapsto \Delta, \kappa[\mathsf{p}] : \langle \mathsf{q}!\,\lambda(S);\ h\,,\ \mathcal{M}\rangle$
2. $\Delta, \kappa[\mathsf{p}] : \langle \mathsf{q}!\,\lambda(S);\ h\,,\ \mathcal{M}\rangle, \kappa[\mathsf{q}] : \langle h'\,,\ \mathsf{p}?\,\lambda(S).\mathcal{M}'\rangle \longmapsto \Delta, \kappa[\mathsf{p}] : \langle h\,,\ \mathcal{M}\rangle, \kappa[\mathsf{q}] : \langle h'\,,\ \mathcal{M}'\rangle$

Theorem 5.15 (Subject reduction) *If* $\Gamma \vdash_\Sigma N \rhd \Delta$ *with* Δ *balanced and* $N \longrightarrow N'$ *then* $\Gamma \vdash_\Sigma N' \rhd \Delta'$, *for some balanced* Δ' *such that either* $\Delta = \Delta'$ *or* $\Delta \longmapsto \Delta'$.

Proof By induction on the last rule applied in the reduction. See Appendix B for details. □

Notice that we can also prove a form of *progress*: i.e., in the absence of internal adaptation events (only external adaptation), every input monitored process will eventually receive a message and conversely every message in a queue will eventually be received by an input monitored process.[1] The proof follows the one presented in [CDV15, CDYP16], which crucially depends on the assumption that a process owns only one session channel.

We now extend *safety* and *consistency* properties to the multiparty case. Similarly as before we define κ-*redexes*, and *error processes*. Major changes are due to the fact that here we consider asynchronous communication.

Definition 5.16 (κ-*redexes, errors*) A network N contains a κ-*redex* if it is of one of the following forms:

(a) $\mathcal{E}\left[\mathtt{loc}_\mathsf{p}\left[\mathcal{M}_{in}[\kappa[\mathsf{p}]?\{\lambda_i(x_i).P_i\}_{i\in I}];\ \mathcal{P}_\mathsf{p};\ loc_\mathsf{p}\lfloor r\rfloor\right] \mid \mathtt{loc}_\mathsf{q}\left[\mathcal{M}_o[\kappa[\mathsf{q}]!\,\lambda(v).P];\ \mathcal{P}_\mathsf{q};\ loc_\mathsf{q}\lfloor r'\rfloor\right]\right]$

(b) $\mathcal{E}\left[\mathtt{loc}\left[\mathcal{M}_{in}[\kappa[\mathsf{p}]?\{\lambda_i(x_i).P_i\}_{i\in I}];\ \mathcal{P};\ loc\lfloor r\rfloor\right] \mid \kappa : (\mathsf{q}, \mathsf{p}, \lambda(v)) \cdot h\right]$

where $\mathcal{M}_{in} = p?\{\lambda_i(S_i).\mathcal{M}_i\}_{i\in I}$, $\mathcal{M}_o = p!\,\lambda(S).\mathcal{M}'$ and there exists $j \in I$ such that $\lambda_j = \lambda$ and $S_j = S$.

A network $N \equiv (\nu\widetilde{\kappa})(N')$ is an *error* if N' does not contain a κ-redex.

The introduction of internal updates breaks the update-consistency property as stated in Sect. 4. This is expected, as internal updates are meant to globally change the communication protocol agreed upon session establishment. Nonetheless, as external updates behave exactly as in the binary case, the property is preserved for them. Below we write $N \longrightarrow_{\text{upd}} N'$ for any reduction inferred using Rule ⟨N:UPDLOC1⟩ (cf. Table 7). We may then define:

Definition 5.17 (*Safety and consistency*) Let N be a network.

- We say N is *safe* if it never reduces into an error.
- We say N is *external update-consistent* if and only if, for all N_0 and κ such that $N \longrightarrow^* N_0$ and N_0 contains a κ-redex, if $N_0 \longrightarrow_{\text{upd}} N_1$ then N_1 contains the same κ-redex.

As before our main result follows as a consequence of Theorem 5.15.

Lemma 5.18 *If* $\Gamma \vdash_\Sigma N \rhd \Delta$ *with* Δ *balanced then network* N *contains no error.*

Proof Follows directly from Definition 5.11. □

Theorem 5.19 (Typing ensures safety and consistency) *If* $\Gamma \vdash_\Sigma N \rhd \Delta$ *with* Δ *balanced then network* N *is safe and external update consistent.*

Proof Safety means that N never reduces into an error. Since Δ is balanced by Lemma 5.18 we know that N is not an error. Theorem 5.15 ensures that reduction preserves balanced typings; therefore, N never reduces to an error.

Proving update-consistency entails showing that for all κ such that N_0 contains a κ-redex, and $N \longrightarrow^* N_0$, if $N_0 \longrightarrow_{\text{upd}} N_1$ then N_1 contains the same κ-redex. By Definition 5.16 N_0, there are two cases. We consider only the case (a), i.e., the redex is of the form

$$\mathcal{E}[\mathtt{loc}_\mathsf{p}\left[\mathcal{M}_{in}[\kappa[\mathsf{p}]?\{\lambda_i(x_i).P_i\}_{i\in I}];\ \mathcal{P}_\mathsf{p};\ loc_\mathsf{p}\lfloor r\rfloor\right] \mid \mathtt{loc}_\mathsf{q}\left[\mathcal{M}_o[\kappa[\mathsf{q}]!\,\lambda(v).P];\ \mathcal{P}_\mathsf{q};\ loc_\mathsf{q}\lfloor r'\rfloor\right]]$$

as case (b) is similar. If Rule ⟨N:UPDLOC1⟩ is applied either $\mathtt{loc}_\mathsf{p}$ or $\mathtt{loc}_\mathsf{q}$ are updated. Without loss of generality suppose that $\mathtt{loc}_\mathsf{p}$ is updated. Now function match guarantees that the type of the updated process P located in $\mathtt{loc}_\mathsf{p}$ is kept unchanged. As P only implements the behavior of participant p, by inversion on typing we know that the κ-process needs to be present also in the updated process. Thus we can conclude that the κ-redex is still present. □

[1] The restriction to external adaptation is expected, as internal updates are meant to modify the protocol for current participants, and so sent messages could be left undelivered.

6. Related work

Many previous works have investigated formal approaches to dynamic/run-time adaptation in different settings; below we describe some of these efforts.

An early work devoted to software process evolution is [JC93], which describes the EPOS approach to evolution and customization. Correctness criteria for adaptive workflows are surveyed in [RRD04]. In the setting of workflows, modifications may occur at the level of schemas or at the level of individual workflow instances. It is natural to represent workflows as (Petri) nets with special characteristics; as such, the focus of the comparison in [RRD04] is on approaches based on Petri nets with different semantics. Two correctness criteria compared are graph equivalence (focused on changes in schemas) and trace equivalence (focused on changes in instances). The work [MGR04] describes a workflow management system that supports rule-based adaptation; it provides automated support for predictive and reactive adaptation strategies. The modeling approach and checked properties in these works are considerably different from those in our work.

The paper [Fox11] develops a model for dynamically adaptive services using COWS (the Calculus for Orchestration of Web Services). In particular, a model for adaptation managers is proposed; it exploits the constructs that COWS provides for timed behaviors, and the associated model checker CMC. The proposed model enables structural and functional modifications: while the former concerns the system configuration graph, the latter involves component replacement. Services are assessed in terms of responsiveness, availability, and reliability. This work does not appeal to static verification based on behavioral types. Also, properties related to protocol conformance, such as safety or consistency, are not explicitly considered in [Fox11]. To the best of our knowledge our work develops the first application of constructs for type-directed checks and event-based communication for specifying and analyzing run-time adaptation for session-based concurrent systems. In fact, although such constructs have been proposed in previous works, their application for run-time adaptation seems to be a new contribution of our work. As such, our work develops an unexplored perspective for existing elements in session-based languages with the aim of enhancing models and reasoning techniques for communication-centric systems which may be updated at run-time.

The combination of static typing and type-directed tests for dynamic reconfiguration is not new. For instance, the work [SC06] studies this combination in the very different setting of object-oriented component programming. As already discussed, we build upon constructs proposed in [HKP+10, KYH11, KYHH16, Kou12] for defining type-directed checks and expressing eventful sessions. The earliest works on eventful sessions, covering theory and implementation issues, are [HKP+10, KYH11]. Kouzapas's Ph.D. thesis [Kou12] provides a unified presentation of the eventful session framework, with case studies including event selectors (a building block in event-driven systems) and transformations between multithreaded and event-driven programs. At the level of types, the work in [Kou12] introduces session set types to support the `typecase` construct.In contrast, we use dynamic type inspection only for run-time adaptation; in [Kou12] `typecase` is part of the process syntax. This choice enables us to retain a standard session type syntax.

Previous works [DP15, AR12, BCH+13, CDV15, CDP14, DGL+14] have addressed forms of run-time adaptation for models of communicating systems based on binary and multiparty session types. As we elaborate next, none of them features the distinctive constructs of our framework: adaptation routines based on type-directed checks on session protocols and eventful constructs that handle and trigger such routines.

We have already mentioned how the present development was motivated as an enhancement to our previous work [DP15], in which update processes do not consider type-directed checks and adaptation of located processes with running sessions is disallowed by typing. While [DP15] addresses binary session types, all of [AR12, BCH+13, CDV15, CDP14, DGL+14] concern models based on multiparty sessions and/or choreographies. The work [AR12] studies dynamic update for message passing programs; a form of consistency for updates over threads is ensured using multiparty session types, following an asynchronous communication discipline. The paper [BCH+13] develops a model of choreographies with interleaved sessions and adaptation as well as an associated endpoint language; however, typing is not addressed. The key ideas of the self-adaptable multiparty sessions in [CDV15] have been summarized in Sect. 2. Our proposal for event-based adaptation for multiparty protocols, described in Sect. 5, departs from the approach in [CDV15] and extends it to cover forms of unanticipated adaptation, including internal and external adaptation requests. Another work derived from [CDV15] is [CDP14] in which adaptation of multiparty protocols is coupled with mechanisms for enforcing secure information flow and access control. As such, it is both technically and conceptually different from the framework that we have developed in Sect. 5. Finally, the paper [DGL+14] describes a framework for programming distributed adaptive applications. The framework relies on a language for choreographies in which adaptation is specified following a rule-based approach.

The work [BCD+13] develops a *monitored* variant of the π-calculus and equips it with a combination of static and dynamic verification techniques based on multiparty session types. The goal is to ensure that a collection of decentralized and possibly monitored local processes properly implement a protocol given as a global type. As in our work, in [BCD+13] we find located processes which represent *principals*; a *network* is a set of principals with a *global queue*—a global transport. The semantics of global specifications induces *assertion environments* which in turn are essential to define the behavior of monitored networks: a network may performed actions allowed by its monitor. Key properties include transparency and safety: while transparency says that a monitored process/network behaves as an unmonitored (but well-behaved) process/network, safety says that a protocol (local and global) is respected.

7. Concluding remarks

In this paper, we have introduced a novel approach to run-time adaptation of communicating processes whose structured protocols adhere to a session type discipline. Our study has addressed the case of protocols abstracted as binary sessions; as a proof of concept, we have also explored the case of adaptation in multiparty sessions with asynchronous communication.

Our approach is based on two kinds of constructs: (i) an update process construct based on type-directed checks and (ii) on constructs for eventful sessions, namely queues equipped with an arrival predicate. This combination of constructs allows us to specify dynamic modifications on the behavior of session-typed processes, therefore providing a basis for reasoning about expressive models of communication-centric systems with run-time adaptation. We may specify *what* should be the content of an adaptation routine, but also *when* it should be triggered. The constructs on which our approach relies are not new to this paper, but have been introduced and throughly studied by Kouzapas et al. in a series of works [HKP+10, KYH11, KYHH16, Kou12]. Such works investigate foundational and practical issues for eventful binary sessions; in our view, this adds significant value to our proposal, as the key syntactic elements on which it stands have been already validated (in theory and in practice) by previous independent studies.

The case of binary sessions represents a well-studied and representative setting, in which the distinguishing aspects of our approach can be cleanly introduced. We identified the strictly necessary eventful process constructs that enhance and refine the mechanisms for run-time adaptation given in [DP13a, DP15]. Adaptation requests are handled via event detectors and queues associated to locations. Our approach enables us to specify rich forms of updates on locations with running sessions; this is a concrete improvement with respect to [DP15], in which updates are only allowed when locations do not enclose running sessions. To rule out update steps that jeopardize running session protocols, we also introduced a type system that ensures communication safety and update consistency for session programs.

Concerning our proof of concept for multiparty sessions, we described how our approach can be integrated on top of the framework for self-adaptable multiparty, asynchronous sessions by Coppo et al. [CDV15]. In this more general setting we may specify internal and external adaptation requests; we rely on a process model which organizes communication and adaptation components in terms of on distributed locations. We notice that expressing both internal and external exceptional events is useful in practice; for instance, both kinds of events coexist in BPMN 2.0 (see, e.g., [DRMR13, Chap. 4]).

Directions for future work include validating our approach in concrete case studies and different settings. We intend to revisit the model of *supervision trees* (a mechanism for fault-tolerance in Erlang) that we gave in [DP13b]. Also, it would be interesting to explore if our approach based on events can be harmonized with the model of security-driven adaptation that have been recently developed in [CDP14], in which read and write violations to security policies determine adaptation events. Finally, it would be worth accommodating existing static techniques for progress (deadlock-freedom) into our typing disciplines for run-time adaptation. We see consistency and progress as orthogonal guarantees; we do not find major obstacles preventing an extension of our typing systems with the existing techniques such as those developed in, e.g., [VV13, PVV14].

Acknowledgements

We are grateful to Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and to the anonymous reviewers, whose remarks and suggestions helped us much to improve the paper. This research was partially supported by a Short-Term Scientific Mission Grant to Di Giusto from COST Action IC1201: Behavioural Types for Reliable Large-

Scale Software Systems. Pérez is also affiliated to the NOVA Laboratory for Computer Science and Informatics (NOVA LINCS Ref. UID/CEC/04516/2013), Universidade Nova de Lisboa, Portugal.

Appendix A: Supplementary definitions

A.1. Structural congruence

Definition A.1 (*Structural congruence*) Structural congruence is the smallest congruence relation on processes that is generated by the following laws:

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$
$$P \mid \mathbf{0} \equiv P \qquad P \equiv Q \text{ if } P \equiv_\alpha Q$$
$$(\nu \mathsf{s})\mathbf{0} \equiv \mathbf{0} \qquad (\nu \mathsf{s})(\nu \mathsf{s}')P \equiv (\nu \mathsf{s}')(\nu \mathsf{s})P$$
$$(\nu \mathsf{s})P \mid Q \equiv (\nu \mathsf{s})(P \mid Q) \ (\text{if } \mathsf{s} \notin \mathsf{fc}(Q) \cup \mathsf{fn}(Q)) \qquad (\nu \mathsf{s})\mathtt{loc}[P] \equiv \mathtt{loc}[(\nu \mathsf{s})P]$$

with $\mathsf{s}, \mathsf{s}', \ldots$ ranges over both names and session channels. The extension of $\equiv$ to contexts is as expected.

A.2. Coinductive subtyping and duality

For all types, define $\mathbf{unfold}(T)$ by recursion on the structure of T:

$$\mathbf{unfold}(\mu\, t.T) = \mathbf{unfold}(T[^{\mu\, t.T}/_t])$$

and $\mathbf{unfold}(T) = T$ otherwise.

The following two definitions (*duality relations* and *type simulation*) are used by Definition A.4. Given an index set $I = \{1, \ldots, m\}$, we use $\&\{n_i : T_i\}_{i \in I}$ and $\oplus\{n_i : T_i\}_{i \in I}$ to abbreviate $\&\{n_1 : T_1, \ldots, n_m : T_m\}$ and $\oplus\{n_1 : T_1, \ldots, n_m : T_m\}$, respectively.

Definition A.2 (*Duality relation*) A relation $\mathcal{R} \subseteq \mathcal{T} \times \mathcal{T}$ is a *duality relation* if $(T, S) \in \mathcal{R}$ implies the following conditions:

1. If $\mathbf{unfold}(T) = \tau$ then $\mathbf{unfold}(S) = \sigma$ and $\tau \leq_{\mathtt{C}} \sigma$ and $\sigma \leq_{\mathtt{C}} \tau$.
2. If $\mathbf{unfold}(T) = \mathbf{end}$ then $\mathbf{unfold}(S) = \mathbf{end}$.
3. If $\mathbf{unfold}(T) = ?(T_2).T_1$ then $\mathbf{unfold}(S) = !(S_2).S_1$ and $(T_1, S_1) \in \mathcal{R}$ and $T_2 \leq_{\mathtt{C}} S_2$ and $S_2 \leq_{\mathtt{C}} T_2$.
4. If $\mathbf{unfold}(T) = !(T_2).T_1$ then $\mathbf{unfold}(S) = ?(S_2).S_1$ and $(T_1, S_1) \in \mathcal{R}$ and $T_2 \leq_{\mathtt{C}} S_2$ and $S_2 \leq_{\mathtt{C}} T_2$.
5. If $\mathbf{unfold}(T) = ?(\tau_1, \ldots, \tau_n).T_1$ then $\mathbf{unfold}(S) = ?(\sigma_1, \ldots, \sigma_n).S_1$ then for all $i \in [1 \ldots n]$, we have that $(T_1, S_1) \in \mathcal{R}$ and $\tau_i \leq_{\mathtt{C}} \sigma_i$ and $\sigma_i \leq_{\mathtt{C}} \tau_i$.
6. If $\mathbf{unfold}(T) = !(\tau_1, \ldots, \tau_n).T_1$ then $\mathbf{unfold}(S) = ?(\sigma_1, \ldots, \sigma_n).S_1$ then for all $i \in [1 \ldots n]$, we have that $(T_1, S_1) \in \mathcal{R}$ and $\tau_i \leq_{\mathtt{C}} \sigma_i$ and $\sigma_i \leq_{\mathtt{C}} \tau_i$.
7. If $\mathbf{unfold}(T) = \&\{n_1 : T_1 \ldots n_m : T_m\}$ then $\mathbf{unfold}(S) = \oplus\{n_1 : S_1 \ldots n_m : S_m\}$ and for all $i \in [1 \ldots m]$, we have that $(T_i, S_i) \in \mathcal{R}$.
8. If $\mathbf{unfold}(T) = \oplus\{n_1 : T_1 \ldots n_m : T_m\}$ then $\mathbf{unfold}(S) = \&\{n_1 : S_1 \ldots n_m : S_m\}$ and for all $i \in [1 \ldots m]$, we have that $(T_i, S_i) \in \mathcal{R}$.

Definition A.3 (*Type simulation*) A relation $\mathcal{R} \subseteq \mathcal{T} \times \mathcal{T}$ is a *type simulation* if $(T, S) \in \mathcal{R}$ implies the following conditions:

1. If $\mathbf{unfold}(T) = \tau$ then $\mathbf{unfold}(S) = \sigma$ and $\tau \leq_{\mathtt{B}} \sigma$.
2. If $\mathbf{unfold}(T) = \mathbf{end}$ then $\mathbf{unfold}(S) = \mathbf{end}$.
3. If $\mathbf{unfold}(T) = ?(T_2).T_1$ then $\mathbf{unfold}(S) = ?(S_2).S_1$ and $(T_1, S_1) \in \mathcal{R}$ and $(T_2, S_2) \in \mathcal{R}$.
4. If $\mathbf{unfold}(T) = !(T_2).T_1$ then $\mathbf{unfold}(S) = !(S_2).S_1$ and $(T_1, S_1) \in \mathcal{R}$ and $(S_2, T_2) \in \mathcal{R}$.

5. If $\mathbf{unfold}(T) = ?(\tau_1, \ldots, \tau_n).T_1$ then $\mathbf{unfold}(S) = ?(\sigma_1, \ldots, \sigma_n).S_1$ then for all $i \in [1 \ldots n]$, we have that $(\tau_i, \sigma_i) \in \mathcal{R}$ and $(T_1, S_1) \in \mathcal{R}$.
6. If $\mathbf{unfold}(T) = !(\tau_1, \ldots, \tau_n).T_1$ then $\mathbf{unfold}(S) = !(\sigma_1, \ldots, \sigma_n).S_1$ then for all $i \in [1 \ldots n]$, we have that $(\sigma_i, \tau_i) \in \mathcal{R}$ and $(T_1, S_1) \in \mathcal{R}$.
7. If $\mathbf{unfold}(T) = \oplus\{n_i : T_i\}_{i \in I}$ then $\mathbf{unfold}(S) = \oplus\{n_j : S_j\}_{j \in J}$ and $I \subseteq J$ for all $i \in I$, we have that $(T_i, S_i) \in \mathcal{R}$.
8. If $\mathbf{unfold}(T) = \&\{n_i : T_i\}_{i \in I}$ then $\mathbf{unfold}(S) = \&\{n_j : S_k\}_{j \in J}$ and $J \subseteq I$ for all $j \in J$, we have that $(T_j, S_j) \in \mathcal{R}$.

Based on the above definitions, we can now define:

Definition A.4 (*Coinductive duality and subtyping*) Let T, S be types.

- The *coinductive duality relation*, denoted $\perp_{\mathsf{c}}$, is defined by $T \perp_{\mathsf{c}} S$ if and only if there exists a duality relation $\mathcal{R}$ such that $(T, S) \in \mathcal{R}$.
- The *coinductive subtyping relation*, denoted $\leq_{\mathsf{c}}$, is defined by $T \leq_{\mathsf{c}} S$ if and only if there exists a type simulation $\mathcal{S}$ such that $(T, S) \in \mathcal{S}$. The extension of $\leq_{\mathsf{c}}$ to typings, written $\Delta \leq_{\mathsf{c}} \Delta'$, arises as expected.

Appendix B: Omitted proofs

B.1. Binary case

The following auxiliary result concerns substitutions for channels, expressions, and process variables. Observe how the case of process variables has been relaxed so as to allow substitution with a process with "smaller" interface (in the sense of $\sqsubseteq$, cf. Definition 4.2). This extra flexibility is in line with the typing rule for located processes (cf. Rule ⟨T:Loc⟩, Table 3), and will be useful later on in proofs.

Lemma B.1 (Substitution Lemma)

1. *If* $\Gamma;\ \Theta \vdash P \triangleright \Delta, x : \alpha;\ \mathcal{I}$ *then* $\Gamma;\ \Theta \vdash P[\kappa^p/x] \triangleright \Delta, \kappa^p : \alpha;\ \mathcal{I}$.
2. *If* $\Gamma, x : \tau;\ \Theta \vdash P \triangleright \Delta;\ \mathcal{I}$ *and* $\Gamma \vdash e \triangleright \tau$ *then* $\Gamma;\ \Theta \vdash P[e/x] \triangleright \Delta;\ \mathcal{I}$.

Proof Easily shown by induction on the structure of P. □

As reduction may occur inside contexts, in proofs it is useful to have *typed contexts*. These are contexts in which the hole has associated typing information—concretely, the typing for processes which may fill in the hole. Defining context requires a simple extension of judgments, in the following way:

$$\mathcal{H};\ \Gamma;\ \Theta \vdash C \triangleright \Delta;\ \mathcal{I}$$

Intuitively, $\mathcal{H}$ contains the description of the type associated to the hole in C. Typing rules are extended in the expected way. Because contexts have a single hole, $\mathcal{H}$ is either empty of has exactly one element. When $\mathcal{H}$ is empty, we write $\Gamma;\ \Theta \vdash P \triangleright \Delta;\ \mathcal{I}$ instead of $\cdot;\ \Gamma;\ \Theta \vdash P \triangleright \Delta;\ \mathcal{I}$. Two additional typing rules are required:

$$\langle \text{T:Hole} \rangle \ \frac{}{\bullet_{\Gamma;\ \Theta \vdash \Delta;\ \mathcal{I}};\ \Gamma;\ \Theta \vdash \bullet \triangleright \Delta;\ \mathcal{I}} \qquad \langle \text{T:Fill} \rangle \ \frac{\bullet_{\Gamma;\ \Theta \vdash \Delta;\ \mathcal{I}};\ \Gamma;\ \Theta \vdash C \triangleright \Delta_1;\ \mathcal{I}_1 \qquad \Gamma;\ \Theta \vdash P \triangleright \Delta;\ \mathcal{I}}{\Gamma;\ \Theta \vdash C\{P\} \triangleright \Delta_1;\ \mathcal{I}_1}$$

Axiom ⟨T:Hole⟩ allows us to introduce typed holes into contexts. In Rule ⟨T:Fill⟩, P is a process (it does not have any holes), and C is a context with a hole of type $\Gamma;\ \Theta \vdash \Delta;\ \mathcal{I}$. The substitution of occurrences of $\bullet$ in C with P, noted $C\{P\}$, is sound as long as the typings of P coincide with those declared in $\mathcal{H}$ for C. We introduce some convenient notation for typed holes.

Notation B.2 Let us use $\mathcal{S}, \mathcal{S}', \ldots$ to range over judgments attached to typed holes. This way, $\bullet_{\mathcal{S}}$ denotes the valid typed hole associated to $\mathcal{S} = \Gamma;\ \Theta \vdash \Delta;\ \mathcal{I}$.

Lemma B.3 *Let* P *and* C *be a process and a typed context such that*

$$\Gamma;\ \Theta \vdash C\{P\} \triangleright \Delta;\ \mathcal{I}$$

is a derivable judgment. There exist $\Delta_1, \mathcal{I}_1$ *such that* (i) $\Gamma;\ \Theta \vdash P \triangleright \Delta_1;\ \mathcal{I}_1$ *is a well-typed process, and* (ii) $\Delta_1 \subseteq \Delta$ *and* $\mathcal{I}_1 \sqsubseteq \mathcal{I}$.

Lemma B.4 *Let C be a context. Suppose $\bullet_{\mathcal{S}};\ \Gamma;\ \Theta \vdash C \triangleright \Delta_C \cup \Delta_{\mathcal{S}};\ \mathcal{I}_C \uplus \mathcal{I}_{\mathcal{S}}$ with $\mathcal{S} = \Gamma;\ \Theta \vdash \Delta_{\mathcal{S}};\ \mathcal{I}_{\mathcal{S}}$ is well-typed. Let $\mathcal{S}' = \Gamma;\ \Theta \vdash \Delta_{\mathcal{S}'};\ \mathcal{I}_{\mathcal{S}'}$. Then*

$$\bullet_{\mathcal{S}'};\ \Gamma;\ \Theta \vdash C \triangleright \Delta_C \cup \Delta_{\mathcal{S}'};\ \mathcal{I}_C \uplus \mathcal{I}_{\mathcal{S}'}$$

is a derivable judgment.

Theorem B.5 (Subject congruence) *If $\Gamma;\ \Theta \vdash P \triangleright \Delta;\ \mathcal{I}$ and $P \equiv Q$ then $\Gamma;\ \Theta \vdash Q \triangleright \Delta;\ \mathcal{I}$.*

Proof By induction on the derivation of $P \equiv Q$, with a case analysis on the last applied rule. □

Theorem B.6 (Subject reduction—Theorem 4.6) *If $\Gamma;\ \Theta \vdash P \triangleright \Delta;\ \mathcal{I}$ with Δ balanced and $P \longrightarrow Q$ then $\Gamma;\ \Theta \vdash Q \triangleright \Delta';\ \mathcal{I}'$, for some $\mathcal{I}' \subseteq \mathcal{I}$ and balanced Δ' such that either $\Delta' \leq_{\mathsf{c}} \Delta$ or $\Delta \longmapsto \Delta'$.*

Proof By induction on the last rule applied in the reduction. We assume that $e \downarrow c$ is a type preserving operation, for every e. We examine only a few interesting cases, namely those for session establishment, runtime update, and intra-session communication; remaining cases are similar or simpler.

1. *Case* ⟨R:Open⟩ From Table 2 we have:

$$C\{u(x:\alpha).P_1\} \mid D\{\overline{u}(y:\beta).P_2\} \longrightarrow (\nu\kappa)\big(C\{P_1[\kappa^+/x] \mid \kappa^+\lfloor\alpha\rfloor\} \mid D\{P_2[\kappa^-/y] \mid \kappa^-\lfloor\beta\rfloor\}\big)$$

with $\alpha \perp_{\mathsf{c}} \beta$. By assumption $\Gamma;\ \Theta \vdash C\{u(x:\alpha).P_1\} \mid D\{\overline{u}(y:\beta).P_2\} \triangleright \Delta;\ \mathcal{I}$ with balanced Δ. Then, by inversion on typing, using Rules ⟨T:Accept⟩, ⟨T:Request⟩, and ⟨T:Par⟩ we infer there exist $\Delta', \mathcal{I}'$ such that

$$\frac{(8)\quad (9)}{\Gamma;\ \Theta \vdash C\{u(x:\alpha).P_1\} \mid D\{\overline{u}(y:\beta).P_2\} \triangleright \Delta;\ \mathcal{I}}$$

where, letting $\Delta = \Delta'_1 \cup \Delta'_2$, subtree (8) is as follows:

$$\dfrac{\bullet_{\mathcal{S}_1};\ \Gamma;\ \Theta \vdash C \triangleright \Delta'_1;\ \mathcal{I}'_1 \uplus u : \alpha_{\mathsf{lin}} \qquad \dfrac{\alpha \leq_{\mathsf{c}} \alpha' \qquad \alpha' \perp_{\mathsf{c}} \beta' \qquad \Gamma \vdash u \triangleright \langle \alpha'_{\mathsf{lin}}, \beta'_{\mathsf{lin}} \rangle \qquad \Gamma;\ \Theta \vdash P_1 \triangleright \Delta_1, x:\alpha;\ \mathcal{I}_1}{\Gamma;\ \Theta \vdash u(x:\alpha).P_1 \triangleright \Delta_1;\ \mathcal{I}_1 \uplus u : \alpha_{\mathsf{lin}}}}{\Gamma;\ \Theta \vdash C\{u(x:\alpha).P_1\} \triangleright \Delta'_1;\ \mathcal{I}'_1 \uplus u : \alpha_{\mathsf{lin}}} \tag{8}$$

with

$$\mathcal{S}_1 = \Gamma;\ \Theta \vdash \Delta_1;\ \mathcal{I}_1 \uplus u : \alpha_{\mathsf{lin}}$$

Then, subtree (9) is as follows:

$$\dfrac{\bullet_{\mathcal{S}_2};\ \Gamma;\ \Theta \vdash D \triangleright \Delta'_2;\ \mathcal{I}'_2 \uplus u : \beta_{\mathsf{lin}} \qquad \dfrac{\beta \leq_{\mathsf{c}} \beta' \qquad \alpha' \perp_{\mathsf{c}} \beta' \qquad \Gamma \vdash u \triangleright \langle \alpha_{\mathsf{lin}}, \beta_{\mathsf{lin}} \rangle \qquad \Gamma;\ \Theta \vdash P_2 \triangleright \Delta_2, y:\beta;\ \mathcal{I}_2}{\Gamma;\ \Theta \vdash \overline{u}(y:\beta).P_2 \triangleright \Delta_2;\ \mathcal{I}_2 \uplus u : \beta_{\mathsf{lin}}}}{\Gamma;\ \Theta \vdash D\{\overline{u}(y:\beta).P_2\} \triangleright \Delta'_2;\ \mathcal{I}'_2 \uplus u : \beta_{\mathsf{lin}}} \tag{9}$$

with

$$\mathcal{S}_2 = \Gamma;\ \Theta \vdash \Delta_2;\ \mathcal{I}_2 \uplus u : \beta_{\mathsf{lin}}$$

By Lemma B.3 we have that $\Delta_1 \subseteq \Delta'_1$ and $\Delta_2 \subseteq \Delta'_2$. We also infer $\mathcal{I}_1 \sqsubseteq \mathcal{I}'_1$, $\mathcal{I}_2 \sqsubseteq \mathcal{I}'_2$, and $\mathcal{I}' \sqsubseteq \mathcal{I}$. Now, using Lemma B.1(1) on judgments for P_1 and P_2, we obtain:

(a) $\Gamma;\ \Theta \vdash P_1[\kappa^+/x] \triangleright \Delta_1, \kappa^+ : \alpha;\ \mathcal{I}_1$.
(b) $\Gamma;\ \Theta \vdash P_2[\kappa^-/y] \triangleright \Delta_2, \kappa^- : \beta;\ \mathcal{I}_2$.

We may now reconstruct the derivation given in (8) using Lemma B.4 and Rule ⟨T:Par⟩:

$$\dfrac{(11) \qquad \dfrac{\Gamma;\ \Theta \vdash P_1[\kappa^+/x] \triangleright \Delta_1, \kappa^+ : \alpha;\ \mathcal{I}_1 \quad \Gamma;\ \Theta \vdash \kappa^+\lfloor\alpha\rfloor \triangleright \kappa^+ : \lfloor\alpha\rfloor;\ \emptyset}{\Gamma;\ \Theta \vdash P_1[\kappa^+/x] \mid \kappa^+\lfloor\alpha\rfloor \triangleright \Delta_1, \kappa^+ : \alpha, \kappa^+ : \lfloor\alpha\rfloor;\ \mathcal{I}_1}}{\Gamma;\ \Theta \vdash C\{P_1[\kappa^+/x] \mid \kappa^+\lfloor\alpha\rfloor\} \triangleright \Delta'_1, \kappa^+ : \alpha, \kappa^+ : \lfloor\alpha\rfloor;\ \mathcal{I}'_1} \tag{10}$$

with

$$\bullet_{\mathcal{S}_3};\ \Gamma;\ \Theta \vdash C \triangleright \Delta'_1, \kappa^+ : \alpha, \kappa^+ : \lfloor\alpha\rfloor;\ \mathcal{I}'_1 \tag{11}$$

and

$$\mathcal{S}_3 = \Gamma;\ \Theta \vdash \Delta_1, \kappa^+ : \alpha, \kappa^+ : \lfloor \alpha \rfloor;\ \mathcal{I}_1.$$

For D, we proceed analogously from (9) and infer:

$$\frac{\bullet_{\mathcal{S}_4};\ \Gamma;\ \Theta \vdash D \triangleright \Delta_2', \kappa^- : \beta, \kappa^- : \lfloor \beta \rfloor;\ \mathcal{I}_2' \qquad \Gamma;\ \Theta \vdash P_2[\kappa^-/y] \mid \kappa^- \lfloor \beta \rfloor \triangleright \Delta_2, \kappa^- : \beta, \kappa^- : \lfloor \beta \rfloor;\ \mathcal{I}_2}{\Gamma;\ \Theta \vdash D\{P_2[\kappa^-/y] \mid \kappa^- \lfloor \beta \rfloor\} \triangleright \Delta_2', \kappa^- : \beta, \kappa^- : \lfloor \beta \rfloor;\ \mathcal{I}_2'} \tag{12}$$

with

$$\mathcal{S}_4 = \Gamma;\ \Theta \vdash \Delta_2, \kappa^- : \beta, \kappa^- : \lfloor \beta \rfloor;\ \mathcal{I}_2.$$

We may finally derive the type for the result of the reduction: using Rules ⟨T:PAR⟩ and ⟨T:CRES⟩ we obtain:

$$\frac{\dfrac{(10) \quad (12)}{\Gamma;\ \Theta \vdash C\{P_1[\kappa^+/x] \mid \kappa^+ \lfloor \alpha \rfloor\} \mid D\{P_2[\kappa^-/y] \mid \kappa^- \lfloor \beta \rfloor\} \triangleright \begin{array}{l} \Delta, \kappa^+ : \alpha, \kappa^- : \beta, \\ \kappa^+ : \lfloor \alpha \rfloor, \kappa^- : \lfloor \beta \rfloor; \\ \mathcal{I}_1' \uplus \mathcal{I}_2' \end{array}}}{\Gamma;\ \Theta \vdash (\nu\kappa) C\{P_1[\kappa^+/x] \mid \kappa^+ \lfloor \alpha \rfloor\} \mid D\{P_2[\kappa^-/y] \mid \kappa^- \lfloor \beta \rfloor\} \triangleright \Delta;\ \mathcal{I}_1' \uplus \mathcal{I}_2'}$$

Notice that Δ is unchanged. This concludes this case.

2. *Case* ⟨R:UPD2⟩ From Table 2 we have:

$$C\{\mathtt{loc}[P]\} \mid D\Big\{\mathtt{loc}\{\mathsf{case}\ x_1, \ldots, x_m\ \mathsf{of}\ \{(\beta_1^i;\ \cdots;\ \beta_m^i) : Q_i\}_{i \in I}\}\Big\} \longrightarrow C\{\mathtt{loc}[V]\} \mid D\{\mathbf{0}\}$$

with $V = Q_l[\kappa_1^p, \ldots, \kappa_m^p / x_1, \ldots, x_m]$. By assumption we have

$$\Gamma;\ \Theta \vdash C\{\mathtt{loc}[P]\} \mid D\Big\{\mathtt{loc}\{\mathsf{case}\ x_1, \ldots, x_m\ \mathsf{of}\ \{(\beta_1^i;\ \cdots;\ \beta_m^i) : Q_i\}_{i \in I}\}\Big\} \triangleright \Delta;\ \mathcal{I}$$

with Δ balanced. Then, by inversion on typing, using
Rules ⟨T:FILL⟩, ⟨T:PAR⟩, ⟨T:ADAPT⟩, and ⟨T:LOC⟩ we infer:

$$\frac{(13) \quad (14)}{\Gamma;\ \Theta \vdash C\{\mathtt{loc}[P]\} \mid D\Big\{\mathtt{loc}\{\mathsf{case}\ x_1, \ldots, x_m\ \mathsf{of}\ \{(\beta_1^i;\ \cdots;\ \beta_m^i) : Q_i\}_{i \in I}\}\Big\} \triangleright \Delta;\ \mathcal{I}}$$

Let $\Delta = \Delta_1' \cup \Delta_2'$ and $\mathcal{I} = \mathcal{I}_1' \uplus \mathcal{I}_2'$, subtree (13) is as follows:

$$\frac{\bullet_{\mathcal{S}_1};\ \Gamma;\ \Theta \vdash C \triangleright \Delta_1';\ \mathcal{I}_1' \qquad \dfrac{\mathcal{I}_1 \sqsubseteq \mathcal{I}_1^* \quad \Theta \vdash \mathtt{loc} \triangleright \mathcal{I}_1^* \quad \Gamma;\ \Theta \vdash P_1 \triangleright \Delta_1;\ \mathcal{I}_1}{\Gamma;\ \Theta \vdash \mathtt{loc}[P] \triangleright \Delta_1;\ \mathcal{I}_1}}{\Gamma;\ \Theta \vdash C\{\mathtt{loc}[P]\} \triangleright \Delta_1';\ \mathcal{I}_1'} \tag{13}$$

with $\mathcal{S}_1 = \Gamma;\ \Theta \vdash \Delta_1;\ \mathcal{I}_1$, and $\mathcal{I}_1 \sqsubseteq \mathcal{I}_1'$ (by Lemma B.3). Subtree (14) is as follows:

$$\frac{\bullet_{\mathcal{S}_2};\ \Gamma;\ \Theta \vdash D \triangleright \Delta_2';\ \mathcal{I}_2' \qquad \dfrac{\Theta \vdash \mathtt{loc} \triangleright \mathcal{I} \quad \forall j \in J,\ \mathcal{I}_j \sqsubseteq \mathcal{I} \quad \Gamma;\ \Theta \vdash Q_i \triangleright \langle x_1 : \beta_1^j \rangle;\ \cdots;\ \langle x_m : \beta_m^j \rangle;\ \mathcal{I}_j}{\Gamma;\ \Theta \vdash \mathtt{loc}\Big\{\mathsf{case}\ x_1, \ldots, x_m\ \mathsf{of}\ \{(\beta_1^i;\ \cdots;\ \beta_m^i) : Q_i\}_{i \in I}\Big\} \triangleright \emptyset;\ \emptyset}}{\Gamma;\ \Theta \vdash D\Big\{\mathtt{loc}\{\mathsf{case}\ x_1, \ldots, x_m\ \mathsf{of}\ \{(\beta_1^i;\ \cdots;\ \beta_m^i) : Q_i\}_{i \in I}\}\Big\} \triangleright \Delta_2';\ \mathcal{I}_2'} \tag{14}$$

with $\mathcal{S}_2 = \Gamma;\ \Theta \vdash \emptyset;\ \emptyset$. We now reconstruct the derivation after the reduction, using Rules ⟨T:PAR⟩, ⟨T:FILL⟩, and Lemma B.4. By derivation (14) we know that

$$\Gamma;\ \Theta \vdash Q_l \triangleright \langle x_1 : \beta_1^l \rangle;\ \cdots;\ \langle x_m : \beta_m^l \rangle;\ \mathcal{I}_l$$

thus applying Lemma B.1(1) we have:

$$\dfrac{\dfrac{\bullet_{\mathcal{S}_4};\ \Gamma;\ \Theta \vdash D \triangleright \Delta_2';\ \mathcal{I}_2' \quad \Gamma;\ \Theta \vdash \mathbf{0} \triangleright \emptyset;\ \emptyset}{\Gamma;\ \Theta \vdash D\{\mathbf{0}\} \triangleright \Delta_2';\ \mathcal{I}_2'}\ (16)}{\Gamma;\ \Theta \vdash C\{V\} \mid D\{\mathbf{0}\} \triangleright \Delta_1'' \cup \Delta_2';\ \mathcal{I}_3'' \uplus \mathcal{I}_2'} \tag{15}$$

$$\dfrac{\bullet_{\mathcal{S}_5};\ \Gamma;\ \Theta \vdash C \triangleright \Delta_1'';\ \mathcal{I}_l' \quad \Gamma;\ \Theta \vdash V \triangleright \langle x_1{:}\beta_1^l\rangle;\ \cdots;\ \langle x_m{:}\beta_m^l\rangle;\ \mathcal{I}_l}{\Gamma;\ \Theta \vdash C\{V\} \triangleright \Delta_1'';\ \mathcal{I}_l'} \tag{16}$$

with $\mathcal{S}_5 = \Gamma;\ \Theta \vdash \langle x_1{:}\beta_1^l\rangle;\ \cdots;\ \langle x_m{:}\beta_m^l\rangle;\ \mathcal{I}_l$. By Lemma B.3 we know $\mathcal{I}_l \sqsubseteq \mathcal{I}_l'$. Moreover by Lemma B.4, and by application of Rule ⟨R:UPD2⟩ we have $\Delta_1'' \leq_{\mathsf{C}} \Delta_1'$. Thus $\Delta_1'' \cup \Delta_2' \leq_{\mathsf{C}} \Delta$. This concludes the analysis for this case.

3. *Case* ⟨R:COM⟩ From Table 2 we have:

$$C\left\{\overline{\kappa}^{\,p}(v).P_1 \mid \kappa^p\lfloor !(\tau).\alpha\rfloor\right\} \mid D\left\{\kappa^{\,\overline{p}}(x).P_2 \mid \kappa^{\overline{p}}\lfloor ?(\tau).\beta\rfloor\right\} \longrightarrow C\left\{P_1 \mid \kappa^p\lfloor\alpha\rfloor\right\} \mid D\left\{P_2[^v/_x] \mid \kappa^{\overline{p}}\lfloor\beta\rfloor\right\}$$

By assumption, we have $\Gamma;\ \Theta \vdash C\left\{\overline{\kappa}^{\,p}(v).P_1 \mid \kappa^p\lfloor !(\tau).\alpha\rfloor\right\} \mid D\left\{\kappa^{\,\overline{p}}(x).P_2 \mid \kappa^{\overline{p}}\lfloor ?(\tau).\beta\rfloor\right\} \triangleright \Delta;\ \mathcal{I}$, with Δ balanced and $(\alpha \perp_{\mathsf{C}} \beta)$. By inversion on typing, using Rules ⟨T:FILL⟩, ⟨T:PAR⟩, ⟨T:IN⟩, and ⟨T:OUT⟩, we infer:

$$\dfrac{(17) \quad (19)}{\Gamma;\ \Theta \vdash C\{\overline{\kappa^{\,p}}(v).P_1 \mid \kappa^p\lfloor !(\tau).\alpha\rfloor\} \mid D\{\kappa^{\,\overline{p}}(x).P_2 \mid \kappa^{\overline{p}}\lfloor ?(\tau).\beta\rfloor\} \triangleright \Delta';\ \mathcal{I}_1' \uplus \mathcal{I}_2'}$$

where:

$$\Delta = \Delta_1' \cup \Delta_2', \kappa^p :!(\tau).\alpha, \kappa^p : \lfloor !(\tau).\alpha\rfloor, \kappa^{\overline{p}} :?(\tau).\beta, \kappa^{\overline{p}} : \lfloor ?(\tau).\beta\rfloor$$
$$\mathcal{I} = \mathcal{I}_1' \uplus \mathcal{I}_2'$$

We have that subtree (17) is as follows:

$$\dfrac{\dfrac{\dfrac{}{\Gamma;\ \Theta \vdash \kappa^p\lfloor !(\tau).\alpha\rfloor \triangleright \kappa^p : \lfloor !(\tau).\alpha\rfloor;\ \emptyset} \quad \dfrac{\Gamma;\ \Theta \vdash P_1 \triangleright \Delta_1, \kappa^p : \alpha;\ \mathcal{I}_1 \quad \Gamma \vdash v : \tau}{\Gamma;\ \Theta \vdash \overline{\kappa^p}(v).P_1 \triangleright \Delta_1, \kappa^p :!(\tau).\alpha;\ \mathcal{I}_1}}{\Gamma;\ \Theta \vdash \overline{\kappa^p}(v).P_1 \mid \kappa^p\lfloor !(\tau).\alpha\rfloor \triangleright \Delta_1, \kappa^p :!(\tau).\alpha, \kappa^p : \lfloor !(\tau).\alpha\rfloor;\ \mathcal{I}_1}\ (18)}{\Gamma;\ \Theta \vdash C\{\overline{\kappa^p}(v).P_1 \mid \kappa^p\lfloor !(\tau).\alpha\rfloor\} \triangleright \Delta_1', \kappa^p :!(\tau).\alpha, \kappa^p : \lfloor !(\tau).\alpha\rfloor;\ \mathcal{I}_1'} \tag{17}$$

with

$$\bullet_{\mathcal{S}_1};\ \Gamma;\ \Theta \vdash C \triangleright \Delta_1', \kappa^p :!(\tau).\alpha, \kappa^p : \lfloor !(\tau).\alpha\rfloor \mathcal{I}_1'; \tag{18}$$

and $\mathcal{S}_1 = \Gamma;\ \Theta \vdash \Delta_1, \kappa^p :!(\tau).\alpha, \kappa^p : \lfloor !(\tau).\alpha\rfloor;\ \mathcal{I}_1$. Similarly, for subtree (19) we obtain (we show only the last step of the derivation):

$$\dfrac{(20) \quad \Gamma;\ \Theta \vdash \kappa^{\overline{p}}(x).P_2 \mid \kappa^{\overline{p}}\lfloor ?(\tau).\beta\rfloor \triangleright \Delta_2, \kappa^{\overline{p}} :?(\tau).\beta, \kappa^{\overline{p}} : \lfloor ?(\tau).\beta\rfloor;\ \mathcal{I}_2}{\Gamma;\ \Theta \vdash D\{\kappa^{\overline{p}}(x).P_2 \mid \kappa^{\overline{p}}\lfloor ?(\tau).\beta\rfloor\} \triangleright \Delta_2', \kappa^{\overline{p}} :?(\tau).\beta, \kappa^{\overline{p}} : \lfloor ?(\tau).\beta\rfloor;\ \mathcal{I}_2'} \tag{19}$$

with

$$\bullet_{\mathcal{S}_2};\ \Gamma;\ \Theta \vdash D \triangleright \Delta_2', \kappa^{\overline{p}} :?(\tau).\beta, \kappa^{\overline{p}} : \lfloor ?(\tau).\beta\rfloor;\ \mathcal{I}_2' \tag{20}$$

and

$$\mathcal{S}_2 = \Gamma;\ \Theta \vdash \Delta_2, \kappa^{\overline{p}} :?(\tau).\beta, \kappa^{\overline{p}} : \lfloor ?(\tau).\beta\rfloor;\ \mathcal{I}_2$$

where Lemma B.3 ensures $\Delta_1 \subseteq \Delta_1'$, $\Delta_2 \subseteq \Delta_2'$.
Now, by Lemma B.1(2) we know $\Gamma;\ \Theta \vdash P_2[^v/_x] \triangleright \Delta_2, \kappa^{\overline{p}} : \beta;\ \mathcal{I}_2$. Moreover by Lemma B.4(3) and Rules ⟨T:PAR⟩ and ⟨T:FILL⟩ we obtain the following type derivations:

$$\dfrac{\bullet_{\mathcal{S}_3};\ \Gamma;\ \Theta \vdash C \triangleright \Delta_1', \kappa^p : \alpha, \kappa^p : \lfloor\alpha\rfloor;\ \mathcal{I}_1' \quad \Gamma;\ \Theta \vdash P_1 \mid \kappa^p\lfloor\alpha\rfloor \triangleright \Delta_1, \kappa^p : \alpha, \kappa^p : \lfloor\alpha\rfloor;\ \mathcal{I}_1}{\Gamma;\ \Theta \vdash C\{P_1 \mid \kappa^p\lfloor\alpha\rfloor\} \triangleright \Delta_1', \kappa^p : \alpha, \kappa^p : \lfloor\alpha\rfloor;\ \mathcal{I}_1'} \tag{21}$$

$$\dfrac{\bullet_{\mathcal{S}_4};\ \Gamma;\ \Theta \vdash D \triangleright \Delta_2', \kappa^{\overline{p}} : \beta, \kappa^{\overline{p}} : \lfloor\beta\rfloor;\ \mathcal{I}_2' \quad \Gamma;\ \Theta \vdash P_2[^v/_x] \mid \kappa^{\overline{p}}\lfloor\beta\rfloor \triangleright \Delta_2, \kappa^{\overline{p}} : \beta, \kappa^{\overline{p}} : \lfloor\beta\rfloor;\ \mathcal{I}_2}{\Gamma;\ \Theta \vdash D\{P_2[^v/_x] \mid \kappa^{\overline{p}}\lfloor\beta\rfloor\} \triangleright \Delta_2', \kappa^{\overline{p}} : \beta, \kappa^{\overline{p}} : \lfloor\beta\rfloor;\ \mathcal{I}_2'} \tag{22}$$

$$\frac{(21) \quad (22)}{\Gamma;\ \Theta \vdash C\{P_1\} \mid D\{P_2[v/x]\} \triangleright \Delta_1' \cup \Delta_2', \kappa^p : \alpha, \kappa^{\overline{p}} : \beta, \kappa^p : \lfloor \alpha \rfloor, \kappa^{\overline{p}} : \lfloor \beta \rfloor;\ \mathcal{I}_1' \uplus \mathcal{I}_2'}$$

with

$\mathcal{S}_3 = \Gamma;\ \Theta \vdash \Delta_1, \kappa^p : \alpha, \kappa^p : \lfloor \alpha \rfloor;\ \mathcal{I}_1$
$\mathcal{S}_4 = \Gamma;\ \Theta \vdash \Delta_2, \kappa^{\overline{p}} : \beta, \kappa^{\overline{p}} : \lfloor \beta \rfloor;\ \mathcal{I}_2$
$\mathcal{S}_5 = \Gamma;\ \Theta \vdash \Delta_1' \cup \Delta_2', \kappa^p : \alpha, \kappa^{\overline{p}} : \beta, \kappa^p : \lfloor \alpha \rfloor, \kappa^{\overline{p}} : \lfloor \beta \rfloor;\ \mathcal{I}_1' \uplus \mathcal{I}_2'$

Since by inductive hypothesis Δ_1' and Δ_2' are balanced, we infer that $\Delta_1' \cup \Delta_2', \kappa^p : \alpha, \kappa^{\overline{p}} : \beta$ is balanced as well and $\Delta \longmapsto \Delta_1' \cup \Delta_2', \kappa^p : \alpha, \kappa^{\overline{p}} : \beta$ [cf. Definition 4.5(1)]. This concludes the proof for this case.

B.2. Multiparty case □

As in the binary case, the proof for multiparty sessions relies on an Inversion Lemma:

Lemma B.7 (Inversion Lemma for networks) *Let P and N be a process and a network, respectively. We have:*

1. *If* $\Gamma \vdash_\Sigma \mathtt{new}(\mathsf{G}) \triangleright \Delta$ *then* $\Sigma = \Delta = \emptyset$.
2. *If* $\Gamma \vdash_\Sigma \mathsf{end}[P] \triangleright \Delta$ *then* $\Sigma = \Delta = \emptyset$.
3. *If* $\Gamma \vdash_\Sigma \mathcal{M}[P] \triangleright \Delta$ *and* $\mathcal{M} \neq \mathsf{end}$ *then* $\Sigma = \emptyset$, $\Delta = \kappa[\mathsf{p}] : \mathcal{M}$, $\Gamma \vdash P \triangleright \kappa[\mathsf{p}] : T$ *and* $T \propto \mathcal{M}$.
4. *If* $\Gamma \vdash_\Sigma \kappa^e \lfloor h \rfloor \triangleright \Delta$ *then* $\Sigma = \Delta = \emptyset$.
5. *If* $\Gamma \vdash_\Sigma \kappa : \emptyset \triangleright \Delta$ *then* $\Sigma = \{\kappa\}$ *and* $\Delta = \emptyset$.
6. *If* $\Gamma \vdash_\Sigma \kappa : (\mathsf{p}, \mathsf{q}, \lambda(v)) \cdot h \triangleright \Delta$ *then* $\Sigma = \{\kappa\}$, $\Delta \approx \Delta' \# \{\kappa[\mathsf{p}] : \mathsf{q}!\,\lambda(S)\}$, $\Gamma \vdash_{\{\kappa\}} \kappa : h \triangleright \Delta'$ *and* $\Gamma \vdash v : S$.
7. *If* $\Gamma \vdash_\Sigma N_1 \mid N_2 \triangleright \Delta$ *then* $\Sigma = \Sigma_1 \cup \Sigma_2$, $\Delta \approx \Delta_1 \star \Delta_2$, $\Gamma \vdash_{\Sigma_1} N_1 \triangleright \Delta_1$ *and* $\Gamma \vdash_{\Sigma_2} N_2 \triangleright \Delta_2$.
8. *If* $\Gamma \vdash_\Sigma (\nu\kappa)N \triangleright \Delta$ *then* $\Sigma = \Sigma' \setminus \{\kappa\}$, $\Delta \approx \Delta' \setminus \{\kappa\}$, *and* $\Gamma \vdash_{\Sigma'} N \triangleright \Delta'$ *and* $\mathsf{bal}(\Delta', \kappa)$.
9. *If* $\Gamma \vdash_\Sigma \mathtt{loc}\big[\mathcal{P};\ \mathtt{loc}\lfloor r \rfloor\big] \triangleright \Delta$ *then* $\Sigma = \Delta = \emptyset$, *and* $\forall (P, T_P) \in \mathcal{P}$, $\Gamma \vdash P \triangleright T_P$.
10. *If* $\Gamma \vdash_\Sigma \mathtt{loc}\big[\mathcal{M}[P];\ \mathcal{P};\ \mathtt{loc}\lfloor r \rfloor\big] \triangleright \Delta$ *then*
$\Sigma = \emptyset$, $\Delta = \kappa[\mathsf{p}] : \mathcal{M}$, $\forall (P, T_P) \in \mathcal{P}$, $\Gamma \vdash P \triangleright T_P$ *and* $\Gamma \vdash_\emptyset \mathcal{M}[P] \triangleright \kappa[\mathsf{p}] : \mathcal{M}$.

Proof Follows by the typing system definition (cf. Table 9). □

We also need to prove the Substitution Lemma that allow us to type processes with variable substitutions.

Lemma B.8 (Substitution Lemma) *Let P be a process.*

1. *If* $\Gamma \vdash P \triangleright T$ *then* $\Gamma \vdash P[\kappa[\mathsf{q}]/y] \triangleright T$.
2. *If* $\Gamma, x : S \vdash P \triangleright T$ *and* $\Gamma \vdash v \triangleright S$ *then* $\Gamma, x : S \vdash P[v/x] \triangleright T$.

Proof Easily shown by induction on the structure of P. □

Finally, the following lemma takes care of typability as preserved by Rule ⟨M:Equiv⟩:

Lemma B.9 (Subject congruence) *If* $\Gamma \vdash_\Sigma N \triangleright \Delta$ *and* $N \equiv N'$ *then* $\Gamma \vdash_\Sigma N' \triangleright \Delta$.

Proof By induction on the derivation of $N \equiv N'$, with a case analysis on the last applied rule. □

We can finally state our main theorem:

Theorem B.10 (Subject reduction—Theorem 5.15) *If* $\Gamma \vdash_\Sigma N \triangleright \Delta$ *with* Δ *balanced and* $N \longrightarrow N'$ *then* $\Gamma \vdash_\Sigma N' \triangleright \Delta'$, *for some balanced* Δ' *such that either* $\Delta = \Delta'$ *or* $\Delta \longmapsto \Delta'$.

Proof By case analysis and induction on the reduction rules for networks (cf. Table 7). The cases can be divided into three families:

1. Session establishment: Rule ⟨N:Open⟩;
2. Intra-session communication: Rules ⟨N:Send⟩, ⟨N:Recv⟩, ⟨N:Tau⟩, and ⟨N:Inmsg⟩;
3. Adaptation: Rules ⟨N:InUpd⟩, ⟨N:AddLoc1⟩, ⟨N:AddLoc2⟩, ⟨N:UpdLoc1⟩ and ⟨N:UpdLoc2⟩.

We give the proof for each one of these families:

1. In this case $N \equiv \mathtt{new}(\mathsf{G}) \mid \mathcal{L}oc \mid N$ with $\mathcal{L}oc = \prod_{\mathsf{p}\in\mathsf{G}} \mathtt{loc}_{\mathsf{p}}\big[\mathcal{P}_{\mathsf{p}};\ \mathtt{loc}_{\mathsf{p}}\lfloor\epsilon\rfloor\big]$ by hypothesis and by the Inversion Lemma we know that $\Gamma \vdash_{\emptyset} \mathtt{new}(\mathsf{G}) \mid \mathcal{L}oc \rhd \emptyset$ and $\Gamma \vdash_{\Sigma} N \rhd \Delta$. After applying Rule ⟨N:Open⟩ we have

$$N' \equiv (\nu\kappa)\left(\prod_{\mathsf{p}\in\Pi} \mathtt{loc}_{\mathsf{p}}\big[\mathcal{M}_{\mathsf{p}}[\![P_{\mathsf{p}}[\kappa[\mathsf{p}]/y]]\!];\ \mathcal{P}_{\mathsf{p}};\ \mathtt{loc}_{\mathsf{p}}\lfloor\epsilon\rfloor\big] \mid \kappa : \emptyset \mid \kappa^{e}\lfloor\epsilon\rfloor\right) \mid N.$$

By hypothesis we know that $\Gamma \vdash_{\Sigma} N \rhd \Delta$. By Rules ⟨M:Qinit⟩ and ⟨M:CQueue⟩ we have $\Gamma \vdash_{\kappa} \kappa : \emptyset \rhd \emptyset$ and $\Gamma \vdash_{\emptyset} \kappa^{e}\lfloor\epsilon\rfloor \rhd \emptyset$. Now for each $\mathsf{p} \in \Pi$ by applying the Inversion Lemma (Lemma B.7) and the substitution Lemma (Lemma B.8) we have:

$$\frac{\forall(P,\ T_P) \in \mathcal{P}_{\mathsf{p}} \quad \Gamma \vdash P \triangleright T_P \qquad \dfrac{\Gamma \vdash_{\emptyset} P[\kappa[\mathsf{p}]/y] \rhd \kappa[\mathsf{p}] : T_P \qquad \mathcal{M}_{\mathsf{p}} \neq \mathsf{end} \qquad T_P \propto \mathcal{M}_{\mathsf{p}}}{\Gamma \vdash_{\emptyset} \mathcal{M}_{\mathsf{p}}[\![P[\kappa[\mathsf{p}]/y]]\!] \rhd \kappa[\mathsf{p}] : \mathcal{M}_{\mathsf{p}}}}{\Gamma \vdash_{\emptyset} \mathtt{loc}_{\mathsf{p}}\big[\mathcal{M}_{\mathsf{p}}[\![P_{\mathsf{p}}[\kappa[\mathsf{p}]/y]]\!];\ \mathcal{P}_{\mathsf{p}};\ \mathtt{loc}_{\mathsf{p}}\lfloor\epsilon\rfloor\big] \rhd \kappa[\mathsf{p}] : \mathcal{M}_{\mathsf{p}}}$$

Finally, we apply Rules ⟨M:Par⟩ and ⟨M:Res⟩, noticing that by Proposition 5.12 the type of

$$\prod_{\mathsf{p}\in\Pi} \mathtt{loc}_{\mathsf{p}}\big[\mathcal{M}_{\mathsf{p}}[\![P_{\mathsf{p}}[\kappa[\mathsf{p}]/y]]\!];\ \mathcal{P}_{\mathsf{p}};\ \mathtt{loc}_{\mathsf{p}}\lfloor\epsilon\rfloor\big]$$

is balanced for κ; we then have that $\Gamma \vdash_{\emptyset} (\nu\kappa)(\prod_{\mathsf{p}\in\Pi} \mathtt{loc}_{\mathsf{p}}\big[\mathcal{M}_{\mathsf{p}}[\![P_{\mathsf{p}}[\kappa[\mathsf{p}]/y]]\!];\ \mathcal{P}_{\mathsf{p}};\ \mathtt{loc}_{\mathsf{p}}\lfloor\epsilon\rfloor\big] \mid \kappa : \emptyset \mid \kappa^{e}\lfloor\epsilon\rfloor) \rhd \emptyset$. Thus we conclude $\Gamma \vdash_{\Sigma} N' \rhd \Delta$ and Δ is unchanged thus balanced by hypothesis.

2. We consider only the case of Rule ⟨N:Send⟩; the other cases are similar or simpler.
We have that $N \equiv \mathcal{E}[\mathtt{loc}\big[\mathcal{M}[\![P]\!];\ \mathcal{P};\ \mathtt{loc}\lfloor\epsilon\rfloor\big] \mid \kappa : (\mathsf{p},\mathsf{q},\lambda(v)) \cdot h \mid \kappa^{e}\lfloor\epsilon\rfloor]$. By Rules ⟨M:QSend⟩ and ⟨M:Par⟩ we have that

$$\Gamma \vdash_{\kappa} \mathtt{loc}\big[\mathcal{M}[\![P]\!];\ \mathcal{P};\ \mathtt{loc}\lfloor\epsilon\rfloor\big] \mid \kappa : (\mathsf{p},\mathsf{q},\lambda(v)) \cdot h \mid \kappa^{e}\lfloor\epsilon\rfloor \rhd \Delta_1$$

with $\Delta_1 = \{\kappa[\mathsf{q}] : \mathcal{M}\} \star (\Delta_2 \# \{\kappa[\mathsf{p}] : \mathsf{q}!\,\lambda(S)\})$. By hypothesis we have that there exists Δ_0 such that $\Delta = \Delta_0 \star \Delta_1$ and Δ balanced.
By applying Lemmas B.7 and B.8 and Rule ⟨<:Par⟩ we have:

$$\Gamma \vdash_{\kappa} \mathtt{loc}\big[\mathcal{M}[\![P']\!];\ \mathcal{P};\ \mathtt{loc}\lfloor\epsilon\rfloor\big] \mid \kappa : h \mid \kappa^{e}\lfloor\epsilon\rfloor \rhd \Delta_1'$$

with $\Delta_1' = \{\kappa[\mathsf{q}] : \mathcal{M}'\} \star \Delta_2$, it is easy to see that $\Delta' = \Delta_0 \star \Delta_1'$ is still balanced and that $\Delta \longmapsto \Delta'$ [cf. Definition 5.14(1)].

3. We give details of the proof for Rules ⟨N:InUpd⟩ and ⟨N:UpdLoc1⟩ the other cases are similar or simpler.
   – The case of Rule ⟨N:InUpd⟩ is similar to the one of Rule ⟨N:Open⟩ above. Let

$$N \equiv (\nu\kappa)\left(\prod_{\mathsf{p}\in\Pi} \mathtt{loc}_{\mathsf{p}}\big[\mathcal{M}_{\mathsf{p}}[\![P_{\mathsf{p}}]\!];\ \mathcal{P}_{\mathsf{p}};\ \mathtt{loc}_{\mathsf{p}}\lfloor\epsilon\rfloor\big] \mid \kappa : h \mid \kappa^{e}\lfloor\mathsf{G}'\rfloor\right) \mid \prod_{\mathsf{q}\in\Pi'\setminus\Pi} \mathtt{loc}_{\mathsf{q}}\big[\mathcal{P}_{\mathsf{q}};\ \mathtt{loc}_{\mathsf{q}}\lfloor\epsilon\rfloor\big]$$

where $\Pi' = \mathtt{part}(\mathsf{G}')$. We have $\Gamma \vdash_{\emptyset} N \rhd \emptyset$ and after the application of Rule ⟨N:InUpd⟩ we also have

$$N' \equiv (\nu\kappa)\left(\prod_{\mathsf{p}\in\Pi'} \mathtt{loc}_{\mathsf{p}}\big[\mathcal{M}'_{\mathsf{p}}[\![P'_{\mathsf{p}}]\!];\ \mathcal{P}_{\mathsf{p}};\ \mathtt{loc}_{\mathsf{p}}\lfloor\epsilon\rfloor\big] \mid \kappa : \epsilon \mid \kappa^{e}\lfloor\epsilon\rfloor\right) \mid \prod_{\mathsf{q}\in\Pi\setminus\Pi'} \mathtt{loc}_{\mathsf{q}}\big[\mathcal{P}_{\mathsf{q}};\ \mathtt{loc}_{\mathsf{q}}\lfloor\epsilon\rfloor\big]$$

By Rules ⟨M:Loc1⟩ and ⟨M:Par⟩ we know that

$$\Gamma \vdash_{\epsilon} \prod_{\mathsf{q}\in\Pi\setminus\Pi'} \mathtt{loc}_{\mathsf{q}}\big[\mathcal{P}_{\mathsf{q}};\ \mathtt{loc}_{\mathsf{q}}\lfloor\epsilon\rfloor\big] \rhd \emptyset.$$

By Proposition 5.12 and as queue κ has been emptied, the type Δ of $\prod_{\mathsf{p}\in\Pi'} \mathtt{loc}_{\mathsf{p}}\big[\mathcal{M}'_{\mathsf{p}}[\![P'_{\mathsf{p}}]\!];\ \mathcal{P}_{\mathsf{p}};\ \mathtt{loc}_{\mathsf{p}}\lfloor\epsilon\rfloor\big]$ is balanced; therefore, by applying typing rules ⟨M:Par⟩ and ⟨M:Res⟩ we can conclude $\Gamma \vdash_{\emptyset} N' \rhd \emptyset$.

– For Rule ⟨N:UPDLOC1⟩, let

$$N \equiv \mathcal{E}[\mathtt{loc}_{\mathtt{p}}\big[(\mathcal{M}[P];\ \mathcal{P};\ \mathtt{loc}_{\mathtt{p}}\lfloor(upd : \mathsf{case}\ x\ \mathsf{of}\ \{(T_i) : Q_i\}_{i\in I})\rfloor\big]]$$

and $\Gamma \vdash_\Sigma N \triangleright \Delta$ with Δ balanced. After the application of Rule ⟨N:UPDLOC1⟩ we have

$$N' \equiv \mathtt{loc}_{\mathtt{p}}\big[\mathcal{M}[R];\ \mathcal{P};\ \mathtt{loc}_{\mathtt{p}}\lfloor\epsilon\rfloor\big].$$

Now notice that the substitution of process P with R has no influence on the type of the system N' as the reduction rule guarantees that the type is left unchanged (cf. premise $\mathsf{match}^{\infty}_{I}(\{\mathcal{M}\}, \{T_R\})$. Thus $\Gamma \vdash_\Sigma N' \triangleright \Delta$, which concludes the proof. □

References

[AR12] Anderson G, Rathke J (2012) Dynamic software update for message passing programs. In: Jhala R, Igarashi A (eds) Programming Languages and Systems—10th Asian Symposium, APLAS. Proceedings, vol. 7705 of Lecture Notes in Computer Science, pages 207–222. Springer, Kyoto, pp 207–222

[BCD+13] Bocchi L, Chen TC, Demangeon R, Honda K, Yoshida N (2013) Monitoring networks through multiparty session types. In: Beyer D, Boreale M (eds) Formal Techniques for Distributed Systems—Joint IFIP WG 6.1 International Conference, FMOODS/FORTE, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec. Proceedings, vol. 7892 of Lecture Notes in Computer Science. Springer, Florence, pp 50–65

[BCH+13] Bravetti M, Carbone M, Hildebrandt TT, Lanese I, Mauro J, Pérez JA, Zavattaro G (2013) Towards global and local types for adaptation. In: Counsell S, Núñez M (eds) Software Engineering and Formal Methods—SEFM Collocated Workshops, vol. 8368 of Lecture Notes in Computer Science. Springer, pp 3–14

[BDPZ12] Bravetti M, Di Giusto C, Pérez JA, Zavattaro G (2012) Adaptable Processes. Log. Methods Comput. Sci. 8(4)

[CDP14] Castellani I, Dezani-Ciancaglini M, Pérez JA (2014) Self-adaptation and secure information flow in multiparty structured communications: a unified perspective. In: Carbone M (ed) Proceedings Third Workshop on Behavioural Types, BEAT, vol. 162 of EPTCS. Rome, pp 9–18

[CDV15] Coppo M, Dezani-Ciancaglini M, Venneri B (2014) Self-adaptive multiparty sessions. Serv Orient Comput Appl 9(3-4):249–268

[CDYP16] Coppo M, Dezani-Ciancaglini M, Yoshida N, Padovani L (2016) Global progress for dynamically interleaved multiparty sessions. Math Struct Comput Sci 26(2):238–302

[DGL+14] Dalla Preda M, Giallorenzo S, Lanese I, Mauro J, Gabbrielli M (2014) AIOCJ: a choreographic framework for safe adaptive distributed applications. In: Combemale B, Pearce DJ, Barais O, Vinju JJ (eds) Software Language Engineering—7th International Conference, SLE, vol. 8706 of Lecture Notes in Computer Science. Springer, pp 161–170

[DP13a] Di Giusto C, Pérez JA (2013) Disciplined structured communications with consistent runtime adaptation. In: Shin SY, Maldonado JC (eds) Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC. ACM, Coimbra 1913–1918

[DP13b] Di Giusto C, Pérez JA (2013) Session types with runtime adaptation: overview and examples. In: PLACES, vol. 137 of EPTCS, pp 21–32

[DP15] Di Giusto C, Pérez JA (2015) Disciplined structured communications with disciplined runtime adaptation. Sci Comput Program 97:235–265

[DP16] Di Giusto C, Pérez JA (2016) An event-based approach to runtime adaptation in communication-centric systems. In: Hildebrandt T, Ravara A, van der Werf JM, Weidlich M (eds) Web services, formal methods, and behavioral types: 11th international workshop, WS-FM 2014, Eindhoven, The Netherlands, September 11–12, 2014, and 12th international workshop, WS-FM/BEAT 2015, Madrid, Spain, September 4–5, 2015, Revised Selected Papers. Springer International Publishing, Cham, pp 67–85. doi:10.1007/978-3-319-33612-1

[DRMR13] Dumas M, Rosa ML, Mendling J, Reijers HA (2013) Fundamentals of Business Process Management. Springer, Nww York

[Fox11] Fox J (2011) A formal orchestration model for dynamically adaptable services with COWS. In: Proceedings International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE 2011). IARIA, pp 67–72

[GH05] Gay SJ, Hole M (2005) Subtyping for session types in the pi calculus. Acta Inf 42(2–3):191–225

[HKP+10] Hu R, Kouzapas D, Pernet O, Yoshida N, Honda K (2010) Type-safe eventful sessions in java. In: D'Hondt T (ed) ECOOP Object-Oriented Programming, 24th European Conference. Proceedings, vol. 6183 of Lecture Notes in Computer Science. Springer, Maribor, pp 329–353

[HVK98] Honda K, Vasconcelos VT, Kubo M (1998) Language primitives and type discipline for structured communication-based programming. In: Hankin C (ed) Programming Languages and Systems—ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Proceedings, vol. 1381 of Lecture Notes in Computer Science. Springer, Lisbon, pp 122–138

[HYC08] Honda K, Yoshida N, Carbone M (2008) Multiparty asynchronous session types. In: Necula GC, Wadler P (eds) POPL. ACM, pp 273–284

[JC93] Jaccheri ML, Conradi R (1993) Techniques for process model evolution in EPOS. IEEE Trans Softw Eng 19(12):1145–1156

[Kou12] Kouzapas D (2012) A Study of Bisimulation Theory for Session Types. Ph.D. thesis, Imperial College London

[KYH11] Kouzapas D, Yoshida N, Honda K (2011) On asynchronous session semantics. In: Bruni R, Dingel J (eds) Formal Techniques for Distributed Systems—Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011. Proceedings, vol. 6722 of Lecture Notes in Computer Science. Springer, Reykjavik, pp 228–243

[KYHH16] Kouzapas D, Yoshida N, Hu R, Honda K (2016) On asynchronous eventful session semantics. Mathe Struct Comput Sci 26(2):303–364

[MGR04] Müller R, Greiner U, Rahm E (2004) $A_{gent}w_{ork}$: a workflow system supporting rule-based workflow adaptation. Data Knowl Eng 51(2):223–256

[MPW92] Milner R, Parrow J, Walker D (1992) A calculus of mobile processes, i. Inf Comput 100(1):1–40

[Pad10] Padovani L (2010) Session types = intersection types + union types. In: Pimentel E, Venneri B, Wells JB (eds) Proceedings Fifth Workshop on Intersection Types and Related Systems, vol. 45 of EPTCS. ITRS, Edinburgh, pp 71–89

[PVV14] Padovani L, Vasconcelos VT, Vieira HT (2014) Typing liveness in multiparty communicating systems. In: Kühn E, Pugliese R (eds) Coordination Models and Languages—16th IFIP WG 6.1 International Conference, COORDINATION, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Proceedings, vol. 8459 of Lecture Notes in Computer Science. Springer, Berlin, pp 147–162

[RRD04] Rinderle S, Reichert M, Dadam P (2004) Correctness criteria for dynamic changes in workflow systems—a survey. Data Knowl Eng 50(1):9–34

[SC06] Seco JC, Caires L (2006) Types for dynamic reconfiguration. In: Sestoft P (ed) Programming Languages and Systems, 15th European Symposium on Programming, ESOP, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS, Proceedings, vol. 3924 of Lecture Notes in Computer Science. Springer, Vienna, pp 214–229

[VV13] Vieira HT, Vasconcelos VT (2013) Typing progress in communication-centred systems. In: De Nicola R, Julien C (eds) Coordination Models and Languages, 15th International Conference, COORDINATION, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec. Proceedings, vol. 7890 of Lecture Notes in Computer Science. Springer, Florencem, pp 236–250

Received 3 March 2015

Revised 19 March 2016

Accepted 31 March 2016 by Thomas Hildebrandt, Joachim Parrow, Matthias Weidlich, and Marco Carbone

Published online 12 May 2016