Self-adaptive multiparty sessions

Self-Adaptive Multiparty Sessions

Mario Coppo · Mariangiola Dezani-Ciancaglini · Betti Venneri

Abstract To model the notion of *self-adaptiveness* for *multiparty sessions*, we propose a formal framework, where participants can access and modify the *global state*, in such a way that the whole system can react promptly to unforeseen events by reconfiguring itself. The adaptation strategy is triggered by the overall communication choreography, represented by a global type. When the global type is dynamically updated, its projections define new *monitors*, which set-up novel communication protocols for the participants. The key result of this paper is that self-adaptations are performed in a type-safe way, while providing a high degree of flexibility. *Subject Reduction* and *Progress* properties are proven: any session executes all required communications in a type safe way and never gets stuck.

1 Introduction

The topic of self-adaptiveness emerged as a key research subject within various application domains, as a response to the growing complexity of software systems operating in many different scenarios and in highly dynamic environments. To manage this complexity at a reasonable cost, novel approaches are needed in which a system can promptly react to crucial changes by reconfiguring its behaviour autonomously and dynamically, in accordance with evolving policies and objectives.

As for a precise definition of *self-adaptivity*, this is still a debated question, due to the wide spectrum of the involved features. In our opinion, a simple, but rather deep, characterisation is the one presented in [7]: *we define adaptation as the run-time modification of the control data ...and a component is self-adaptive if it is able to modify its own control data at run-time*. We follow [7] in claiming that we need to distinguish between standard data and control data: a change in the system behaviour is part of the application logic if it is based on standard data, it is an adaptation if it is based on control data.

This paper injects the above notion of *self-adaptivity* into the formal framework of *multiparty sessions* [28], where each participant can access and modify the *global state* representing those (control) data whose values are critical for planning the adaptation steps, in such a way that the whole system can react to changes in the global data by reconfiguring itself. A system comprises four active parties: *global types*, *monitors*, *processes*, and *adaptation functions*.

A global type represents the overall communication choreography [9]; its projections onto participants generate the monitors, which are essentially local types and set-up the communication protocols of the participants. The association of a monitor with a compliant process, dubbed *monitored process*, incarnates a participant where the process provides the implementation of the monitoring protocol. Notably, we exploit intersection types, union types and subtyping to make this compliance relation flexible. Processes are able to follow different incompatible computational paths. For instance, a process could contain both the code needed to buy a book and the one needed to arrange a friend meeting, the choice between the two being determined by the monitor controlling it.

The adaptation strategy is defined by global types and adaptation functions. The choreography decides *when* the adaptation takes place, since its monitors prescribe when some participants have to check global data, and then send a request of adaptation to the other participants together with

M. Coppo · M. Dezani-Ciancaglini
Dipartimento di Informatica, Università di Torino, corso Svizzera 185, 10149 Torino, Italy {coppo,dezani}@di.unito.it

B. Venneri
Dipartimento di Statistica, Informatica, Applicazioni, Università di Firenze Viale Morgagni 65, 50134 Firenze, Italy betti.venneri@unifi.it

an adaptation function. The adaptation functions contain the dynamic evolution policy, since they prescribe *how* the system needs to reconfigure itself based on the changes of the critical data.

When an adaptation flag is thrown, new monitors are generated, according to a new choreography: indeed, the community involved in the session modifies both its set of participants and the internal communication patterns. Therefore, dynamic adaptations are essentially triggered by control data and monitors.

Most of the approaches to self-adaptive systems in the literature do not face the main challenge of including formal tools to ensure correctness of dynamic adaptations. Some approaches address this issue by providing verification techniques for testing properties of the performed adaptation (e.g., model checking in [25] and web services testbed in [36]). Differently, the focus of the present paper is on the formal properties of the proposed framework, which ensure that adaptations steps are performed in a correct way, being controlled by global types, monitors and process types. The key results are the proofs of Subject Reduction and Progress theorems: in any session, all outputs will eventually be consumed and all processes waiting for an input will eventually receive it.

Typical scenarios that can benefit from our self-adaptation framework are those characterised by the following features:
• a community, established for a common task or mission, has many distributed entities which interact with each other according to a given operational plan,
• the complex dynamic environment can present crucial events which require the community to modify its plan dynamically,
• those critical events are observed in any separate component of the system: this component can be checked by the session participants, so that the whole system can react promptly by updating itself,
• the dynamic changes need to be rather flexible: in each new phase, other participants can be introduced or some of the old participants are no longer involved (temporarily or permanently),
• these dynamic changes need to be safe: interactions must proceed correctly to pursue the common task.

Example As an example of such a scenario, let us consider a company which has various productive units and sale organisations scattered around the world. Each factory has a number of machines and produces several products for nearby markets or for export. The state of the plants is checked periodically. Communications among factories and sellers exchange several data about products and prices, according to a given combination factory-seller for each product. The company chief supervises the whole organisation. In particular, she equipped the company with an adaptation policy, which gives potential alternative plans for moving productions and/or sales of a product to different entities. All the interactions among these participants run under the control of the monitors that are originated from a global type. Finally, a global state contains crucial data, for instance the performance of machineries, plants and sale organisations. Unforeseen circumstances, such as the catastrophic event of a fire incapacitating an entire plant, can require the company organisation to update itself: new production and sale plans have to be adopted to maintain uninterrupted supply to customers.

We simplify the above scenario in the case of a Company which has two factories, iF (Italian factory) and aF (American factory), and two sellers, iS (Italian seller) and aS (American seller). In order to give a preliminary intuition of our system, we use a simplified and incomplete syntax (w.r.t. the formal presentation of next section). In Section 5 we will enrich and formalise this example.

To show how self-adaptation works, we consider the case when a fire incapacitates a factory. The global state contains either OK or KO for each of the two plants. When both plants are OK the interaction takes place according to the following global type:

$$\mathsf{G}_1 = \begin{cases} \mathsf{iS} \to \mathsf{iF} : (\mathrm{String}, \mathrm{Int}). \\ \mathsf{aS} \to \mathsf{aF} : (\mathrm{String}, \mathrm{Int}). \\ \mathsf{Ada} \to \{\mathsf{iS}, \mathsf{iF}, \mathsf{aS}, \mathsf{aF}\} : \mathit{check} \end{cases}$$

Each seller requires to the corresponding factory a certain amount (of type Int) of an item (of type String), then the chief Ada sends a checking flag to all, as an alert for a possible adaptation. When the Italian factory is OK, while the American factory is KO, the global type is:

$$\mathsf{G}_2 = \begin{cases} \mathsf{Ada} \to \mathsf{Bob} : \mathrm{String}. \\ \mathsf{iS} \to \mathsf{iF} : (\mathrm{String}, \mathrm{Int}). \\ \mathsf{aS} \to \mathsf{iF} : (\mathrm{String}, \mathrm{Int}). \\ \mathsf{Ada} \to \{\mathsf{iS}, \mathsf{iF}, \mathsf{aS}, \mathsf{Bob}\} : \mathit{check} \end{cases}$$

where Ada sends to Bob a contract (type String) for rebuilding the plant and both sellers send their requests to the Italian factory. In the symmetric case, the global type G_3 prescribes that both sellers send their requests to the American factory. Finally, when both factories are KO, Ada just closes down the business by sending the label *bye* to both sellers:

$$\mathsf{G}_4 = \mathsf{Ada} \to \{\mathsf{iS}, \mathsf{aS}\} : \mathit{bye}.\mathsf{end}$$

The processes in Table 1 are implementations of the monitors generated by projection from all the above global types. For instance, the monitor of aS from G_1 and G_3 is

$$\mathsf{aF}!(\mathrm{String}, \mathrm{Int}).\mathsf{Ada}?\mathit{check},$$

where ! represents output and ? represents input. The monitor of the American seller from G_2 is similar:

$$\mathsf{iF}!(\mathrm{String}, \mathrm{Int}).\mathsf{Ada}?\mathit{check}.$$

$$
\begin{array}{ll}
\mathit{Seller} & = \mu X.y!(\text{item},\text{amount}).y?\mathit{check}.X + y?\mathit{bye} \\
\mathit{Factory} & = \mu X.y?(x,w).\mathsf{if}\ \ldots\ \mathsf{then}\ y?\mathit{check}.X + y?(x,w).y?\mathit{check}.X \\
 & \qquad\qquad \mathsf{else\ write\ KO}.(y?\mathit{check} + y?(x,w).y?\mathit{check}) \\
\mathit{Ada} & = \mu X.y!\mathit{check}(F).X + y!\text{contract}.y!\mathit{check}(F).X + y!\mathit{bye} \\
\mathit{Bob} & = \mu X.y?(z).\mathsf{if}\ \ldots\ \mathsf{then\ write\ OK}.y?\mathit{check}\ \mathsf{else}\ y?\mathit{check}.X
\end{array}
$$

Table 1 Processes for the Company example

Its monitor from G_4 is simpler: $\mathsf{Ada}?\mathit{bye}.\mathsf{end}$. The process code for the seller has only two alternative behaviours, since processes do not mention senders and receivers. The seller can send on channel y item and amount, receive the *check* and then restart. Otherwise, he can receive *bye* and stop.

The control data can be modified by the factory, writing KO when it is incapacitated, and by Bob, writing OK when he accomplished the rebuilding task. The adaptation function F in Ada's process gives the new global type when applied to the pair $(\mathit{state}\ \mathsf{iF}, \mathit{state}\ \mathsf{aF})$, i.e.

$$
\begin{array}{ll}
F(\mathsf{OK},\mathsf{OK}) = \mathsf{G}_1 & F(\mathsf{OK},\mathsf{KO}) = \mathsf{G}_2 \\
F(\mathsf{KO},\mathsf{OK}) = \mathsf{G}_3 & F(\mathsf{KO},\mathsf{KO}) = \mathsf{G}_4
\end{array}
$$

A process can implement several different monitors also thanks to the external choice constructor. For instance, the process *Seller* can fill all the monitors that are generated by projecting the above global types onto the participants iS and aS.

Let us consider the system choreographed by G_1 with the global data $(\mathsf{OK},\mathsf{OK})$. The American factory changes its state to KO and then, when the chief checks the global data, the function F generates the adaptation step which produces the global type G_2. After this adaptation Bob is a new participant, while the American factory is out. Then the American seller, as prescribed by his monitor, sends his requests to the Italian factory. When process Bob writes OK for the American factory and the Italian factory is still OK, the global type produced by the adaptation step is again G_1. Then the American factory comes back into the scene.

The present paper is a revised and extended version of [12]. Key additions with respect to [12] are a full formalisation of the safety properties and more examples which illustrate characterising features of our framework.

Structure of the paper Sections 2 and 3 present the syntax of our calculus and its type system, respectively. The formal semantics is given in Section 4. Examples in Section 5 enlighten key technical points of our approach. Formal properties are the content of Section 6. Related works are discussed in Section 7. Section 8 concludes.

2 Syntax

Global types Following a widely common approach, the set-up of protocols starts from global types. Global types establish overall communication schemes. In our setting they also control the reconfiguration phase, in which a system adapts itself to new environmental conditions.

Let L be a set of *labels*, ranged over by ℓ, which mark the exchanged values as in [19] and Λ be a set of *flags*, ranged over by λ, which transmit the adaptation information. We assume to have some basic *sorts*, ranged over by S, i.e.

$$S ::= \mathsf{Bool} \mid \mathsf{Int} \mid \ldots$$

Definition 1 *Global types* are defined by:

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

In writing $\{\ell_i(S_i).\mathsf{G}_i\}_{i\in I}$ and $\{\lambda_i\}_{i\in I}$ we implicitly assume that $\ell_i \neq \ell_j$ and $\lambda_i \neq \lambda_j$ for all $i \neq j$. There are only two kinds of communications: value exchange and adaptation flag exchange. Each value exchange is characterised by a label which allows to represent choices. The sender is p, while Π is the set of the receivers, which does not contain p and cannot be the empty set. The participants of a global type G are all the senders and the receivers in G, ranged over by $\mathsf{p},\mathsf{q},\ldots$. We denote by $\mathsf{pa}(\mathsf{G})$ the set of all participants in G.

Global types can terminate in two ways: either with the usual end or with the exchange of adaptation flags. In the latter case the adaptation flags are sent by a participant to all the other ones. Adaptation flags can be seen as synchronisation points, interleaved in a conversation, at which different interaction paths can be taken. In the global types syntax there is no recursion operator, but recursive protocols can be obtained by reconfiguring the system with the same global type. Recursion can then be considered as a particular case of reconfiguration. A recursion operator is included instead in the syntax of processes. For instance, all processes in the company example of the Introduction are recursive and they implement monitors which are projections of non-recursive global types.

Notably, we do not allow parallel composition of global types, which is quite common in the literature [28,9,3,10]. As a matter of fact many papers [28,9,3] require that two global types can be put in parallel only if their sets of participants are disjoint, so parallel composition can be expressed by interleaving. Without this condition parallel composition of global types requires some care [10]. This issue is orthogonal to the present framework, where each participant, in all reconfiguration steps, follows one global type only (see Tables 8 and 9).

Monitors Monitors can be viewed as local types that are obtained as projections of global types onto individual participants, as in the standard approach of [28] and [2]. The only syntactic differences are the presence of the adaptation flags and the absence of recursion and delegation. In our calculus, however, monitors are more than types: they have an active role in system dynamics, since they guide communications and adaptations.

$$(\mathsf{p} \to \Pi : \{\ell_i(S_i).\mathsf{G}_i\}_{i\in I}) \upharpoonright \mathsf{q} =$$
$$\begin{cases} \mathsf{p}?\{\ell_i(S_i).\mathsf{G}_i \upharpoonright \mathsf{q}\}_{i\in I} & \text{if } \mathsf{q} \in \Pi \\ \Pi!\{\ell_i(S_i).\mathsf{G}_i \upharpoonright \mathsf{q}\}_{i\in I} & \text{if } \mathsf{q} = \mathsf{p} \\ \mathsf{G}_{i_0} \upharpoonright \mathsf{q} & \text{where } i_0 \in I \text{ if } \mathsf{q} \neq \mathsf{p} \text{ and } \mathsf{q} \notin \Pi \\ & \text{and } \mathsf{G}_i \upharpoonright \mathsf{q} = \mathsf{G}_j \upharpoonright \mathsf{q} \text{ for all } i, j \in I \end{cases}$$

$$(\mathsf{p} \to \Pi : \{\lambda_i\}_{i\in I}) \upharpoonright \mathsf{q} = \begin{cases} \mathsf{p}?\{\lambda_i\}_{i\in I} & \text{if } \mathsf{q} \in \Pi \\ \Pi!\{\lambda_i\}_{i\in I} & \text{if } \mathsf{q} = \mathsf{p} \\ \mathsf{end} & \text{if } \mathsf{q} \neq \mathsf{p} \text{ and } \mathsf{q} \notin \Pi \end{cases}$$
$$\mathsf{end} \upharpoonright \mathsf{p} = \mathsf{end}$$

Table 2 Projection of a global type onto a participant

Definition 2 The set of *monitors* is defined by:

$$\begin{array}{lllll} \mathscr{M} & ::= & \mathsf{p}?\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I} & \mid & \Pi!\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I} \mid \\ & & \mathsf{p}?\{\lambda_i\}_{i\in I} & \mid & \Pi!\{\lambda_i\}_{i\in I} \mid \\ & & \mathsf{end} & & \end{array}$$

The constructs in the first line correspond to input and output actions, respectively. An input monitor $\mathsf{p}?\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I}$ fits with a process that can receive, for each $i \in I$, a value of sort S_i, labeled by ℓ_i, having as continuation a process which agrees with $\mathscr{M}_i$. This corresponds to an external choice. Dually an output monitor $\Pi!\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I}$ fits with a process which can send (by an internal choice) a value of sort S_i, distinguished by the label ℓ_i for each $i \in I$, and then continues as prescribed by $\mathscr{M}_i$.

The projection of global types onto participants is given in Table 2. A projection is undefined when two participants not involved, as sender or receiver, in a choice have different projections in different branchings (condition $\mathsf{G}_i \upharpoonright \mathsf{q} = \mathsf{G}_j \upharpoonright \mathsf{q}$ for all $i, j \in I$). Monitors are the results of such projections.

A global type G is *well formed* if its projections are defined for all participants and all occurrences of

$$\mathsf{p} \to \Pi : \{\lambda_i\}_{i\in I}$$

are such that $\Pi \cup \{\mathsf{p}\} = \mathsf{pa}(\mathsf{G})$: i.e. all participants are involved in each flag exchange. In the following we assume that all global types are well formed.

Processes Processes represent code that is associated to monitors in order to implement participants.

Differently from session calculi [27,23,24,28,2,33,19, 11,34,3], processes do not specify the participants involved in sending and receiving actions. The associated monitors determine senders and receivers. Processes represent flexible code that can be associated to different monitors to incarnate different participants. Besides communicating, processes can access the global state to read or change it.

The communication actions of processes are performed through *channels*. Each process owns a unique channel. We use y to denote this channel in the user code. As usual, the user channel y will be replaced at run time by a session channel $\mathsf{s}[\mathsf{p}]$ (where s is the session name and p is the current participant). Let c denote a user channel or a session channel. We could avoid to write explicitly the channel in the user syntax, but not in the run-time syntax. We have chosen to write all channels to simplify the definition of processes.

Definition 3 *Processes* are defined by:

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

The syntax of processes is rather standard, in particular the operator $+$ represents external choice. Notice that internal and external choices with many branches can easily be encoded in our calculus, which gains in simplicity. In writing processes we assume the precedence: prefix, external choice, recursion. Note that in the sending and receiving actions the involved participants are missing. For instance, $\mathsf{c}!\ell(\mathsf{e}).P$ denotes a process which sends via the channel c the label ℓ and the value of the expression e and then has P as continuation. Notably, a system has a global state (see Definition 5) and the op operator represents an action on this global state, for instance a "read" or "write" operation. We leave unspecified the kind of actions since we are only interested in the dynamic changes of this state, which plays the role of the control data for the self-reconfiguration of the whole system.

Types, which are statically assigned to processes, will be formally introduced in Section 3 (Definition 7). Types are mainly aimed at checking the matching between processes and monitors. It is convenient to include a type annotation in the syntax of the adaptation flag. The input flag $\mathsf{c}?(\lambda,\mathsf{T}).P$ represents a process that, after receiving the adaptation flag λ, has a continuation of type T. Thus the explicit annotation T makes it easy to dynamically check if, after the adaptation, the current process can continue with that type, inside the new monitor. The output flag $\mathsf{c}!(\lambda(F),\mathsf{T}).P$ contains also the *adaptation function* F. The application of F to the global state will determine the new global type, which provides a new choreography for the system.

Network A process is always controlled by a monitor, which ensures that all performed actions fit the protocol prescribed by the global type. Each monitor controls a single process. So participants correspond to pairs of processes and monitors. We write $\mathscr{M}[P]$ to represent a process P controlled by a monitor $\mathscr{M}$, dubbed *monitored process*. In a reconfiguration phase the monitor controlling the process is changed according to the new global type resulting from the application of the adaptation function to the global state. At this

point the processes whose type does not fit the new monitor must leave the system and new ones can enter it. The data exchange among the participants is done by means of runtime queues (one for each active session). We denote by $\mathsf{s}:h$ the *named queue* associated with the session s, where h is a *message queue*. The empty queue is denoted by ø. Messages in queues can be either value messages $(\mathsf{p},\Pi,\ell(v))$, indicating that the label ℓ and the value v are sent by participant p to all participants in Π, or adaptation messages $(\mathsf{p},\Pi,\lambda(\mathsf{G}))$, indicating that the flag λ and the global type G are sent by participant p to all participants in Π. Queue concatenation, denoted by "·", has ø as neutral element. A queue is λ-*free* if it contains no flag.

The sessions are initiated by the "new" constructor applied to a global type (*session initiator*), denoted by $\mathsf{new}(\mathsf{G})$, which generates the monitors and associates them with adequate processes (see Definition 8).

The parallel composition of session initiators, processes with the corresponding monitors and runtime queues form a network. Networks can be restricted on session names.

Definition 4 *Networks* are defined by:

$$N ::= \mathsf{new}(\mathsf{G}) \mid \mathscr{M}[P] \mid \mathsf{s}:h \mid N \mid N \mid (\nu\mathsf{s})N$$

System A system includes a network, a global state and assumes a collection of processes together with their types (according to the typing rules of Section 3). We use σ to range over global states and we denote by $\mathscr{P}$ the collection of pairs (P,T). We represent systems as the composition (via "$\|$") of a networks and a global state, without mentioning the process collection, which is considered an implicit parameter.

Definition 5 *Systems* are defined by:

$$\mathscr{S} ::= N \parallel \sigma$$

3 Process types

Process types (called simply *types* where not ambiguous) describe process communication behaviours [27]. They have prefixes corresponding to sending and receiving of labels and flags. In particular an *input type* is a type whose first prefix corresponds to an input action and an *output type* is a type whose first prefix corresponds to an output action, while the *continuation* of a type is the type following its first prefix. A *communication type* is either an input or an output type. The external choice is typed by an intersection type, since an external choice offers both behaviours of the composing processes. Dually, a conditional is an internal choice and so it is typed by a union type. Notice that union and intersection being binary constructor feet well with our binary internal and external choices.

$$\begin{gathered}
lin(?\ell(S).T) = lout(!\ell(S).T) = \{\ell\} \\
lin(?\lambda) = lout(!\lambda) = \{\lambda\} \\
lin(!\ell(S).T) = lin(!\lambda) = lout(?\ell(S).T) = lout(?\lambda) = \emptyset \\
lin(T_1 \wedge T_2) = lin(T_1 \vee T_2) = lin(T_1) \cup lin(T_2) \\
lout(T_1 \wedge T_2) = lout(T_1 \vee T_2) = lout(T_1) \cup lout(T_2)
\end{gathered}$$

Table 3 The mappings *lin* and *lout*

To formally define types, we start with the more liberal syntax of pre-types and then we define some restrictions that characterise types of processes.

Definition 6 The set of *pre-types* is inductively defined by:

$$T ::= ?\ell(S).T \mid !\ell(S).T \mid ?\lambda \mid !\lambda \mid T \wedge T \mid T \vee T \mid \mathsf{end}$$

where $\wedge$ and $\vee$ are considered modulo idempotence, commutativity and associativity.

In pre-types and types we assume that . has precedence over $\wedge$ and $\vee$.

In order to define types for processes, we have to avoid intersection between input types with the same first label, which would represent ambiguous external choices: indeed, the types following a same input prefix could be different and this would lead to a communication mismatch, as illustrated in Example 1 of Section 5. For the same reason, process types cannot contain intersections between output types with the same label. Since we have to match types with monitors, where internal choices are always taken by participants sending a label or a flag, we force unions to take output types (possibly combined by intersections or unions) as arguments. Therefore, we formalise the above restrictions by means of two mappings from pre-types to sets of labels and flags (Table 3) and then we define types by using those mappings.

Definition 7 A *type* is a pre-type satisfying the following constraints modulo idempotence, commutativity and associativity of unions and intersections:

- all occurrences of the shape $T_1 \wedge T_2$ are such that
$$lin(T_1) \cap lin(T_2) = lout(T_1) \cap lout(T_2) = \emptyset$$
- all occurrences of the shape $T_1 \vee T_2$ are such that
$$lin(T_1) = lin(T_2) = lout(T_1) \cap lout(T_2) = \emptyset.$$

We use T to range over types and $\mathscr{T}$ to denote the set of types. Note that, for example, $(T \wedge T) \vee T$ is a type, whenever T is a type, since types are considered modulo idempotence.

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

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

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

We assume that expressions are typed by sorts, as usual. The typing judgments for expressions are of the shape

$$\Gamma \vdash \mathsf{e} : S$$

and the typing rules for expressions are standard.

Only processes with at most one channel can be typed. This choice is justified by the design of monitored processes as session participants and by the absence of delegation. Therefore the typing judgments for processes have the form

$$\Gamma \vdash P \triangleright \mathsf{c} : \mathsf{T}.$$

Typing rules for processes are given in Table 4. Observe that the type of a process after a reconfiguration is memorised in the (input or output) action in which the adaptation flag is exchanged (see Definition 3). In rules IF and CHOICE we require that the applications of union and intersection on two types form a type (conditions $\mathsf{T}_1 \vee \mathsf{T}_2 \in \mathscr{T}$ and $\mathsf{T}_1 \wedge \mathsf{T}_2 \in \mathscr{T}$). Adaptation allows us to avoid recursive types. A recursion variable is always preceded by an adaptation action, i.e. $\mathsf{c}?(\lambda, \mathsf{T}).X$ (rule RV1) and $\mathsf{c}!(\lambda(F), \mathsf{T}).X$ (rule RV2). In typing a recursive process $\mu X.P$, rule REC ensures that the type of P is the same as the type associated to X in the environment. Note that $\mu X.P$ is equivalent to $P\{\mu X.P/X\}$ and so, unfolding the process, P will always be associated to all the reconfiguration flags which precede the occurrences of X.

For example, writing the process *Ada* (considered in the Introduction) using the formal syntax, but leaving out labels, $y!(check(F), \mathsf{T}_{Ada})$ replaces $y!check(F)$, where the type of the whole process *Ada* is:

$$\mathsf{T}_{Ada} = !check \wedge !\mathsf{String}.!check \wedge !bye.\mathsf{end}$$

The matching between process types and monitors (adequacy) is made rather flexible by using the *subtype* relation on types defined in Table 5. Subtyping is monotone, for input/output prefixes, with respect to continuations and it follows the usual set theoretic inclusion of intersection and union. Notice that we use a weaker definition than standard subtyping on intersection and union types, since it is sufficient to define subtyping on types.

The intuitive meaning of subtyping is that a process with a smaller type has all the behaviours required by a bigger type and possibly more. Therefore end is the top type. An input monitor naturally corresponds to an external choice, while an output monitor naturally corresponds to an internal choice. So intersections of input types are adequate for input monitors and unions of output types are adequate for output monitors. Formally, we say that a type is adequate for a monitor if the conditions of the following definition hold.

$\leq$ is the minimal reflexive and transitive relation on $\mathscr{T}$ such that:

$$\begin{gathered}
\mathsf{T} \leq \mathsf{end} \quad \mathsf{T}_1 \wedge \mathsf{T}_2 \leq \mathsf{T}_i \quad \mathsf{T}_i \leq \mathsf{T}_1 \vee \mathsf{T}_2 \quad (i = 1,2) \\
\mathsf{T}_1 \leq \mathsf{T}_2 \text{ implies } !\ell(S).\mathsf{T}_1 \leq !\ell(S).\mathsf{T}_2 \\
\mathsf{T}_1 \leq \mathsf{T}_2 \text{ implies } ?\ell(S).\mathsf{T}_1 \leq ?\ell(S).\mathsf{T}_2 \\
\mathsf{T} \leq \mathsf{T}_1 \text{ and } \mathsf{T} \leq \mathsf{T}_2 \text{ imply } \mathsf{T} \leq \mathsf{T}_1 \wedge \mathsf{T}_2 \\
\mathsf{T}_1 \leq \mathsf{T} \text{ and } \mathsf{T}_2 \leq \mathsf{T} \text{ imply } \mathsf{T}_1 \vee \mathsf{T}_2 \leq \mathsf{T} \\
(\mathsf{T}_1 \vee \mathsf{T}_2) \wedge \mathsf{T}_3 \leq \mathsf{T} \text{ iff } \mathsf{T}_1 \wedge \mathsf{T}_3 \leq \mathsf{T} \text{ and } \mathsf{T}_2 \wedge \mathsf{T}_3 \leq \mathsf{T} \\
\mathsf{T} \leq (\mathsf{T}_1 \wedge \mathsf{T}_2) \vee \mathsf{T}_3 \text{ iff } \mathsf{T} \leq \mathsf{T}_1 \vee \mathsf{T}_3 \text{ and } \mathsf{T} \leq \mathsf{T}_2 \vee \mathsf{T}_3
\end{gathered}$$

Table 5 Subtyping

Definition 8 A type T is *adequate* for a monitor $\mathscr{M}$ (notation $\mathsf{T} \propto \mathscr{M}$) if $\mathsf{T} \leq |\mathscr{M}|$, where the mapping $|\ |$ is defined by:

$$|\mathsf{p}?\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I}| = \bigwedge_{i\in I} ?\ell_i(S_i).|\mathscr{M}_i|$$

$$|\Pi!\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I}| = \bigvee_{i\in I} !\ell_i(S_i).|\mathscr{M}_i|$$

$$|\mathsf{p}?\{\lambda_i\}_{i\in I}| = \bigwedge_{i\in I} ?\lambda_i \quad |\Pi!\{\lambda_i\}_{i\in I}| = \bigvee_{i\in I} !\lambda_i \quad |\mathsf{end}| = \mathsf{end}$$

For instance, the type $\mathsf{T}_{\mathsf{Ada}}$ defined above is adequate for the monitor of Ada obtained by projecting the global type G_2 discussed in the Introduction:

$$\mathsf{Bob}!\mathsf{String}.\{\mathsf{iS}, \mathsf{iF}, \mathsf{aS}, \mathsf{Bob}\}!check$$

Decidability of adequacy relies on decidability of subtyping. We show that subtyping is decidable. The proof exploits standard distributivity properties on intersections and unions.

Lemma 1 *Subtyping is decidable.*

Proof A subtyping between two types is equivalent to a set of subtypings, in which no union occurs in the left type and no intersection occurs in the right type. In fact:

- $\mathsf{T}_1 \vee \mathsf{T}_2 \leq \mathsf{T}_3$ iff $\mathsf{T}_1 \leq \mathsf{T}_3$ and $\mathsf{T}_2 \leq \mathsf{T}_3$
- $(\mathsf{T}_1 \vee \mathsf{T}_2) \wedge \mathsf{T}_3 \leq \mathsf{T}_4$ iff $\mathsf{T}_1 \wedge \mathsf{T}_3 \leq \mathsf{T}_4$ and $\mathsf{T}_2 \wedge \mathsf{T}_3 \leq \mathsf{T}_4$.

Notice that if $(\mathsf{T}_1 \vee \mathsf{T}_2) \wedge \mathsf{T}_3$ is a type, then both $\mathsf{T}_1 \wedge \mathsf{T}_3$ and $\mathsf{T}_2 \wedge \mathsf{T}_3$ are types. A similar argument can be used for erasing the intersections in the right type. Then we have to decide only subtypings of the shape $\mathsf{T} \leq \mathsf{T}'$, where T is an intersection of communication types and possibly end, while T' is either a union of output types or a single input type, both possibly in union with end (by Definition 7). Since end is the top type:
- If T' contains end, then the subtyping holds.
- If T is end, then the subtyping fails, unless T' contains end.
Otherwise, the occurrences of end in T can be erased.
Thus we are reduced to consider the cases in which T is an intersection of communication types and T' is either a union of output types or a single input type. In the first case,

$$\Gamma \vdash \mathbf{0} \triangleright \mathsf{c} : \mathsf{end}\ \text{END} \qquad \frac{\Gamma \vdash P \triangleright \mathsf{c} : \mathsf{T}}{\Gamma \vdash \mathsf{op}.P \triangleright \mathsf{c} : \mathsf{T}}\ \text{OP} \qquad \Gamma, X : \mathsf{T} \vdash \mathsf{c}?(\lambda, \mathsf{T}).X \triangleright \mathsf{c} :?\lambda\ \text{RV1} \qquad \Gamma, X : \mathsf{T} \vdash \mathsf{c}!(\lambda(F), \mathsf{T}).X \triangleright \mathsf{c} :!\lambda\ \text{RV2}$$

$$\frac{\Gamma, X : \mathsf{T} \vdash P \triangleright \mathsf{c} : \mathsf{T}}{\Gamma \vdash \mu X.P \triangleright \mathsf{c} : \mathsf{T}}\ \text{REC} \qquad \frac{\Gamma, x : S \vdash P \triangleright \mathsf{c} : \mathsf{T}}{\Gamma \vdash \mathsf{c}?\ell(x).P \triangleright \mathsf{c} :?\ell(S).\mathsf{T}}\ \text{RCV} \qquad \frac{\Gamma \vdash P \triangleright \mathsf{c} : \mathsf{T} \qquad \Gamma \vdash \mathsf{e} : S}{\Gamma \vdash \mathsf{c}!\ell(\mathsf{e}).P \triangleright \mathsf{c} :!\ell(S).\mathsf{T}}\ \text{SEND}$$

$$\frac{\Gamma \vdash P \triangleright \mathsf{c} : \mathsf{T}}{\Gamma \vdash \mathsf{c}?(\lambda, \mathsf{T}).P \triangleright \mathsf{c} :?\lambda}\ \text{FRCV} \qquad \frac{\Gamma \vdash P \triangleright \mathsf{c} : \mathsf{T}}{\Gamma \vdash \mathsf{c}!(\lambda(F), \mathsf{T}).P \triangleright \mathsf{c} :!\lambda}\ \text{FSEND}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{Bool} \quad \Gamma \vdash P_1 \triangleright \mathsf{c} : \mathsf{T}_1 \quad \Gamma \vdash P_2 \triangleright \mathsf{c} : \mathsf{T}_2 \quad \mathsf{T}_1 \vee \mathsf{T}_2 \in \mathscr{T}}{\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \triangleright \mathsf{c} : \mathsf{T}_1 \vee \mathsf{T}_2}\ \text{IF}$$

$$\frac{\Gamma \vdash P_1 \triangleright \mathsf{c} : \mathsf{T}_1 \quad \Gamma \vdash P_2 \triangleright \mathsf{c} : \mathsf{T}_2 \quad \mathsf{T}_1 \wedge \mathsf{T}_2 \in \mathscr{T}}{\Gamma \vdash P_1 + P_2 \triangleright \mathsf{c} : \mathsf{T}_1 \wedge \mathsf{T}_2}\ \text{CHOICE}$$

Table 4 Typing rules for processes

$$\mathsf{op}.P \xrightarrow{\mathsf{op}} P \qquad \mu X.P \xrightarrow{\tau} P\{\mu X.P/X\} \qquad \mathsf{s}[\mathsf{p}]?\ell(x).P \xrightarrow{\mathsf{s}[\mathsf{p}]?\ell(v)} P\{v/x\} \qquad \mathsf{s}[\mathsf{p}]!\ell(\mathsf{e}).P \xrightarrow{\mathsf{s}[\mathsf{p}]!\ell(v)} P\ \ \mathsf{e} \downarrow v$$

$$\mathsf{s}[\mathsf{p}]?(\lambda, \mathsf{T}).P \xrightarrow{\mathsf{s}[\mathsf{p}]?(\lambda,\mathsf{T})} P \qquad \mathsf{s}[\mathsf{p}]!(\lambda(F), \mathsf{T}).P \xrightarrow{\mathsf{s}[\mathsf{p}]!(\lambda(F),\mathsf{T})} P$$

$$\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} P\ \ \mathsf{e} \downarrow \mathsf{true} \qquad \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{\tau} Q\ \ \mathsf{e} \downarrow \mathsf{false}$$

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\delta} P'}{P + Q \xrightarrow{\delta} P' + Q}$$

Table 6 LTS of processes

subtyping $\mathsf{T} \leq \mathsf{T}'$ holds if and only if at least one of the output prefixes of types in T' is equal to the output prefix of a type in T, and the corresponding continuations are in the subtype relation. In the second case, subtyping $\mathsf{T} \leq \mathsf{T}'$ holds if and only if T contains a type which has the same input prefix of T' and the corresponding continuations are in the subtype relation. □

4 Semantics

The evolution of a system depends on the evolution of its network and global state. The basic components of networks are the openings of sessions (through the new operator on global types) and the processes associated with monitors. So we start by describing how processes can evolve inside monitors. Monitors guide the communications of processes by choosing the senders/receivers and by allowing only some actions among those offered by the processes. This is formalised by the following LTS for monitors:

$$\mathsf{p}?\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I} \xrightarrow{\mathsf{p}?\ell_j} \mathscr{M}_j \qquad j \in I$$

$$\Pi!\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I} \xrightarrow{\Pi!\ell_j} \mathscr{M}_j \qquad j \in I$$

$$\mathsf{p}?\{\lambda_i\}_{i\in I} \xrightarrow{\mathsf{p}?\lambda_j} \qquad \Pi!\{\lambda_i\}_{i\in I} \xrightarrow{\Pi!\lambda_j} \qquad j \in I$$

Processes can communicate labels and values, flags, adaptation functions and types, or can read/modify the global state trough op operations. These behaviours are made explicit by the LTS in Table 6, where the treatment of recursions and conditionals is standard. In the rules for external choice, α ranges over

$$\mathsf{s}[\mathsf{p}]?\ell(v), \mathsf{s}[\mathsf{p}]!\ell(v), \mathsf{s}[\mathsf{p}]?(\lambda, \mathsf{T}), \mathsf{s}[\mathsf{p}]!(\lambda(F), \mathsf{T}),$$

and δ ranges over $\{\mathsf{op}, \tau\}$. We omit the symmetric rules.

$$h \cdot (\mathsf{q}, \Pi, \zeta) \cdot (\mathsf{q}', \Pi', \zeta') \cdot h' \equiv h \cdot (\mathsf{q}', \Pi', \zeta') \cdot (\mathsf{q}, \Pi, \zeta) \cdot h' \quad \text{if } \Pi \cap \Pi' = \emptyset \text{ or } \mathsf{q} \neq \mathsf{q}'$$

$$h \cdot (\mathsf{q}, \Pi, \zeta) \cdot h' \equiv h \cdot (\mathsf{q}, \Pi', \zeta) \cdot (\mathsf{q}, \Pi'', \zeta) \cdot h' \quad \text{if } \Pi = \Pi' \cup \Pi'' \text{ and } \Pi' \cap \Pi'' = \emptyset$$

where $\zeta ::= \ell(v) \mid \lambda(G)$.

Table 7 Equivalence on message queues

The choices are done by the communication actions, while the operations on the global state are transparent. This is needed since the operations on the memory are recorded neither in the process types nor in the monitors. An operation on the state in an external choice can be performed also if a branch, different from that containing the operation, is executed. The rationale is the independence of the changes in the control data from the communications among session participants. For example,

$$\mathsf{op.s[p]}?\ell_1(x).P_1 + \mathsf{s[p]}?\ell_2(x).P_2 \xrightarrow{\mathsf{op}}$$
$$\mathsf{s[p]}?\ell_1(x).P_1 + \mathsf{s[p]}?\ell_2(x).P_2 \xrightarrow{\mathsf{s[p]}?\ell_2(v)} P_2\{v/x\}.$$

A recursion in an external choice can be unfolded independently front the chosen branch, since unfolding is an internal computation. Similarly, a conditional in an external choice can be evaluated also if a different branch is then executed. For example,

$$(\mu X.\mathsf{if\ true\ then\ s[p]}?\ell_1(x).P_1 \mathsf{\ else\ } X) + \mathsf{s[p]}?\ell_2(x).P_2 \xrightarrow{\tau}$$
$$(\mathsf{if\ true\ then\ s[p]}?\ell_1(x).P_1 \mathsf{\ else\ } \mu X.\cdots) + \mathsf{s[p]}?\ell_2(x).P_2 \xrightarrow{\tau}$$
$$\mathsf{s[p]}?\ell_1(x).P_1 + \mathsf{s[p]}?\ell_2(x).P_2 \xrightarrow{\mathsf{s[p]}?\ell_2(v)} P_2\{v/x\}.$$

We assume a standard structural equivalence on networks, in which the parallel operator is commutative and associative, and restrictions commute and enlarge their scopes without name clashes. Moreover, any monitored process with end monitor behaves as the neutral element for the parallel and absorbs restrictions, that is:

$$\mathsf{end}[P] \mid N \equiv N \quad \text{and} \quad (\nu \mathsf{s})\mathsf{end}[P] \equiv \mathsf{end}[P].$$

For message queues, we need an equivalence for commuting independent messages and another one for splitting a message to multiple receivers, see Table 7. The equivalence on message queues induces an equivalence on labelled queues in the obvious way:

$$h \equiv h' \ \text{ implies } \ \mathsf{s} : h \equiv \mathsf{s} : h'.$$

We can distinguish between the transitions which do or do not involve the global state. For simplicity, Table 8 lists the reduction rules of the networks and Table 9 lists the reduction rules of the systems, in which all rules need the global state.

A session starts by reducing a network new G (rule INIT). For each p in the set $\mathsf{pa}(\mathsf{G})$ of the participants in the global type G, we need to find a process P_p in the collection $\mathscr{P}$ associated to the current system. The process P_p must be such that its type T_p is adequate for the monitor which is the projection of G onto p. Then the process (where the channel y has been replaced by $s[\mathsf{p}]$) is associated to the corresponding monitor and the empty queue s is created. Lastly, the name s is restricted. In this way we ensure the privacy of the communications in a session (as standard in session calculi [28]). We are interested here in modelling the overall adaptation strategy, based on decoupling interfaces (i.e. monitors) and implementations (i.e. processes), rather than in the details related to the choice of processes associated to monitors. So we have left this choice arbitrary, the only condition being type adequacy. Note, however, that a natural way of controlling the processes associated to the monitors is given by the choice of the labels and flags which relate them.

The rules IN and OUT define the exchange of messages through queues. The type assignment system ensures that both the type of P is adequate for $\mathscr{M}$ and the type of P' is adequate for $\mathscr{M}'$. Following [11,3], the agreement between monitors and processes is required; a novelty is that only the monitors define the senders and the receivers of messages.

The rules ADAINCONT and ADAINNEW of Table 8 deal with adaptations, for the session participants which receive the adaptation flag with the new global type. The new global type is needed to compute the new monitor $\mathscr{M}'$ by projection. In the first rule the continuation of the current process inside the monitor has a type which is adequate for $\mathscr{M}'$, so this process will fill $\mathscr{M}'$. In the second rule, instead, it is needed to take from a different process with a type adequate for $\mathscr{M}'$ the collection $\mathscr{P}$. In case such a process does not exists in $\mathscr{P}$, then the system gets stuck.

Evaluation contexts are defined by

$$\mathscr{E} ::= [\,] \mid \mathscr{E} \mid N \mid (\nu s)\mathscr{E}$$

The reduction rules for networks can be used for reducing systems thanks to rule SN in Table 9. Rule CXT is a standard contextual rule. Rule OP allows processes to read-/modify the global state.

The most interesting rules are ADAOUTCONT and ADAOUTNEW. In both rules participant p sends an adaptation flag and an adaptation function, whose application to the global state gives a new global type G. The global type G may involve new participants (in $\Pi' \setminus (\Pi \cup \{\mathsf{p}\})$) which are added to the network by taking processes in $\mathscr{P}$ as in rule INIT. As regards to participant p, the new monitor $\mathsf{G}\upharpoonright\mathsf{p}$ will be associated or not to the current process according to whether its type is adequate or not for $\mathsf{G}\upharpoonright\mathsf{p}$, as in rules ADAINCONT and ADAINNEW. In both rules ADAOUTCON and ADAOUTNEW the message with the reconfiguration flag and the global type G will be sent to all participants of the session before the reconfiguration. This is ensured by the well-formedness of global types.

The restriction to λ-free queues deserves some comments. It ensures no new adaptation flag can be thrown until all the receivers of the previous adaptation flag have adapted themselves. A design choice of our framework is to allow a participant to skip an adaptation phase (since it does not appear in the new global type) and then to appear again in the following adaptation. This models a common scenario in which a component is temporarily unavailable and so a new choreography is needed. In the introductory example, the Amer-

$$\frac{\Pi = \mathsf{pa}(\mathsf{G}) \quad \mathscr{M}_\mathsf{p} = \mathsf{G}\upharpoonright\mathsf{p} \quad \forall \mathsf{p} \in \Pi.\ (P_\mathsf{p}, \mathsf{T}_\mathsf{p}) \in \mathscr{P}\ \&\ \mathsf{T}_\mathsf{p} \propto \mathscr{M}_\mathsf{p}}{\mathsf{new}(\mathsf{G}) \longrightarrow (\nu\,\mathsf{s})\,(\prod_{\mathsf{p}\in\Pi} \mathscr{M}_\mathsf{p}[P_\mathsf{p}\{\mathsf{s}[\mathsf{p}]/y\}] \mid \mathsf{s} : \emptyset)}\ \textsc{Init} \qquad \frac{P \xrightarrow{\tau} P'}{\mathscr{M}[P] \longrightarrow \mathscr{M}[P']}\ \textsc{Tau}$$

$$\frac{\mathscr{M} \xrightarrow{\mathsf{q}?\ell} \mathscr{M}' \quad P \xrightarrow{\mathsf{s}[\mathsf{p}]?\ell(v)} P'}{\mathscr{M}[P] \mid \mathsf{s} : (\mathsf{q},\mathsf{p},\ell(v))\cdot h \longrightarrow \mathscr{M}'[P'] \mid \mathsf{s} : h}\ \textsc{In} \qquad \frac{\mathscr{M} \xrightarrow{\Pi!\ell} \mathscr{M}' \quad P \xrightarrow{\mathsf{s}[\mathsf{p}]!\ell(v)} P'}{\mathscr{M}[P] \mid \mathsf{s} : h \longrightarrow \mathscr{M}'[P'] \mid \mathsf{s} : h\cdot(\mathsf{p},\Pi,\ell(v))}\ \textsc{Out}$$

$$\frac{\mathscr{M} \xrightarrow{\mathsf{q}?\lambda} \quad P \xrightarrow{\mathsf{s}[\mathsf{p}]?(\lambda,\mathsf{T})} P' \ \ \mathsf{G}\upharpoonright\mathsf{p} = \mathscr{M}' \quad \mathsf{T} \propto \mathscr{M}'}{\mathscr{M}[P] \mid \mathsf{s} : (\mathsf{q},\mathsf{p},\lambda(\mathsf{G}))\cdot h \longrightarrow \mathscr{M}'[P'] \mid \mathsf{s} : h}\ \textsc{AdaInCont}$$

$$\frac{\mathscr{M} \xrightarrow{\mathsf{q}?\lambda} \quad P \xrightarrow{\mathsf{s}[\mathsf{p}]?(\lambda,\mathsf{T})} P' \ \ \mathsf{G}\upharpoonright\mathsf{p} = \mathscr{M}' \quad \mathsf{T} \not\propto \mathscr{M}' \quad (Q,\mathsf{T}') \in \mathscr{P} \quad \mathsf{T}' \propto \mathscr{M}'}{\mathscr{M}[P] \mid \mathsf{s} : (\mathsf{q},\mathsf{p},\lambda(\mathsf{G}))\cdot h \longrightarrow \mathscr{M}'[Q\{\mathsf{s}[\mathsf{p}]/y\}] \mid \mathsf{s} : h}\ \textsc{AdaInNew}$$

$$\frac{N_1 \equiv N_1' \quad N_1' \longrightarrow N_2' \quad N_2 \equiv N_2'}{N_1 \longrightarrow N_2}\ \textsc{Equiv}$$

Table 8 Network reduction

$$\frac{P \xrightarrow{\mathsf{op}} P'}{\mathscr{M}[P] \parallel \sigma \longrightarrow \mathscr{M}[P'] \parallel \mathsf{op}(\sigma)}\ \text{OP}$$

$$\frac{\begin{array}{c}\mathscr{M} \xrightarrow{\Pi!\lambda} \quad P \xrightarrow{\mathsf{s}[\mathsf{p}]!(\lambda(F),\mathsf{T})} P' \quad F(\sigma) = \mathsf{G} \quad \mathscr{M}_\mathsf{p} = \mathsf{G}\upharpoonright\mathsf{p} \quad \mathsf{T} \propto \mathscr{M}_\mathsf{p} \quad h\ \lambda\text{-free} \\ \Pi' = \mathsf{pa}(\mathsf{G}) \quad \forall \mathsf{q} \in \Pi'.\mathscr{M}_\mathsf{q} = \mathsf{G}\upharpoonright\mathsf{q} \quad \forall \mathsf{q} \in \Pi' \setminus (\Pi \cup \{\mathsf{p}\}).\ (P_\mathsf{q},\mathsf{T}_\mathsf{q}) \in \mathscr{P}\ \&\ \mathsf{T}_\mathsf{q} \propto \mathscr{M}_\mathsf{q}\end{array}}{\mathscr{M}[P] \mid \mathsf{s} : h \parallel \sigma \longrightarrow \mathscr{M}_\mathsf{p}[P'] \mid \prod_{\mathsf{q}\in\Pi'\setminus(\Pi\cup\{\mathsf{p}\})} \mathscr{M}_\mathsf{q}[P_\mathsf{q}\{\mathsf{s}[\mathsf{q}]/y_\mathsf{q}\}] \mid \mathsf{s} : h\cdot(\mathsf{p},\Pi,\lambda(\mathsf{G})) \parallel \sigma}\ \textsc{AdaOutCont}$$

$$\frac{\begin{array}{c}\mathscr{M} \xrightarrow{\Pi!\lambda} \quad P \xrightarrow{\mathsf{s}[\mathsf{p}]!(\lambda(F),\mathsf{T})} P' \quad F(\sigma) = \mathsf{G} \quad \mathscr{M}_\mathsf{p} = \mathsf{G}\upharpoonright\mathsf{p} \quad \mathsf{T} \not\propto \mathscr{M}_\mathsf{p} \quad h\ \lambda\text{-free} \\ \Pi' = \mathsf{pa}(\mathsf{G}) \quad \forall \mathsf{q} \in \Pi'.\mathscr{M}_\mathsf{q} = \mathsf{G}\upharpoonright\mathsf{q} \quad \forall \mathsf{q} \in \Pi' \setminus \Pi.\ (P_\mathsf{q},\mathsf{T}_\mathsf{q}) \in \mathscr{P}\ \&\ \mathsf{T}_\mathsf{q} \propto \mathscr{M}_\mathsf{q}\end{array}}{\mathscr{M}[P] \mid \mathsf{s} : h \parallel \sigma \longrightarrow \prod_{\mathsf{q}\in\Pi'\setminus\Pi} \mathscr{M}_\mathsf{q}[P_\mathsf{q}\{\mathsf{s}[\mathsf{q}]/y_\mathsf{q}\}] \mid \mathsf{s} : h\cdot(\mathsf{p},\Pi,\lambda(\mathsf{G})) \parallel \sigma}\ \textsc{AdaOutNew}$$

$$\frac{N \longrightarrow N'}{\mathscr{E}[N] \parallel \sigma \longrightarrow \mathscr{E}[N'] \parallel \sigma}\ \text{SN} \qquad \frac{N \parallel \sigma \longrightarrow N' \parallel \sigma'}{\mathscr{E}[N] \parallel \sigma \longrightarrow \mathscr{E}[N'] \parallel \sigma'}\ \text{CTX}$$

Table 9 System reduction

ican factory becomes temporarily out of the current choreography. Without the given restriction, when the component becomes available again, we could have two monitored processes with the same session channel, so loosing channel linearity. Observe, however, that by this restriction some participants are allowed to finish their communications before performing an adaptation, while other participants have already self-adapted and then started the new communications.

Note that the λ-freeness of queues can be implemented in several ways without breaking decentralisation, for example by semaphores on queues.

We use $\longrightarrow^*$ with the usual meaning and $\longrightarrow_\mathscr{P}$, $\longrightarrow^*_\mathscr{P}$ when we want emphasise the use of $\mathscr{P}$ in rules Init, AdaInNew, AdaOutNew.

5 Examples

In this section we discuss examples. The first example motivates the restrictions on the definition of process types (Definition 7). The second and the third examples extend the example given in the Introduction, using the syntax of Section 2. Example 2 shows a use of different adaptation flags. Example 3 illustrates the possibility of exchanging the participant who is in charge of sending the adaptation flag. For readability we omit $\{,\}$ in writing global types and monitors.

Example 1 This example shows the necessity of the conditions on types given in Definition 7.

Suppose that any pre-type is a type, by removing the conditions from Definition 7. Then $\mathscr{P}$ could contain (P_1, T_1) and (P_2, T_2) such that:

$$P_1 = y!\ell(3).y?\ell'(x).\mathbf{0} + y!\ell(\mathsf{true}).\mathbf{0}$$
$$P_2 = y?\ell(x').y!\ell'(-x').\mathbf{0}$$
$$T_1 = !\ell(\mathsf{int}).?\ell'(\mathsf{int}).\mathsf{end} \wedge !\ell(\mathsf{Bool}).\mathsf{end}$$
$$\mathsf{T}_2 = ?\ell(\mathsf{int}).!\ell'(\mathsf{int}).\mathsf{end}$$

In particular, we observe that T_1 has an intersection between output types with the same label. Take then the global type $\mathsf{G} = 1 \to 2 : \ell(\mathsf{int}).2 \to 1 : \ell'(\mathsf{int}).\mathsf{end}$, whose projections on participants 1 and 2 are

$$\mathscr{M}_1 = 2!\ell(\mathsf{int}).2?\ell'(\mathsf{int}).\mathsf{end},\ \mathscr{M}_2 = 1?\ell(\mathsf{int}).1!\ell'(\mathsf{int}).\mathsf{end},$$

respectively. According to our definition of adequacy, T_1 and T_2 are adequate for $\mathscr{M}_1$ and $\mathscr{M}_2$, respectively. It is easy to verify that the network $\mathsf{new}(\mathsf{G})$ can reduce to

$$(\nu s)(\mathsf{s}[2]1!\ell'(-\mathsf{true}).\mathbf{0} \mid \mathsf{s} : \emptyset),$$

which is *stuck*. On the other hand, taking

$$P_1' = y!\ell(3).y?\ell'(x).\mathbf{0} + y?\ell(x).\mathbf{0}$$

with type $\mathsf{T}_1' = !\ell(\mathsf{int}).?\ell'(\mathsf{int}).\mathsf{end} \wedge ?\ell(\mathsf{Bool}).\mathsf{end}$, we still have that T_1' and T_2 are adequate for $\mathscr{M}_1$ and $\mathscr{M}_2$, but the network $\mathsf{new}(\mathsf{G})$ smoothly terminates the computation. Note that T_1' satisfies the conditions of Definition 7, so there is no possible ambiguity on which branch of the external choice must be chosen.

Example 2 Ada wants to consider the possibility to keep the business going, also when both factories are out of use. In this case she can choose either to stop all activities or to start the reconstruction of both factories. In the former case she sends a *stop* adaptation flag and in the latter case a *go* adaptation flag. The global type G_4 of the example in the Introduction must then be replaced by:

$$\mathsf{G}_4 = \mathsf{Ada} \to \{\mathsf{iS}, \mathsf{aS}\} : \{stop, go\}$$

If Ada decides to go on with the business when both factories are out, both sellers send their requests to Ada. So we need two new global types:

$$\mathsf{G}_g = \begin{cases} \mathsf{Ada} \to \mathsf{Bob} : \mathsf{String}. \\ \mathsf{iS} \to \mathsf{Ada} : req(\mathsf{String}, \mathsf{Int}). \\ \mathsf{aS} \to \mathsf{Ada} : req(\mathsf{String}, \mathsf{Int}). \\ \mathsf{Ada} \to \{\mathsf{iS}, \mathsf{aS}\} : \{check\} \end{cases} \qquad \mathsf{G}_s = \mathsf{end}$$

where *req* is the label used to ask the amount of an item, and two adaptation functions:

$$F_g(\mathsf{KO}, \mathsf{KO}) = \mathsf{G}_g \qquad F_s(\mathsf{KO}, \mathsf{KO}) = \mathsf{G}_s$$

A new process for implementing Ada can be:

$$\begin{aligned} Ada' = \mu X.y!(&check(F), \mathsf{T}_{Ada'}).X + \\ &y!help(\mathsf{contract}).(y!(check(F), \mathsf{T}_{Ada'}).X + \\ &\quad y?req(x,w).y?req(x',w').y!(check(F), \mathsf{T}_{Ada'}).X) + \\ &\mathsf{if}\ \ldots\ \mathsf{then}\ !(stop(F_s), \mathsf{end}).\mathbf{0}\ \mathsf{else}\ !(go(F_g), \mathsf{T}_{Ada'}).X \end{aligned}$$

where

$$\begin{aligned} \mathsf{T}_{Ada'} = {} & !check \wedge \\ & !\mathsf{String}.(!check \wedge ?(\mathsf{String}, \mathsf{Int}).?(\mathsf{String}, \mathsf{Int}).!check) \\ & \wedge (!stop \vee !go) \end{aligned}$$

is the new type of Ada. No modification is required for the processes representing the sellers and the factories.

Note that the process *Ada* of the Introduction could work as well as Ada' until both factories are out. In this case *Ada* would no longer agree with the monitor corresponding to the projection of the above G_4 and rule AdaOutNew would replace Ada' to *Ada*.

Example 3 Assume that Ada has to take a maternity leave. She then decides to transfer the job of monitoring the factories to one of the two sellers, who will play the role of deputy chief in the company. Then Ada writes iS (Italian seller) or aS (America seller) in the global data. The deputy chief is only in charge to keep the business working, unless both factories are out, in which case he also closes down the business. The deputy chief checks the state of the factories sending as Ada the reconfiguration flag *check*. When Ada is back, the standard management policy is restored.

In this example we use the same global types and adaptation function of the Introduction, but we add eight global types and another adaptation function.

The first four global types consider the case in which the American seller is the deputy chief. The global type G_1^a is the communication protocol when both factories are working:

$$\mathsf{G}_1^a = \begin{cases} \mathsf{iS} \to \mathsf{iF} : req(\mathsf{String}, \mathsf{Int}). \\ \mathsf{aS} \to \mathsf{aF} : req(\mathsf{String}, \mathsf{Int}). \\ \mathsf{aS} \to \{\mathsf{iS}, \mathsf{iF}, \mathsf{aF}\} : check \end{cases}$$

If the Italian factory is OK, but the American one is KO the communication protocol becomes:

$$\mathsf{G}_2^a = \begin{cases} \mathsf{iS} \to \mathsf{iF} : req(\mathsf{String}, \mathsf{Int}). \\ \mathsf{aS} \to \mathsf{iF} : req(\mathsf{String}, \mathsf{Int}). \\ \mathsf{aS} \to \{\mathsf{iS}, \mathsf{iF}\} : check \end{cases}$$

If the American factory is OK, but the Italian one is KO, the global type G_3^a is as expected.
When both factories are KO the deputy chief is forced to close the business:

$$\mathsf{G}_4^a = \mathsf{aS} \to \{\mathsf{iS}\} : bye.\mathsf{end}$$

The other four global types G_1^i-G_4^i prescribe that the adaptation flag is sent by the Italian rather than by the American seller.

The new adaptation function F', which has a third argument corresponding to the deputy chief in charge, is defined as:

$$\begin{array}{ll} F'(\mathsf{OK},\mathsf{OK},\mathsf{aS}) = \mathsf{G}_1^a & F'(\mathsf{OK},\mathsf{KO},\mathsf{aS}) = \mathsf{G}_2^a \\ F'(\mathsf{KO},\mathsf{OK},\mathsf{aS}) = \mathsf{G}_3^a & F'(\mathsf{KO},\mathsf{KO},\mathsf{aS}) = \mathsf{G}_4^a \\ F'(\mathsf{OK},\mathsf{OK},\mathsf{iS}) = \mathsf{G}_1^i & F'(\mathsf{OK},\mathsf{KO},\mathsf{iS}) = \mathsf{G}_2^i \\ F'(\mathsf{KO},\mathsf{OK},\mathsf{iS}) = \mathsf{G}_3^i & F'(\mathsf{KO},\mathsf{KO},\mathsf{iS}) = \mathsf{G}_4^i \end{array}$$

A process Ada'' implementing Ada includes a conditional to take into account the maternity leave:

$$Ada'' = \mu X.P + y!\mathit{help}(\text{contract}).P + !y.\mathit{bye}.\mathbf{0}$$

where help is the label for the contract with Bob and

$$\begin{array}{l} P = \text{if } \ldots \text{ then } !(\mathit{check}(F), \mathsf{T}_{Ada''}).X \\ \qquad\qquad \text{else write } \mathsf{dep}.!(\mathit{check}(F'), \text{end}).\mathbf{0} \end{array}$$

with $\mathsf{dep} = \mathsf{aS}$ or $\mathsf{dep} = \mathsf{iS}$ and

$$\mathsf{T}_{Ada''} = !\mathit{check} \wedge !\mathit{help}(\text{String}).!\mathit{check} \wedge !\mathit{bye}.\text{end}$$

A process implementing sellers uses the adaptation function F' or F according to whether or not Ada is on leave:

$$\begin{array}{l} \mathit{Seller}' = \mu X.y!\mathit{req}(\text{item}, \text{amount}).(y?(\mathit{check}, \mathsf{T}_{\mathit{Seller}'}).X + \\ \qquad\qquad \text{if } \ldots \text{ then } y!(\mathit{check}(F), \mathsf{T}_{\mathit{Seller}'}).X \\ \qquad\qquad\quad \text{else } y!(\mathit{check}(F'), \mathsf{T}_{\mathit{Seller}'}).X) \\ \qquad + y?\mathit{bye}.\mathbf{0} + y!\mathit{bye}.\mathbf{0} \end{array}$$

where:

$$\begin{array}{l} \mathsf{T}_{\mathit{Seller}'} = !\mathit{req}(\text{String}, \text{Int}).(?\mathit{check} \\ \qquad \wedge !\mathit{check}) \wedge ?\mathit{bye}.\text{end} \wedge !\mathit{bye}.\text{end} \end{array}$$

No modification is needed, instead, for the processes representing the other participants, i.e. the two factories and Bob.

Notice that $\mathsf{T}_{Ada''}$ differs from T_{Ada} (see page 6) only for the label help (since in the Introduction we used a simplified syntax). Instead $\mathsf{T}_{\mathit{Seller}'}$ offers more choices that the type of the process *Seller* in the Introduction. As a consequence, the process *Ada* of the Introduction is an implementation of Ada also in the present new scenario, which never uses the right of the maternity leave. Instead the process *Seller* is no longer adequate since it does not implement the chief's behaviour.

6 Safety

In this section, we show subject reduction and progress theorems, following essentially the proof pattern of similar results for multiparty sessions, see e.g. [13]. Indeed, the main innovation of our calculus is that global types, with the corresponding monitors, are reconfigured at each adaptation step. Furthermore, participants of two different global types can coexist inside the same session. This happens when some participants have already performed the adaptation and then they are following the new global type, while other participants are still completing the interactions prescribed by the old global type. These are the crucial technical difficulties in proving that monitored well-typed processes always behave in a type safe way. Therefore, we need to introduce typing rules for systems, which associate types to session channels.

The type of channel $\mathsf{s}[\mathsf{p}]$ is formed by taking into account the monitor, which controls the process owning $\mathsf{s}[\mathsf{p}]$, and the messages in the queue of the session s. The type of a monitored process is the association of the monitor to the session channel owned by the process. For each value message $(\mathsf{p}, \Pi, \ell(v))$ in the queue of session s we associate the type $\Pi!\ell(S)$ to $\mathsf{s}[\mathsf{p}]$, where S is the sort of the value v, preserving the order of messages of queue. So lists of types of this shape form the types of session channels.

A queue can also contain adaptation messages. Note that, thanks to the condition of λ-freeness in rules AdaOutCont and AdaOutNew (Table 9), at most one adaptation message can occur in a queue (modulo structural equivalence, see Table 7). If the queue of session s contains the message $(\mathsf{p}, \Pi, \lambda(\mathsf{G}))$, then $\Pi!\lambda$ occurs in the type of $\mathsf{s}[\mathsf{p}]$. A *message type* is then a list of types of the shape $\Pi!\ell(S)$ possibly containing a type of the shape $\Pi!\lambda$. After receiving the message $(\mathsf{p}, \Pi, \lambda(\mathsf{G}))$, each participant $\mathsf{q} \in \Pi$ of session s will behave according to the monitor $\mathsf{G} \upharpoonright \mathsf{q}$. Therefore the type of $\mathsf{s}[\mathsf{q}]$ can involve two monitors. One (explicit) monitor (dubbed *active monitor*) is the monitor of the monitored process owing $\mathsf{s}[\mathsf{p}]$. The other (implicit) monitor (dubbed *virtual monitor*) is the projection onto q of the global type contained in the adaptation message waiting to be received by $\mathsf{s}[\mathsf{q}]$ (and to become active). A missing virtual monitor is denoted by "$-$". In particular, the virtual monitor of the sender of the adaptation message is always missing. So typing rules for queues associate types of the shape $\langle$message type, virtual monitor$\rangle$ (corresponding to the sent messages and the virtual monitor) to session channels.

To sum up, a type of a session channel is either an active monitor, or a pair consisting of a message type and a virtual monitor, or a triple consisting of a message type, an active monitor and a virtual monitor.

Definition 9 *Message types*, *queue types*, *virtual monitors*, and *generalised types* are defined by:

$$\begin{array}{llll} \text{Message Types} & \mathsf{m} & ::= & \varepsilon \mid \Pi!\ell(S) \mid \Pi!\lambda \mid \mathsf{m};\mathsf{m} \\ \text{Virtual Monitors} & \mathscr{V} & ::= & \mathscr{M} \mid - \\ \text{Queue Types} & \mathscr{Q} & ::= & \langle \mathsf{m}, \mathscr{V} \rangle \\ \text{Generalised Types} & \chi & ::= & \mathscr{M} \mid \mathscr{Q} \mid \langle \mathsf{m}, \mathscr{M}, \mathscr{V} \rangle \end{array}$$

where "$;$" is associative, ε is the type of the empty sequence of messages, such that $\varepsilon;\mathsf{m} = \mathsf{m};\varepsilon = \mathsf{m}$, and "$-$" denotes a missing monitor.

The typing judgements for systems are of the shape

$$\vdash_\Sigma \mathscr{S} \rhd \Delta$$

where Σ is a set of session names (the names of the queues

$$(\Pi!\ell(S);\mathsf{m})\upharpoonright\mathsf{q}=\begin{cases}!\ell(S).(\mathsf{m}\upharpoonright\mathsf{q}) & \text{if } \mathsf{q}\in\Pi\\ \mathsf{m}\upharpoonright\mathsf{q} & \text{otherwise}\end{cases}\qquad(\Pi!\lambda;\mathsf{m})\upharpoonright\mathsf{q}=\begin{cases}!\lambda.(\mathsf{m}\upharpoonright\mathsf{q}) & \text{if } \mathsf{q}\in\Pi\\ \mathsf{m}\upharpoonright\mathsf{q} & \text{otherwise}\end{cases}\qquad \varepsilon\upharpoonright\mathsf{q}=\varepsilon$$

$$\mathsf{p}?\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I}\upharpoonright\mathsf{q}=\begin{cases}?\{\ell_i(S_i).(\mathscr{M}_i\upharpoonright\mathsf{q})\}_{i\in I} & \text{if } \mathsf{q}=\mathsf{p}\\ \mathscr{M}_{i_0}\upharpoonright\mathsf{q} & \text{where } i_0\in I, \text{if } \mathsf{q}\neq\mathsf{p} \text{ and } \mathscr{M}_i\upharpoonright\mathsf{q}=\mathscr{M}_j\upharpoonright\mathsf{q}\,\forall i,j\in I\end{cases}$$

$$\Pi!\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I}\upharpoonright\mathsf{q}=\begin{cases}!\{\ell_i(S_i).(\mathscr{M}_i\upharpoonright\mathsf{q})\}_{i\in I} & \text{if } \mathsf{q}\in\Pi\\ \mathscr{M}_{i_0}\upharpoonright\mathsf{q} & \text{where } i_0\in I, \text{if } \mathsf{q}\notin\Pi \text{ and } \mathscr{M}_i\upharpoonright\mathsf{q}=\mathscr{M}_j\upharpoonright\mathsf{q}\,\forall i,j\in I\end{cases}$$

$$\mathsf{p}?\{\lambda_i\}_{i\in I}\upharpoonright\mathsf{q}=\begin{cases}?\{\lambda_i\}_{i\in I} & \text{if } \mathsf{q}=\mathsf{p}\\ \varepsilon & \text{otherwise}\end{cases}\qquad\Pi!\{\lambda_i\}_{i\in I}\upharpoonright\mathsf{q}=\begin{cases}!\{\lambda_i\}_{i\in I} & \text{if } \mathsf{q}\in\Pi\\ \varepsilon & \text{otherwise}\end{cases}$$

$$\mathsf{end}\upharpoonright\mathsf{q}=\varepsilon\qquad-\upharpoonright\mathsf{q}=\varepsilon$$

$$\langle\mathsf{m},\mathscr{V}\rangle\upharpoonright\mathsf{q}=\mathsf{m}\upharpoonright\mathsf{q}.\mathscr{V}\upharpoonright\mathsf{q}\qquad\langle\mathsf{m},\mathscr{M},\mathscr{V}\rangle\upharpoonright\mathsf{q}=\mathsf{m}\upharpoonright\mathsf{q}.\mathscr{M}\upharpoonright\mathsf{q}.\mathscr{V}\upharpoonright\mathsf{q}$$

Table 10 Projection of generalised types onto participants

$$\Theta\bowtie\Theta_j\ \&\ j\in I \text{ imply } !\ell_j(S_j).\Theta\bowtie?\{\ell_i(S_i).\Theta_i\}_{i\in I}$$
$$\forall i\in I\ \Theta_i\bowtie\Theta_i'\ \text{ imply } !\{\ell_i(S_i).\Theta_i\}_{i\in I}\bowtie?\{\ell_i(S_i).\Theta_i'\}_{i\in I}$$
$$j\in I \text{ implies } !\lambda_j\bowtie?\{\lambda_i\}_{i\in I}\qquad!\{\lambda_i\}_{i\in I}\bowtie?\{\lambda_i\}_{i\in I}\qquad\varepsilon\bowtie\varepsilon$$
$$\Theta_1\bowtie\Theta_2 \text{ and } \Theta_3\bowtie\Theta_4 \text{ imply } \Theta_1.\Theta_3\bowtie\Theta_2.\Theta_4$$

Table 11 Duality between projections of generalised types onto participants

which occur free in the network) and Δ is a *session typing*. *Session typings* associate session channels to generalised types:

$$\Delta ::= \emptyset \mid \Delta,\mathsf{s}[\mathsf{p}]:\chi$$

We apply to the session typings the same conventions used for environments. In particular a session typing Δ_1,Δ_2 is defined only if the domains of Δ_1 and Δ_2 are disjoint.

To ensure type safety it is essential that the communications are performed in a consistent way, i.e. that data are exchanged in the right order and with the right type. Consistency of session typings is defined using *projection of generalised types* and *duality*, given in Tables 10 and 11, respectively. The projection of a generalised type onto a participant q represents the communications offered to q.

The projection of generalised types uses the projection of message types and virtual monitors. We denote these projections by $\chi\upharpoonright\mathsf{q}$, $\mathsf{m}\upharpoonright\mathsf{q}$ and $\mathscr{V}\upharpoonright\mathsf{q}$, respectively. The conditions on the equalities of projections correspond to the similar conditions in Table 2. The projection of a generalised type of the shape $\langle\mathsf{m},\mathscr{M},\mathscr{V}\rangle$ is the concatenation of the projections of m, $\mathscr{M}$ and $\mathscr{V}$. This is meaningful since m represents the message already sent, $\mathscr{M}$ guides the behaviour of the participant before its adaptation and $\mathscr{V}$ will guide its behaviour after the adaptation.
Projection of generalised types (ranged over by Θ) are defined by the following syntax

$$\Theta ::= !\ell(S)\mid!\lambda\mid\Theta.\Theta\mid?\{\Theta_i\}_{i\in I}\mid!\{\Theta_i\}_{i\in I}\mid\varepsilon$$

We assume $\varepsilon.\Theta=\Theta.\varepsilon=\Theta$, since ε represents no communication.

We write $\Theta\bowtie\Theta'$ when Θ and Θ' are dual according to the definition of Table 11. Note that duality is defined only on Θ which are projections of generalised types.

We can now define consistency as duality of projections.

Definition 10 A session typing Δ is *consistent for the session* s, notation $\mathsf{con}(\Delta,\mathsf{s})$, if $\mathsf{s}[\mathsf{p}]:\chi\in\Delta$ and $\mathsf{s}[\mathsf{q}]:\chi'\in\Delta$ with $\mathsf{p}\neq\mathsf{q}$ imply $\chi\upharpoonright\mathsf{q}\bowtie\chi'\upharpoonright\mathsf{p}$. A session typing is *consistent* if it is consistent for all sessions which occur in it.

It is easy to check that projections of the same global type are always dual.

Proposition 1 *Let* G *be a global type and* $\mathsf{p}\neq\mathsf{q}$.
Then $(\mathsf{G}\upharpoonright\mathsf{p})\upharpoonright\mathsf{q}\bowtie(\mathsf{G}\upharpoonright\mathsf{q})\upharpoonright\mathsf{p}$.

This proposition ensures that session typings obtained by projecting global types are consistent.

Table 12 gives the typing rules for systems. A session initiator is typed with the empty set of session names and with the empty session typing (rule NEW). To type a monitored process, we distinguish two cases. If the monitor is end, then the session typing is empty for any process P (rule endP). Otherwise, the channel owned by the process is associated to the monitor, provided that the type of the process (Table 4) is adequate for the monitor, according to Definition 8 (rule MP).

The next three rules type named queues. In these rules the turn-style is decorated by the name of the queue. An empty queue $\emptyset$ is typed with the empty session typing (QINT). Two queue types can be composed only if at most one of them contains a monitor, while the sequence of message types is a message type. Then we define the operator $\sharp$ by:

$$\langle\mathsf{m},\mathscr{V}\rangle\sharp\langle\mathsf{m}',-\rangle=\langle\mathsf{m},-\rangle\sharp\langle\mathsf{m}',\mathscr{V}\rangle=\langle\mathsf{m};\mathsf{m}',\mathscr{V}\rangle$$

$$\frac{}{\vdash_\emptyset \mathsf{new}(\mathsf{G}) \rhd \emptyset}\ \textsc{New} \qquad \frac{}{\vdash_\emptyset \mathsf{end}[P] \rhd \emptyset}\ \mathsf{endP} \qquad \frac{\vdash P \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T} \quad \mathscr{M} \neq \mathsf{end} \quad \mathsf{T} \propto \mathscr{M}}{\vdash_\emptyset \mathscr{M}[P] \rhd \{\mathsf{s}[\mathsf{p}] : \mathscr{M}\}}\ \textsc{MP}$$

$$\frac{}{\vdash_{\{\mathsf{s}\}} \mathsf{s} : \emptyset \rhd \emptyset}\ \textsc{QInit} \qquad \frac{\vdash_{\{\mathsf{s}\}} \mathsf{s} : h \rhd \Delta \qquad \vdash v : S}{\vdash_{\{\mathsf{s}\}} \mathsf{s} : h \cdot (\mathsf{p}, \Pi, \ell(v)) \rhd \Delta \sharp \{\mathsf{s}[\mathsf{p}] : \langle \Pi ! \ell(S), - \rangle\}}\ \textsc{QSendV}$$

$$\frac{\vdash_{\{\mathsf{s}\}} \mathsf{s} : h \rhd \Delta}{\vdash_{\{\mathsf{s}\}} \mathsf{s} : h \cdot (\mathsf{p}, \Pi, \lambda(\mathsf{G})) \rhd \Delta \sharp (\{\mathsf{s}[\mathsf{p}] : \langle \Pi ! \lambda, - \rangle\} \cup \{\mathsf{s}[\mathsf{q}] : \langle \varepsilon, \mathsf{G} \upharpoonright \mathsf{q} \rangle \mid \mathsf{q} \in \Pi\})}\ \textsc{QSendG}$$

$$\frac{\vdash_{\Sigma_1} N_1 \rhd \Delta_1 \qquad \vdash_{\Sigma_2} N_2 \rhd \Delta_2 \qquad \Sigma_1 \cap \Sigma_2 = \emptyset}{\vdash_{\Sigma_1 \cup \Sigma_2} N_1 \mid N_2 \rhd \Delta_1 * \Delta_2}\ \textsc{NPar} \qquad \frac{\vdash_\Sigma N \rhd \Delta \qquad \Delta \approx \Delta'}{\vdash_\Sigma N \rhd \Delta'}\ \textsc{Equiv}$$

$$\frac{\vdash_\Sigma N \rhd \Delta \qquad \mathsf{con}(\Delta, \mathsf{s})}{\vdash_{\Sigma \setminus \{\mathsf{s}\}} (\nu\, \mathsf{s}) N \rhd \Delta \setminus \mathsf{s}}\ \textsc{Res} \qquad \frac{\vdash_\Sigma N \rhd \Delta}{\vdash_\Sigma N \parallel \sigma \rhd \Delta}\ \textsc{System}$$

Table 12 Typing rules for networks and systems

Rules QSendV and QSendG use the extension of $\sharp$ to session typings:

$$\begin{aligned}\Delta \sharp \Delta' = {} & \{\mathsf{s}[\mathsf{p}] : \chi \sharp \chi' \mid \mathsf{s}[\mathsf{p}] : \chi \in \Delta \ \&\ \mathsf{s}[\mathsf{p}] : \chi' \in \Delta'\} \cup \\ & \{\mathsf{s}[\mathsf{p}] : \chi \mid \mathsf{s}[\mathsf{p}] : \chi \in \Delta \cup \Delta' \ \& \\ & \qquad \mathsf{s}[\mathsf{p}] \notin dom(\Delta) \cap dom(\Delta')\}\end{aligned}$$

Notice that in rules QSendV and QSendG the session typings only contain queue types. The queue type $\langle \Pi ! \ell(S), - \rangle$ is pushed in the queue type of $\mathsf{s}[\mathsf{p}]$ for a value message $(\mathsf{p}, \Pi, \ell(v))$, where S is the sort of v (rule QSendV). The queue type $\langle \Pi ! \lambda, - \rangle$ is pushed in the queue type of $\mathsf{s}[\mathsf{p}]$ for an adaptation message $(\mathsf{p}, \Pi, \lambda(\mathsf{G}))$, while the queue type of $\mathsf{s}[\mathsf{q}]$ has the projection of G on q as virtual monitor, for all $\mathsf{q} \in \Pi$ (rule QSendG).

For typing the parallel composition of networks, rule NPar prescribes that each named queue does not occur twice (condition $\Sigma_1 \cap \Sigma_2 = \emptyset$) and composes session typings forming a generalised type out of a queue type and a monitor. We define the composition $*$ between queue types and monitors as:

$$\langle \mathsf{m}, \mathscr{V} \rangle * \mathscr{M} = \mathscr{M} * \langle \mathsf{m}, \mathscr{V} \rangle = \langle \mathsf{m}, \mathscr{M}, \mathscr{V} \rangle$$

We extend $*$ to generalised types and to session typings as expected:

$$\begin{aligned}\Delta * \Delta' = {} & \{\mathsf{s}[\mathsf{p}] : \chi * \chi' \mid \mathsf{s}[\mathsf{p}] : \chi \in \Delta \ \&\ \mathsf{s}[\mathsf{p}] : \chi' \in \Delta'\} \cup \\ & \{\mathsf{s}[\mathsf{p}] : \chi \mid \mathsf{s}[\mathsf{p}] : \chi \in \Delta \cup \Delta' \ \& \\ & \qquad \mathsf{s}[\mathsf{p}] \notin dom(\Delta) \cap dom(\Delta')\}\end{aligned}$$

For example, if $\vdash_\Sigma N \rhd \Delta$, then $\vdash_\Sigma \mathsf{end}[P] \mid N \rhd \emptyset * \Delta$ (by rules endP and NPar) and $\emptyset * \Delta = \Delta$; this fits with the structural equivalence $\mathsf{end}[P] \mid N \equiv N$.

Notice that both $\sharp$ and $*$ are partial operators on session typings, since they can be undefined when applied to arbitrary generalised types.

In order to take into account the structural congruence between queues (see Table 7), we consider message types modulo the equivalence relation $\approx$ induced by the following rules (where Z stands for either $\ell(S)$ or λ):

$$\begin{aligned}\mathsf{m}; \Pi ! Z; \Pi' ! Z'; \mathsf{m}' &\approx \mathsf{m}; \Pi' ! Z'; \Pi ! Z; \mathsf{m}' \\ &\qquad \text{if } \Pi \cap \Pi' = \emptyset \\ \mathsf{m}; \Pi ! Z; \mathsf{m}' &\approx \mathsf{m}; \Pi_1 ! Z; \Pi_2 ! Z; \mathsf{m}' \\ &\qquad \text{if } \Pi = \Pi_1 \cup \Pi_2, \Pi_1 \cap \Pi_2 = \emptyset\end{aligned}$$

This equivalence relation on message types extends to generalised types by:

$$\begin{gathered}\mathsf{m} \approx \mathsf{m}' \text{ implies} \\ \langle \mathsf{m}, \mathscr{V} \rangle \approx \langle \mathsf{m}', \mathscr{V} \rangle \text{ and } \langle \mathsf{m}, \mathscr{M}, \mathscr{V} \rangle \approx \langle \mathsf{m}', \mathscr{M}, \mathscr{V} \rangle\end{gathered}$$

We say that two session typings Δ and Δ' are equivalent (notation $\Delta \approx \Delta'$) if
$\mathsf{s}[\mathsf{p}] : \chi \in \Delta$ implies $\mathsf{s}[\mathsf{p}] : \chi' \in \Delta'$ with $\chi \approx \chi'$ and vice versa. Rule Equiv allows to use this equivalence relation.

Rule Res requires the session typing to be consistent for the session s in order to type the restriction on s.

A system can be typed if the network can be typed, while the global state is arbitrary, see rule System.

A crucial observation is that virtual monitors occur in generalised types only if queues contain adaptation flags. In other words using the condition of being λ-free (that is a premise of rules AdaOutCont and AdaOutNew) we get:

If $\vdash_\mathsf{s} \mathsf{s} : h \rhd \Delta$ and h is λ-free, then no virtual monitor occurs in Δ.

It is standard to prove an inversion lemma for networks and systems by induction on derivations (Table 12).

Lemma 2 *(Inversion Lemma)*

1. *If* $\vdash_\Sigma \mathsf{new}(\mathsf{G}) \triangleright \Delta$, *then* $\Sigma = \Delta = \emptyset$.
2. *If* $\vdash_\Sigma \mathsf{end}[P] \triangleright \Delta$, *then* $\Sigma = \Delta = \emptyset$.
3. *If* $\vdash_\Sigma \mathscr{M}[P] \triangleright \Delta$ *and* $\mathscr{M} \neq \mathsf{end}$, *then* $\Sigma = \emptyset$ *and* $\Delta = \{\mathsf{s}[\mathsf{p}] : \mathscr{M}\}$ *and* $\vdash P \triangleright \mathsf{s}[\mathsf{p}] : \mathsf{T}$ *and* $\mathsf{T} \propto \mathscr{M}$.
4. *If* $\vdash_\Sigma \mathsf{s} : \phi \triangleright \Delta$, *then* $\Sigma = \{\mathsf{s}\}$ *and* $\Delta = \emptyset$.
5. *If* $\vdash_\Sigma \mathsf{s} : h \cdot (\mathsf{p}, \Pi, \ell(v)) \triangleright \Delta$, *then* $\Sigma = \{\mathsf{s}\}$ *and* $\Delta \approx \Delta' \sharp \{\mathsf{s}[\mathsf{p}] : \langle \Pi ! \ell(S), - \rangle\}$ *and* $\vdash_{\{\mathsf{s}\}} \mathsf{s} : h \triangleright \Delta'$ *and* $\vdash v : S$.
6. *If* $\vdash_\Sigma \mathsf{s} : h \cdot (\mathsf{p}, \Pi, \lambda(\mathsf{G})) \triangleright \Delta$, *then* $\Sigma = \{\mathsf{s}\}$ *and* $\Delta \approx \Delta' \sharp (\{\mathsf{s}[\mathsf{p}] : \langle \Pi ! \lambda, - \rangle\} \cup \{\mathsf{s}[\mathsf{q}] : \langle \varepsilon, \mathsf{G} \upharpoonright \mathsf{q} \rangle \mid \mathsf{q} \in \Pi\})$ *and* $\vdash_{\{\mathsf{s}\}} \mathsf{s} : h \triangleright \Delta'$.
7. *If* $\vdash_\Sigma N_1 \mid N_2 \triangleright \Delta$, *then* $\Sigma = \Sigma_1 \cup \Sigma_2$ *and* $\Delta = \Delta_1 * \Delta_2$ *and* $\vdash_{\Sigma_1} N_1 \triangleright \Delta_1$ *and* $\vdash_{\Sigma_2} N_2 \triangleright \Delta_2$ *and* $\Sigma_1 \cap \Sigma_2 = \emptyset$.
8. *If* $\vdash_\Sigma (\nu \mathsf{s}) N \triangleright \Delta$, *then* $\Sigma = \Sigma' \setminus \{\mathsf{s}\}$ *and* $\Delta = \Delta' \setminus \mathsf{s}$ *and* $\vdash_{\Sigma'} N \triangleright \Delta'$ *and* $\mathsf{con}(\Delta, \mathsf{s})$.
9. *If* $\vdash_\Sigma N \parallel \sigma \triangleright \Delta$, *then* $\vdash_\Sigma N \triangleright \Delta$.

We also need a lemma stating how the typing depends on the first message on the queue. The proof follows immediately from the typing rules of queues.

Lemma 3 1. *If* $\vdash_\Sigma \mathsf{s} : (\mathsf{p}, \Pi, \ell(v)) \cdot h \triangleright \Delta$, *then* $\Sigma = \{\mathsf{s}\}$ *and* $\Delta \approx \{\mathsf{s}[\mathsf{p}] : \langle \Pi ! \ell(S), - \rangle\} \sharp \Delta'$ *and* $\vdash_{\{\mathsf{s}\}} \mathsf{s} : h \triangleright \Delta'$ *and* $\vdash v : S$.
2. *If* $\vdash_\Sigma \mathsf{s} : (\mathsf{p}, \Pi, \lambda(\mathsf{G})) \cdot h \triangleright \Delta$, *then* $\Sigma = \{\mathsf{s}\}$ *and* $\Delta \approx (\{\mathsf{s}[\mathsf{p}] : \langle \Pi ! \lambda, - \rangle\} \cup \{\mathsf{s}[\mathsf{q}] : \langle \varepsilon, \mathsf{G} \upharpoonright \mathsf{q} \rangle \mid \mathsf{q} \in \Pi\}) \sharp \Delta'$ *and* $\vdash_{\{\mathsf{s}\}} \mathsf{s} : h \triangleright \Delta'$.

Monitor LTS transactions reveal the monitor shapes, as detailed in the next lemma, which can be proved by straightforward case analysis.

Lemma 4 1. *If* $\mathscr{M} \xrightarrow{\mathsf{p}?\ell} \mathscr{M}'$, *then* $\mathscr{M} = \mathsf{p}?\{\ell_i(S_i).\mathscr{M}_i\}_{i \in I}$ *and* $\ell = \ell_j$ *and* $\mathscr{M}' = \mathscr{M}_j$ *for some* $j \in I$.
2. *If* $\mathscr{M} \xrightarrow{\Pi ! \ell} \mathscr{M}'$, *then* $\mathscr{M} = \Pi ! \{\ell_i(S_i).\mathscr{M}_i\}_{i \in I}$ *and* $\ell = \ell_j$ *and* $\mathscr{M}' = \mathscr{M}_j$ *for some* $j \in I$.
3. *If* $\mathscr{M} \xrightarrow{\mathsf{p}?\lambda}$, *then* $\mathscr{M} = \mathsf{p}?\{\lambda_i\}_{i \in I}$ *and* $\lambda = \lambda_j$ *for some* $j \in I$.
4. *If* $\mathscr{M} \xrightarrow{\Pi ! \lambda}$, *then* $\mathscr{M} = \Pi ! \{\lambda_i\}_{i \in I}$ *and* $\lambda = \lambda_j$ *for some* $j \in I$.

The following lemma relates communications offered by processes (as LTS transactions) with their types.

Lemma 5 1. *If* $P \xrightarrow{\mathsf{s}[\mathsf{p}]?\ell(v)} P'$ *and* $\vdash P \triangleright \mathsf{s}[\mathsf{p}] : \mathsf{T}$, *then either* $P = \mathsf{s}[\mathsf{p}]?\ell(x).P_0$ *and* $\mathsf{T} = ?\ell(S).\mathsf{T}'$ *or* $P = \mathsf{s}[\mathsf{p}]?\ell(x).P_0 + P''$ *and* $\mathsf{T} = ?\ell(S).\mathsf{T}' \wedge \mathsf{T}''$, *and in both cases* $\vdash \mathsf{s}[\mathsf{p}]?\ell(x).P_0 \triangleright \mathsf{s}[\mathsf{p}] : ?\ell(S).\mathsf{T}'$ *and* $P' = P_0\{v/x\}$.
2. *If* $P \xrightarrow{\mathsf{s}[\mathsf{p}]!\ell(v)} P'$ *and* $\vdash P \triangleright \mathsf{s}[\mathsf{p}] : \mathsf{T}$, *then either* $\mathsf{T} = !\ell(S).\mathsf{T}'$ *or* $\mathsf{T} = !\ell(S).\mathsf{T}' \wedge \mathsf{T}''$, *and* $\vdash P' \triangleright \mathsf{s}[\mathsf{p}] : \mathsf{T}'$ *and* $\vdash v : S$.
3. *If* $P \xrightarrow{\mathsf{s}[\mathsf{p}]?(\lambda, \mathsf{T}')} P'$ *and* $\vdash P \triangleright \mathsf{s}[\mathsf{p}] : \mathsf{T}$, *then either* $\mathsf{T} = ?\lambda$ *or* $\mathsf{T} = ?\lambda \wedge \mathsf{T}''$, *and* $\vdash P' \triangleright \mathsf{s}[\mathsf{p}] : \mathsf{T}'$.
4. *If* $P \xrightarrow{\mathsf{s}[\mathsf{p}]!(\lambda(F), \mathsf{T}')} P'$ *and* $\vdash P \triangleright \mathsf{s}[\mathsf{p}] : \mathsf{T}$, *then either* $\mathsf{T} = !\lambda$ *or* $\mathsf{T} = !\lambda \wedge \mathsf{T}''$, *and* $\vdash P' \triangleright \mathsf{s}[\mathsf{p}] : \mathsf{T}'$.

Proof The proof is by structural induction on P. We show only Point (1), the proof for the other points being simpler. If $P \xrightarrow{\mathsf{s}[\mathsf{p}]?\ell(v)} P'$, then either $P = \mathsf{s}[\mathsf{p}]?\ell(x).P_0$ or $P = P_1 + P_2$ and $P_i \xrightarrow{\mathsf{s}[\mathsf{p}]?\ell(v)} P'$ for $i = 1$ or $i = 2$. In the first case P is typed by rule RCV and $\mathsf{T} = ?\ell(S).\mathsf{T}'$. In the second case P is typed by rule CHOICE. Then $\vdash P_i \triangleright \mathsf{s}[\mathsf{p}] : \mathsf{T}_i$ and $\mathsf{T} = \mathsf{T}_1 \wedge \mathsf{T}_2$ for $i = 1, 2$. By induction, either $P_i = \mathsf{s}[\mathsf{p}]?\ell(x).P_0$ and $\mathsf{T}_i = ?\ell(S).\mathsf{T}'$ or $P_i = \mathsf{s}[\mathsf{p}]?\ell(x).P_0 + P_i'$ and $\mathsf{T}_i = ?\ell(S).\mathsf{T}' \wedge \mathsf{T}''$ for $i = 1$ or $i = 2$. In both cases $\vdash \mathsf{s}[\mathsf{p}]?\ell(x).P_0 \triangleright \mathsf{s}[\mathsf{p}] : ?\ell(S).\mathsf{T}'$ and $P' = P_0\{v/x\}$. □

As usual, session types are not preserved under system reduction: they evolve according to the actions performed by the corresponding participants. This is formalised by the reduction rules given in Table 13, where message types are considered modulo the equivalence relation defined above. The rules in the first line allow us to create monitors and queue types. The rules in the second line get rid of types carrying no information. The subsequent four rules deal with outputs and inputs of labels with sorts and flags. In particular, the rule in the second to last line shows how a virtual monitor becomes the current monitor when a participant adapts itself.

Notice that not all the left-hand sides of the reduction rules for networks and systems are typed by consistent session typings. For example,

$$\vdash_{\{\mathsf{s}\}} \mathscr{M}[\mathsf{s}[1]?\ell(x).\mathsf{s}[1]?\ell'(y).\mathbf{0}] \mid \mathsf{s} : (2, 1, \ell(\mathsf{true})) \triangleright \Delta$$

where $\mathscr{M} = 2?\ell(\mathsf{Bool}).2?\ell'(\mathsf{Int}).\mathsf{end}$,
$\Delta = \{\mathsf{s}[1] : \mathscr{M}, \mathsf{s}[2] : \langle 1!\ell(\mathsf{Bool}), - \rangle\}$. Observe that

$$\mathscr{M}[\mathsf{s}[1]?\ell(x).\mathsf{s}[1]?\ell'(y).\mathbf{0}] \mid \mathsf{s} : (2, 1, \ell(\mathsf{true}))$$

matches the left-hand side of the reduction rule IN and Δ is not consistent. The network obtained by putting this network in parallel with $1!\ell'(\mathsf{Int}).\mathsf{end}[\mathsf{s}[2]!\ell'(7).\mathbf{0}]$ has a consistent session typing. It is then crucial to show that if the left-hand side of a reduction rule is typed by a session typing, which is consistent when composed with some session typing, then the same property holds for the right-hand side too. It is sufficient to consider the reduction rules which do not contain network and system reductions as premises, i.e. which are the leafs in the reduction trees. This is formalised in the following lemma, which is the key step for proving the Subject Reduction Theorem.

Lemma 6 *(Key Lemma)*

$$\emptyset \implies \{\mathsf{s}[\mathsf{p}] : \mathscr{M}\} \qquad \emptyset \implies \{\mathsf{s}[\mathsf{p}] : \langle \varepsilon, \mathscr{V}\rangle\}$$
$$\{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, \mathsf{end}, -\rangle\} \implies \{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, -\rangle\} \qquad \{\mathsf{s}[\mathsf{p}] : \langle \varepsilon, -\rangle\} \implies \emptyset$$
$$\{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, \Pi!\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I}, \mathscr{V}\rangle\} \implies \{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}; \Pi!\ell_j(S_j), \mathscr{M}_j, \mathscr{V}\rangle\}\ j \in I$$
$$\{\mathsf{s}[\mathsf{p}] : \langle \mathsf{q}!\ell_j(S_j); \mathsf{m}, \mathscr{V}\rangle, \mathsf{s}[\mathsf{q}] : \langle \mathsf{m}', \mathsf{p}?\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I}, \mathscr{V}'\rangle\} \implies \{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, \mathscr{V}\rangle, \mathsf{s}[\mathsf{q}] : \langle \mathsf{m}', \mathscr{M}_j, \mathscr{V}'\rangle\}\ j \in I$$
$$\{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, \Pi!\{\lambda_i\}_{i\in I}, -\rangle\} \implies \{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}; \Pi!\lambda_j, \mathscr{M}, -\rangle\}\ j \in I$$
$$\{\mathsf{s}[\mathsf{p}] : \langle \mathsf{q}!\lambda_j; \mathsf{m}, -\rangle, \mathsf{s}[\mathsf{q}] : \langle \mathsf{m}', \mathsf{p}?\{\lambda_i\}_{i\in I}, \mathscr{V}\rangle\} \implies \{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, -\rangle,\ \mathsf{s}[\mathsf{q}] : \langle \mathsf{m}', \mathscr{V}, -\rangle\}\ j \in I$$
$$\Delta_1 \sharp \Delta \implies \Delta_2 \sharp \Delta \ \text{ if } \Delta_1 \implies \Delta_2 \qquad \Delta_1 * \Delta \implies \Delta_2 * \Delta \ \text{ if } \Delta_1 \implies \Delta_2$$

Table 13 Reduction of session typings

1. *Let $\vdash_\Sigma N \rhd \Delta$, and $N \longrightarrow N'$ be obtained by any reduction rule different from* EQUIV*, and $\Delta * \Delta_0$ be consistent for some Δ_0. Then there is Δ' such that $\vdash_\Sigma N' \rhd \Delta'$ and $\Delta \implies^* \Delta'$ and $\Delta' * \Delta_0$ is consistent.*
2. *Let $\vdash_\Sigma \mathscr{S} \rhd \Delta$, and $\mathscr{S} \longrightarrow \mathscr{S}'$ be obtained by any reduction rule different from* SN, CTX*, and $\Delta * \Delta_0$ be consistent for some Δ_0. Then there is Δ' such that $\vdash_\Sigma \mathscr{S}' \rhd \Delta'$ and $\Delta \implies^* \Delta'$ and $\Delta' * \Delta_0$ is consistent.*

Proof (1). The proof is by cases on network reduction rules. The cases of rule INIT and TAU are trivial, since $\Delta = \Delta'$.

Rule IN. By Lemma 2(7),

$$\vdash_\Sigma \mathscr{M}[P] \mid \mathsf{s} : (\mathsf{q}, \mathsf{p}, \ell(v)) \cdot h \rhd \Delta$$

implies $\Sigma = \Sigma_1 \cup \Sigma_2$

$$\Delta = \Delta_1 * \Delta_2 \tag{1}$$
$$\vdash_{\Sigma_1} \mathscr{M}[P] \rhd \Delta_1 \tag{2}$$
$$\vdash_{\Sigma_2} \mathsf{s} : (\mathsf{q}, \mathsf{p}, \ell(v)) \cdot h \rhd \Delta_2 \tag{3}$$

By Lemma 4(1) $\mathscr{M} \xrightarrow{\mathsf{p}?\ell} \mathscr{M}'$ implies $\mathscr{M} = \mathsf{q}?\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I}$ and $\ell = \ell_j$ and $\mathscr{M}' = \mathscr{M}_j$ for some $j \in I$. By Lemma 2(3), the judgment (2) gives $\Sigma_1 = \emptyset$ and

$$\Delta_1 = \{\mathsf{s}[\mathsf{p}] : \mathscr{M}\} \tag{4}$$

and $\vdash P \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}$ and $\mathsf{T} \propto \mathscr{M}$. By Lemma 3(1), the judgment (3) gives $\Sigma_2 = \{\mathsf{s}\}$ and

$$\Delta_2 \approx \{\mathsf{s}[\mathsf{q}] : \langle \mathsf{p}!\ell(S), -\rangle\} \sharp \Delta_3 \tag{5}$$

and

$$\vdash_{\{\mathsf{s}\}} \mathsf{s} : h \rhd \Delta_3 \tag{6}$$

and $\vdash v : S$.

By Lemma 5(1) $P \xrightarrow{\mathsf{s}[\mathsf{p}]?\ell(v)} P'$ and $\vdash P \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}$ imply either $P = \mathsf{s}[\mathsf{p}]?\ell(x).P_0$ and $\mathsf{T} = ?\ell(S').\mathsf{T}'$ or $P = \mathsf{s}[\mathsf{p}]?\ell(x).P_0 + P_1$ and $\mathsf{T} = ?\ell(S').\mathsf{T}' \wedge \mathsf{T}''$. In both cases
$\vdash \mathsf{s}[\mathsf{p}]?\ell(x).P_0 \rhd \mathsf{s}[\mathsf{p}] : ?\ell(S').\mathsf{T}'$ and $P' = P_0\{v/x\}$.
The shapes of $\mathsf{T}, \mathscr{M}$ and $\mathsf{T} \propto \mathscr{M}$ imply $S' = S_j$ and $\mathsf{T}' \propto \mathscr{M}'$. The consistency of $\Delta * \Delta_0$ implies $S = S'$. The we can derive $\vdash P' \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}'$ and

$$\vdash_\emptyset \mathscr{M}'[P'] \rhd \{\mathsf{s}[\mathsf{p}] : \mathscr{M}'\} \tag{7}$$

Applying NPAR to (6) and (7) we derive

$$\vdash_{\{\mathsf{s}\}} \mathscr{M}'[P'] \mid \mathsf{s} : h \rhd \{\mathsf{s}[\mathsf{p}] : \mathscr{M}'\} * \Delta_3$$

Then $\Delta' = \{\mathsf{s}[\mathsf{p}] : \mathscr{M}'\} * \Delta_3$. From (1), (4), and (5) we get

$$\Delta \approx \{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, \mathsf{q}?\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I}, \mathscr{V}\rangle, \mathsf{s}[\mathsf{q}] : \langle \mathsf{p}!\ell(S); \mathsf{m}', \mathscr{V}'\rangle\} \cup \Delta_3' \quad \text{where } \ell = \ell_j \text{ and } S = S_j \text{ and } j \in I$$

for some $\mathsf{m}, \mathscr{V}, \mathsf{m}', \mathscr{V}', \Delta_3'$ such that

$$\Delta_3 = \{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, \mathscr{V}\rangle, \mathsf{s}[\mathsf{q}] : \langle \mathsf{m}', \mathscr{V}'\rangle\} \cup \Delta_3'.$$

Since

$\{\mathsf{s}[\mathsf{q}] : \langle \mathsf{p}!\ell(S); \mathsf{m}', \mathscr{V}'\rangle, \mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, \mathsf{q}?\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I}, \mathscr{V}\rangle\} \implies \{\mathsf{s}[\mathsf{q}] : \langle \mathsf{m}', \mathscr{V}'\rangle, \mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, \mathscr{M}', \mathscr{V}\rangle\}$

then we get $\Delta \implies^* \Delta'$. The only differences between Δ and Δ' are:

- the erasure of the message $\mathsf{p}!\ell(S)$ in the type of $\mathsf{s}[\mathsf{q}]$;
- the replacement of the monitor $\mathscr{M}'$ to the monitor $\mathsf{q}?\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I}$ in the type of $\mathsf{s}[\mathsf{p}]$.

It is then easy to check that the consistency of $\Delta * \Delta_0$ implies the consistency of $\Delta' * \Delta_0$.

Rule ADAINNEW. By Lemma 2(7),

$$\vdash_\Sigma \mathscr{M}[P] \mid \mathsf{s} : (\mathsf{q}, \mathsf{p}, \lambda(\mathsf{G})) \cdot h \rhd \Delta$$

implies $\Sigma = \Sigma_1 \cup \Sigma_2$

$$\Delta = \Delta_1 * \Delta_2 \tag{8}$$
$$\vdash_{\Sigma_1} \mathscr{M}[P] \rhd \Delta_1 \tag{9}$$
$$\vdash_{\Sigma_2} \mathsf{s} : (\mathsf{q}, \mathsf{p}, \lambda(\mathsf{G})) \cdot h \rhd \Delta_2 \tag{10}$$

By Lemma 4(3) $\mathscr{M} \xrightarrow{\mathsf{q}?\lambda}$ implies

$$\mathscr{M} = \mathsf{q}?\{\lambda_i\}_{i\in I} \tag{11}$$

and $\lambda = \lambda_j$ for some $j \in I$. By Lemma 2(3), the judgment (9) gives $\Sigma_1 = \emptyset$ and

$$\Delta_1 = \{\mathsf{s}[\mathsf{p}] : \mathscr{M}\} \tag{12}$$

and $\vdash P \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}$ and $\mathsf{T} \propto \mathscr{M}$. By Lemma 3(2), the judgment (10) gives $\Sigma_2 = \{\mathsf{s}\}$ and

$$\Delta_2 \approx \{\mathsf{s}[\mathsf{q}] : \langle \mathsf{p}!\lambda, -\rangle, \mathsf{s}[\mathsf{p}] : \langle \varepsilon, \mathscr{M}'\rangle\} \sharp \Delta_3 \tag{13}$$

(taking into account that $\mathsf{G}\upharpoonright\mathsf{q} = \mathscr{M}'$) and

$$\vdash_{\{\mathsf{s}\}} \mathsf{s} : h \rhd \Delta_3 \tag{14}$$

We can obtain

$$\vdash_{\emptyset} \mathscr{M}'[Q\{\mathsf{s}[\mathsf{p}]/y\}] \rhd \{\mathsf{s}[\mathsf{p}] : \mathscr{M}'\} \tag{15}$$

by using rule MP, since $(Q, \mathsf{T}') \in \mathscr{P}$, $\mathsf{T}' \propto \mathscr{M}'$ and $\mathscr{M}' \neq \mathsf{end}$ (which is implied by the premise $\mathsf{T} \not\propto \mathscr{M}'$ of the rule ADAINNEW). Hence, we apply rule NPAR to the judgments (14) and (15) and we derive:

$$\vdash_{\{\mathsf{s}\}} \mathscr{M}'[Q\{\mathsf{s}[\mathsf{p}]/y\}] \mid \mathsf{s} : h \rhd \{\mathsf{s}[\mathsf{p}] : \mathscr{M}'\} * \Delta_3$$

Then $\Delta' = \{\mathsf{s}[\mathsf{p}] : \mathscr{M}'\} * \Delta_3$. Notice that (8), (12), (11) and (13) imply

$$\Delta \approx \{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, \mathsf{q}?\{\lambda_i\}_{i\in I}, \mathscr{M}'\rangle\} \cup (\{\mathsf{s}[\mathsf{q}] : \langle \mathsf{p}!\lambda ; \mathsf{m}', -\rangle\} \sharp \Delta_3')$$

for some $\mathsf{m}, \mathsf{m}', \Delta_3'$ such that

$$\Delta_3 \approx \{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, -\rangle, \mathsf{s}[\mathsf{q}] : \langle \mathsf{m}', -\rangle\} \cup \Delta_3'.$$

Note that $\mathsf{s}[\mathsf{p}]$ has $\mathscr{M}'$ as virtual monitor in Δ_2 and then no virtual monitor in Δ_3. Instead $\mathsf{s}[\mathsf{q}]$ has no virtual monitor being the sender of the adaptation. Since

$$\{\mathsf{s}[\mathsf{q}] : \langle \mathsf{p}!\lambda ; \mathsf{m}', -\rangle, \mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, \mathsf{q}?\{\lambda_i\}_{i\in I}, \mathscr{M}'\rangle\} \Longrightarrow \{\mathsf{s}[\mathsf{q}] : \langle \mathsf{m}', -\rangle, \mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, \mathscr{M}', -\rangle\}$$

we get $\Delta \Longrightarrow^* \Delta'$. The only differences between Δ and Δ' are:

- the erasure of the message $\mathsf{p}!\lambda$ in the type of $\mathsf{s}[\mathsf{q}]$;
- the erasure of the monitor $\mathsf{q}?\{\lambda_i\}_{i\in I}$ in the type of $\mathsf{s}[\mathsf{p}]$;
- the monitor $\mathscr{M}'$ is virtual in the type of $\mathsf{s}[\mathsf{p}]$ in Δ and it is active in the type of $\mathsf{s}[\mathsf{p}]$ in Δ' .

It is then easy to check that the consistency of $\Delta * \Delta_0$ implies the consistency of $\Delta' * \Delta_0$.

The proof for rules OUT and ADAINCONT are similar and simpler than those for rules IN and ADAINNEW, respectively.

(2). The proof is by cases on system reduction rules. The case of rule OP is trivial, since $\Delta = \Delta'$.

Rule ADAOUTCONT. Being h λ-free, Δ does not contain virtual monitors. By Lemma 2(9) and (7),
$\vdash_\Sigma \mathscr{M}[P] \mid \mathsf{s} : h \parallel \sigma \rhd \Delta$ implies $\Sigma = \Sigma_1 \cup \Sigma_2$,

$$\Delta = \Delta_1 * \Delta_2 \tag{16}$$

$$\vdash_{\Sigma_1} \mathscr{M}[P] \rhd \Delta_1 \tag{17}$$

$$\vdash_{\Sigma_2} \mathsf{s} : h \rhd \Delta_2 \tag{18}$$

By Lemma 4(4), $\mathscr{M} \xrightarrow{\Pi!\lambda}$ implies

$$\mathscr{M} = \Pi!\{\lambda_i\}_{i\in I} \tag{19}$$

and $\lambda = \lambda_j$ for some $j \in I$. By Lemma 2(3), the judgment (17) gives $\Sigma_1 = \emptyset$ and

$$\Delta_1 = \{\mathsf{s}[\mathsf{p}] : \mathscr{M}\} \tag{20}$$

$$\vdash P \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T} \tag{21}$$

and $\mathsf{T} \propto \mathscr{M}$.

By Lemma 5(4), the judgment (21) and $P \xrightarrow{\mathsf{s}[\mathsf{p}]!(\lambda(F),\mathsf{T}')} P'$ imply $\vdash P' \rhd \mathsf{s}[\mathsf{p}] : \mathsf{T}'$. We consider only the case $\mathscr{M}_\mathsf{p} \neq \mathsf{end}$, the proof for the case $\mathscr{M}_\mathsf{p} = \mathsf{end}$ being similar but for the use of rule $\{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, \mathsf{end}, -\rangle\} \Longrightarrow \{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, -\rangle\}$. Since $\mathsf{T}' \propto \mathscr{M}_\mathsf{p}$, then we derive

$$\vdash_{\emptyset} \mathscr{M}_\mathsf{p}[P'] \rhd \{\mathsf{s}[\mathsf{p}] : \mathscr{M}_\mathsf{p}\} \tag{22}$$

by rule MP. Similarly, for all $\mathsf{q} \in \Pi' \setminus (\Pi \cup \{\mathsf{p}\})$ we can derive

$$\vdash_{\emptyset} \mathscr{M}_\mathsf{q}[P_\mathsf{q}\{\mathsf{s}[\mathsf{q}]/y_\mathsf{q}\}] \rhd \{\mathsf{s}[\mathsf{q}] : \mathscr{M}_\mathsf{q}\} \tag{23}$$

by rule MP, since $(P_\mathsf{q}, \mathsf{T}_\mathsf{q}) \in \mathscr{P}$ and $\mathsf{T}_\mathsf{q} \propto \mathscr{M}_\mathsf{q}$.

By Lemma 2(4), (5) and (6), the judgment (18) gives $\Sigma_2 = \{\mathsf{s}\}$. Rule QSENDG applied to the judgment (18) derives

$$\vdash_{\{\mathsf{s}\}} \mathsf{s} : h \cdot (\mathsf{p}, \Pi, \lambda(\mathsf{G})) \rhd \Delta_2 \sharp \Delta_3 \tag{24}$$

where $\Delta_3 = \{\mathsf{s}[\mathsf{p}] : \langle \Pi!\lambda, -\rangle\} \cup \{\mathsf{s}[\mathsf{q}] : \langle \varepsilon, \mathsf{G}\upharpoonright\mathsf{q}\rangle \mid \mathsf{q} \in \Pi\}$. Notice that $\Delta_2 \sharp \Delta_3$ is defined, since Δ_2 does not contain virtual monitors. Applying rule NPAR to judgments (22), (23) and (24) we conclude

$$\vdash_{\{\mathsf{s}\}} \mathscr{M}_\mathsf{p}[P'] \mid N \mid \mathsf{s} : h \cdot (\mathsf{p}, \Pi, \lambda(\mathsf{G})) \rhd \Delta'$$

where
$N = \prod_{\mathsf{q}\in\Pi'\setminus(\Pi\cup\{\mathsf{p}\})} \mathscr{M}_\mathsf{q}[P_\mathsf{q}\{\mathsf{s}[\mathsf{q}]/y_\mathsf{q}\}]$ and
$\Delta' = \{\mathsf{s}[\mathsf{p}] : \mathscr{M}_\mathsf{p}\} * \{\mathsf{s}[\mathsf{q}] : \mathscr{M}_\mathsf{q} \mid \mathsf{q} \in \Pi' \setminus (\Pi \cup \{\mathsf{p}\})\} * (\Delta_2 \sharp \Delta_3)$.
Notice that (16), (20) and (19) imply

$$\Delta = \{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, \Pi!\{\lambda_i\}_{i\in I}, -\rangle\} * \Delta_2'$$

for some m such that $\Delta_2 \approx \{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, -\rangle\} \sharp \Delta_2'$. Since
$\{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}, \Pi!\{\lambda_i\}_{i\in I}, -\rangle\} \Longrightarrow \{\mathsf{s}[\mathsf{p}] : \langle \mathsf{m}; \overline{\Pi}!\lambda, \mathscr{M}_\mathsf{p}, -\rangle\}$ for $\lambda = \lambda_j$ with $j \in I$
$\emptyset \Longrightarrow \{\mathsf{s}[\mathsf{q}] : \langle \varepsilon, \mathsf{G}\upharpoonright\mathsf{q}\rangle\}$ for $\mathsf{q} \in \Pi$
$\emptyset \Longrightarrow \{\mathsf{s}[\mathsf{q}] : \mathscr{M}_\mathsf{q}\}$ for $\mathsf{q} \in \Pi' \setminus \Pi \cup \{\mathsf{p}\}$
we get $\Delta \Longrightarrow^* \Delta'$. The session typing Δ' contains only monitors which are projections of the global type G. Therefore the consistency of $\Delta * \Delta_0$ implies the the consistency of $\Delta' * \Delta_0$.

The proof for rule ADAOUTNEW proceeds as in the previous case. □

The next lemma shows that typings for systems are invariant under structural equivalence of networks, as expected.

Lemma 7 *If $\vdash_\Sigma N \parallel \sigma \rhd \Delta$ and $N \equiv N'$, then $\vdash_\Sigma N' \parallel \sigma \rhd \Delta$.*

Proof By Lemma 2(9) and rule SYSTEM it is enough to show that $\vdash_\Sigma N \rhd \Delta$ and $N \equiv N'$ imply $\vdash_\Sigma N' \rhd \Delta$. The proof is by induction on the definition of structural equivalence, observing that $\vdash_\emptyset \mathsf{end}[P] \rhd \emptyset$ and using typing rule EQUIV. □

Theorem 1 *(Subject Reduction) If $\vdash_\Sigma \mathscr{S} \rhd \Delta$ with Δ consistent and $\mathscr{S} \longrightarrow^* \mathscr{S}'$, then $\vdash_\Sigma \mathscr{S}' \rhd \Delta'$ for some consistent Δ' such that $\Delta \Longrightarrow^* \Delta'$.*

Proof It is enough to show the statement for the case $\mathscr{S} \equiv \mathscr{E}[N] \parallel \sigma$ and $\mathscr{S}' \equiv \mathscr{E}[N'] \parallel \sigma'$, where either $N \longrightarrow N'$ or $N \parallel \sigma \longrightarrow N' \parallel \sigma'$ by one of the rules considered in Lemma 6. By the structural equivalence on networks we can assume $\mathscr{E} = (\overrightarrow{\nu \mathsf{s}})([\,] \mid N_0)$ without loss of generality. Lemma 7 and Lemma 2(9), (8) and (7) applied to $\vdash_\Sigma \mathscr{S} \rhd \Delta$ give $\vdash_{\Sigma_1} N \rhd \Delta_1$ and $\vdash_{\Sigma_0} N_0 \rhd \Delta_0$, where $\Sigma = (\Sigma_0 \cup \Sigma_1) \setminus \overrightarrow{\mathsf{s}}$ and $\Delta = (\Delta_0 * \Delta_1) \setminus \overrightarrow{\mathsf{s}}$. The consistency of Δ implies the consistency of $\Delta_0 * \Delta_1$ by Lemma 2(8). In the case $N \longrightarrow N'$, by Lemma 6(1) there is Δ_1' such that $\vdash_{\Sigma_1} N' \rhd \Delta_1'$ and $\Delta_1 \Longrightarrow^* \Delta_1'$ and $\Delta_0 * \Delta_1'$ is consistent. In the case $N \parallel \sigma \longrightarrow N' \parallel \sigma'$, by Lemma 6(2) there is Δ_1' such that $\vdash_{\Sigma_1} N' \parallel \sigma' \rhd \Delta_1'$ (that is $\vdash_{\Sigma_1} N' \rhd \Delta_1'$ by Lemma 2(9)) and $\Delta_1 \Longrightarrow^* \Delta_1'$ and $\Delta_0 * \Delta_1'$ is consistent. Therefore, we derive $\vdash_\Sigma \mathscr{S}' \rhd \Delta'$, where $\Delta' = (\Delta_0 * \Delta_1') \setminus \overrightarrow{\mathsf{s}}$ by applying typing rules NPAR, RES, and SYSTEM. Observe that $\Delta \Longrightarrow^* \Delta'$ and Δ' is consistent. □

We say that a system is *initial* when its network is a parallel composition of session initiators, which is always typeable. The type system can guarantee progress, proviso that the collection of processes and types contains at least one process for each monitor which is created at run time in the adaptations. This can also be statically checked when the domains of the adaptation functions which occur in processes are finite. We say that a collection $\mathscr{P}$ is *complete* if, for every global type G in the domain of an adaptation function which occurs in a process belonging to $\mathscr{P}$, there are processes in $\mathscr{P}$ whose types are adequate for the monitors obtained by projecting G onto its participants.

Theorem 2 *(Progress) If $\mathscr{P}$ is complete, $\mathscr{S}$ is an initial system and $\mathscr{S} \longrightarrow^*_{\mathscr{P}} \mathscr{S}'$, then $\mathscr{S}'$ has progress, i.e.*

1. *every input monitored process will eventually receive a message, and*
2. *every message in a queue will eventually be received by an input monitored process.*

Proof Coppo et al. [13] show that an initial system without adaptation flags has progress (by rewriting the result of [13] in our setting).

With respect to the framework of [13], each adaptation step, in our model, can be seen as the starting of a new interaction protocol where all participants can be implemented thanks to the completeness of $\mathscr{P}$. Therefore, the following key features ensure the progress property in our case:

1. all the interaction protocols prescribed by a global type are terminating, either by exchanging adaptation flags or reaching end;
2. a participant is a process, monitored by a projection of a global type, with at most one channel, so there is no communications among participants monitored by different global types, even when they are in the same session;
3. by Subject Reduction, systems are well typed and reduction preserves the consistency of session typings, thus all communications take place in the order prescribed by the global types.

Notice that Point (2) holds thanks to the condition of λ-freeness in rules ADAOUTCONT and ADAOUTNEW. Concluding, the computation can be seen as a succession of independent terminating communication protocols, each of which has the progress property. So the whole computation has it. □

7 Related Work

The literature includes several works aimed at studying adaptive systems in different application contexts and by different perspectives on the conceptual notion of adaptation. The paper [7] provides a valuable discussion on this issue and an interesting classification of various approaches. The state-of-the-art in service choreography adaptation is analysed in [31]. We focus here on the papers which are more related to the distinguishing features of our approach.

Adaptable processes In [5] Bravetti et al. present a calculus in which adaptable processes can be modified by "update patterns". Run-time adaptation of structured communications is approached in [20] by combining the constructors for adaptable processes of [5] with the session type system of [22] for the Boxed Ambient calculus [8]. Session behaviours are never disrupted by adaptation actions, since processes engaged in active sessions cannot be updated. This calculus deals with adaptations of single processes, not with adaptations of the choreography of communicating processes, and only considers dyadic sessions and synchronous communications.

Adaptable choreographies The paper most similar to ours is [1], where global and session types are used to guarantee deadlock-freedom in a calculus of multiparty sessions with asynchronous communications. Only part of the running code is updated. Two different conditions are given for ensuring liveness. The first condition requires that all channel queues are empty before updating. The second condition requires a partial order between the session participants with a unique minimal element. The participants are then updated following this order. Our adaptation framework allows the progress property to be guaranteed without assuming such conditions.

The paper [14], building on [30, 16, 17] and [15], proposes a rule-based approach in which all interactions, under all possible changes produced by the adaptation rules, proceed as prescribed by an abstract model. In particular, the system is deadlock-free by construction. The adaptive system is composed by interacting participants deployed on different locations, each executing its own code. Adaptation

is performed by distributed adaptation servers, which are repositories of adaptation rules. Rules can be added or removed at any moment, while the system is running. Applicability depends on execution environment and properties of the code region to be replaced. If a rule is applied, it replaces part of the code of (some of) the participants with a newer version, able to better meet the requirements. Adaptations of different participants are coordinated ensuring coherent behaviour. Data and control flow statements are done in a Java-style syntax. Central to the technical development are the notions of adaptive interaction oriented choreography and adaptive process oriented choreography, which resemble our global types and monitors. Although there are many analogies between this and our paper, there are important differences. In [14] auxiliary communications are needed to ensure that all participants take the same branch in conditionals, and new participants cannot be added by an adaptation. Moreover in [14] adaptation involves only a part of the choreography and can be applied in any moment, while in our calculus the interaction protocols contain the adaptation points and the reconfiguration step applies to the whole system.

The language of service choreographies defined in [6] is extended with two operators in [4] to take into account adaptation. One operator allows for the specification of adaptable scopes that can be dynamically modified, while the second may dynamically update code in one of such scopes. A similar extension is given for the service contract language of [6], so that projection can relate the two adaptable languages. The main difference with our proposal is that the set of possible participants to the modified choreographies is fixed once for all, in the adaptable scopes.

Monitors In the literature there are many calculi in which the process behaviour is statically and/or dynamically controlled by means of monitors, for example [21,26]. The works that most influenced the present paper are [11,3]. The calculus in those papers is a multiparty session calculus with assertions, and therefore it is much more expressive than our calculus. In fact the monitors in [11,3] prescribe not only the types of the exchanged data, but also that the values of these data satisfy some predicates. Another main difference is that those monitors contain information on the behaviours of all session participants, while our monitors represent the behaviour of single participants.

Intersection and union types In the present paper we type processes with intersections and unions taking inspiration from [33]. The type syntax in that paper is more liberal than ours, for example not requiring labels in an intersection and in a union be different, so more processes can be typed. Also in [33], the most interesting processes correspond to external choices between inputs and internal choices between outputs.

Subtyping for intersections and unions is naturally inspired by their set-theoretical interpretation. Considering the mapping between monitors and types of Definition 8, in this paper we give a subtyping which is the opposite of that considered in [33]. Both subtypings have been largely used [23, 24,37,34,35,18,9,29,32]. The main reason of this difference is that in typing processes one can either *assume* or *derive* the types of channels. In the simple case of a process P with only one channel y the typing judgments have the shapes $y : \mathsf{T} \vdash P$ and $\vdash P \rhd \{y : \mathsf{T}\}$, respectively. This is the reason why subtyping in [23] and in [32] is defined in opposite ways. Choices between fewer inputs are smaller in the subtyping of [23] and bigger in the subtyping of [32]. Choices between outputs behave dually.

8 Conclusion

We have presented a formal model of self-adaptation in multiparty sessions. The framework is based on self-adaptive monitors and global types. In the service-oriented context, global types can be exploited as choreographies of service interactions and monitors as local protocol specifications of services. From this perspective, the present paper provides a formal model for assessing the impact of highly evolving environments, which demand for dynamic self-reconfigurations of the whole system. Interactions among services may be added or removed as well as new services may be required. We use a type discipline to ensure that service collaborations will behave in a safe way after dynamic adaptations.

Differently from approaches focusing on adaptation as code modification in software systems, our approach is choreography centred (similar to [14] for this aspect). When dynamic conditions demand a change, the global choreography updates itself together with the new monitors which prescribe the new behaviours to the participants. A process fills (implements) a given monitor if its type is adequate for that monitor, otherwise a different implementation (process) need to be found. Notably, we proved that all monitored processes behave correctly and interact with each other in a safe way, once the adaptation has been performed.

As a main feature, we achieve a decentralised control of the adaptation and a notable flexibility in the dynamics of self-reconfiguration. According to its monitor, any participant can be in charge of checking global data and sending the adaptation request, instead of devoting a centralised mechanism to this task. Furthermore, the dynamic reconfiguration can add new participants, while some of the old participants are not longer involved. Finally, processes, that are simply implementation code, can follow different incompatible computational paths, thus each participant can be differently implemented in the various adaptation steps.

One apparent limitation of our calculus is that processes can only operate on a single channel. This limitation can

be addressed by extending the process language and its typing rules, without major consequences on the rest of the development. Instead, adding session delegation would require further investigation.

We plan to experiment with implementations of our approach, to evaluate its feasibility.

We are working toward a quantitative version of our model, where the global state also contains dynamically evolving semantic information about processes, such as reputation or performance rates. Using this information, adaptation functions will be able to choose a single process among all the processes matching a monitor, as one of the best implementations for that participant. In the present calculus, this issue results in an arbitrary choice, since processes can be taken solely on the basis of their compatibility with monitors from the point of view of safe adaptations. In a realistic application, instead, it would be interesting to involve other requests concerning quantitative aspects.

Acknowledgements The authors gratefully thank the anonymous referees for their accurate and enlightening remarks, that strongly improved both the presentation and the technical development.

This work was partially supported by EU Collaborative project ASCENS 257414, ICT COST Action IC1201 BETTY, MIUR PRIN Project CINA Prot. 2010LHT4KM and Torino University/Compagnia San Paolo Project SALT.

References

1. Anderson, G., Rathke, J.: Dynamic Software Update for Message Passing Programs. In: APLAS'12, *LNCS*, vol. 7705, pp. 207–222. Springer (2012)
2. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global Progress in Dynamically Interleaved Multiparty Sessions. In: CONCUR'08, *LNCS*, vol. 5201, pp. 418–433. Springer (2008)
3. Bocchi, L., Chen, T.C., Demangeon, R., Honda, K., Yoshida, N.: Monitoring Networks through Multiparty Session Types. In: FMOODS/FORTE'13, *LNCS*, vol. 7892, pp. 50–65. Springer (2013)
4. Bravetti, M., Carbone, M., Hildebrandt, T., Lanese, I., Mauro, J., Perez, J.A., Zavattaro, G.: Towards Global and Local Types for Adaptation. In: SEFM'13, *LNCS*, vol. 8368, pp. 3–14. Springer (2014)
5. Bravetti, M., Di Giusto, C., Pérez, J.A., Zavattaro, G.: Adaptable Processes. Logical Methods in Computer Science **8**(4) (2012)
6. Bravetti, M., Zavattaro, G.: Towards a Unifying Theory for Choreography Conformance and Contract Compliance. In: SC'07, *LNCS*, vol. 4829, pp. 34–50. Springer (2007)
7. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: A Conceptual Framework for Adaptation. In: FASE'12, *LNCS*, vol. 7212, pp. 240–254. Springer (2012)
8. Bugliesi, M., Castagna, G., Crafa, S.: Access Control for Mobile Agents: The Calculus of Boxed Ambients. ACM Transactions on Programming Languages and Systems **26**(1), 57–124 (2004)
9. Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centered Programming for Web Services. ACM Transactions on Programming Languages and Systems **34**(2), 8:1–8:78 (2012)
10. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On Global Types and Multi-Party Sessions. Logical Methods in Computer Science **8**, 1–45 (2012)
11. Chen, T.C., Bocchi, L., Deniélou, P.M., Honda, K., Yoshida, N.: Asynchronous Distributed Monitoring for Multiparty Session Enforcement. In: TGC'11, *LNCS*, vol. 7173, pp. 25–45. Springer (2012)
12. Coppo, M., Dezani-Ciancaglini, M., Venneri, B.: Self-Adaptive Monitors for Multiparty Sessions. In: PDP'14, pp. 688–696. IEEE (2014)
13. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global Progress for Dynamically Interleaved Multiparty Sessions. Mathematical Structures in Computer Science (2013). To appear
14. Dalla Preda, M., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J.: Deadlock Freedom by Construction for Distributed Adaptative Applications. CoRR (2014). URL `http://arxiv.org/abs/1407.0970`
15. Dalla Preda, M., Giallorenzo, S., Lanese, I., Mauro, J., Gabbrielli, M.: AIOCJ: A Choreographic Framework for Safe Adaptive Distributed Applications. In: SLE'14, *LNCS*, vol. 8706, pp. 161–170. Springer (2014)
16. Dalla Preda, M., Lanese, I., Mauro, J., Gabbrielli, M.: Adaptive Choreographies (2013). URL `http://www.cs.unibo.it/~lanese/publications/adaptchor.pdf.gz`
17. Dalla Preda, M., Lanese, I., Mauro, J., Gabbrielli, M., Giallorenzo, S.: Safe Run-time Adaptation of Distributed Systems (2013). URL `http://www.cs.unibo.it/~lanese/publications/fulltext/safeadapt.pdf.gz`
18. Demangeon, R., Honda, K.: Full Abstraction in a Subtyped pi-Calculus with Linear Types. In: CONCUR'11, *LNCS*, vol. 6901, pp. 280–296. Springer (2011)
19. Deniélou, P.M., Yoshida, N.: Dynamic Multirole Session Types. In: POPL'11, pp. 435–446. ACM Press (2011)
20. Di Giusto, C., Pérez, J.A.: Disciplined Structured Communications with Consistent Runtime Adaptation. In: SAC'13, pp. 1913–1918. ACM Press (2013)
21. Ferrari, G.L., Moggi, E., Pugliese, R.: Guardians for Ambient-based Monitoring. ENTCS **66**(3), 52–75 (2002)
22. Garralda, P., Compagnoni, A.B., Dezani-Ciancaglini, M.: BASS: Boxed Ambients with Safe Sessions. In: PPDP'06, pp. 61–72. ACM Press (2006)
23. Gay, S., Hole, M.: Subtyping for Session Types in the Pi Calculus. Acta Informatica **42**(2/3), 191–225 (2005)
24. Gay, S.J.: Bounded Polymorphism in Session Types. Mathematical Structures in Computer Science **18**(5), 895–930 (2008)
25. Ghezzi, C., Pradella, M., Salvaneschi, G.: An Evaluation of the Adaptation Capabilities in Programming Languages. In: SEAMS'11, pp. 50–59. ACM Press (2011)
26. Gorla, D., Hennessy, M., Sassone, V.: Security Policies as Membranes in Systems for Global Computing. ENTCS **138**(1), 23–42 (2005)
27. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Disciplines for Structured Communication-based Programming. In: ESOP'98, *LNCS*, vol. 1381, pp. 22–138. Springer (1998)
28. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL'08, pp. 273–284. ACM Press (2008)
29. Kouzapas, D., Yoshida, N., Honda, K.: On Asynchronous Session Semantics. In: FMOODS/FORTE'11, *LNCS*, vol. 6722, pp. 228–243. Springer (2011)
30. Lanese, I., Bucchiarone, A., Montesi, F.: A Framework for Rule-Based Dynamic Adaptation. In: TGC'10, *LNCS*, vol. 6084, pp. 284–300. Springer (2010)
31. Leite, L.A.F., Oliva, G.A., Nogueira, G.M., Gerosa, M.A., Kon, F., Milojicic, D.S.: A Systematic Literature Review of Service Choreography Adaptation. Service Oriented Computing and Applications **7**(3), 199–216 (2013)

32. Mostrous, D., Yoshida, N., Honda, K.: Global Principal Typing in Partially Commutative Asynchronous Sessions. In: ESOP'09, *LNCS*, vol. 5502, pp. 316–332. Springer (2009)
33. Padovani, L.: Session Types = Intersection Types + Union Types. In: ITRS'10, *EPTCS*, vol. 45, pp. 71–89 (2010)
34. Padovani, L.: Fair Subtyping for Multi-party Session Types. In: COORDINATION'11, *LNCS*, vol. 6721, pp. 127–141. Springer (2011)
35. Padovani, L.: Fair Subtyping for Open Session Types. In: ICALP'13, *LNCS*, vol. 7966, pp. 373–384. Springer (2013)
36. Psaier, H., Juszczyk, L., Skopik, F., Schall, D., Dustdar, S.: Runtime Behavior Monitoring and Self-Adaptation in Service-Oriented Systems. In: SASO'10, pp. 164–173. IEEE Computer Society (2010)
37. Vasconcelos, V.T.: Fundamentals of Session Types. In: SFM'09, *LNCS*, vol. 5569, pp. 158–186. Springer (2009)