Self-Adaptive Monitors for Multiparty Sessions

Self-Adaptive Monitors for Multiparty Sessions

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

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

***Abstract*—This paper aims at incorporating the notion of *self-adaptiveness* in the context of *multiparty sessions*, by focusing on the issue of ensuring correctness for dynamic adaptations. A formal framework is presented centred around these main ingredients: *global types*, *monitors* and *global state*. A global type represents the overall communication choreography. Its projections are the monitors, which set-up the protocols of the participants. The association of a monitor with a compliant process incarnates a single participant. It is the choreography that is updated at runtime, in response to changing conditions in the global state. Monitors result to be self-adaptive in the sense that they react to these changes by modifying themselves, in order to prescribe new behaviours to the participants.**

I. Introduction

The growing complexity of software systems, intended to operate in many different scenarios and in highly dynamic environments, brought out the necessity of managing this complexity at a reasonable cost. This leads naturally to develop models of *self-adaptive systems*, that are able to modify their behaviours autonomously and dynamically, in response to changing conditions in the execution environment and in accordance with evolving policies and objectives. Therefore (self-)adaptive systems came out as a key research subject in many fields of Computer Science (e.g., Distributed Systems, Service-Oriented Architectures, etc..), taking adaptivity metaphors from natural and social sciences, as for example in [1].
Most of the approaches in the literature, however, 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 [2] and web services testbed in [3]).

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, rather deep, characterization is the one presented in [4]: *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 [4] 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.

The framework is that of *multiparty sessions* [5], where each participant can access and modify the *global state* representing those (control) data whose values are critical for planning the adaptation steps. The system comprises four active parties: *global types*, *monitors*, *processes*, and *adaptation functions*.

A global type represents the overall communication choreography [6]; its projections onto participants generate the monitors, which are essentially local types and set-up the 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 to the monitoring protocol. Notably, we exploit intersection types, union types and subtyping to make flexible this compliance relation. 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. The adaptation functions contain the dynamic evolution policy, since they say *how* the system needs to reconfigure itself based on the changes of the critical data.

Therefore, dynamic adaptations are triggered by control data and monitors.

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 unforeseen events, which require the community to modify its plan dynamically,
• those critical events are observed in any separate component of the system, which 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 not longer involved (temporarily or permanently),
• these dynamic changes need to be safe: community interactions must proceed correctly to pursue the common task.

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 then produces a bunch of products for nearby markets or for export. However, no plant has all possible types of machinery, nor produces each possible product. Local sellers sell products from near or far away plants. 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 function, which contains 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 a whole plant, can require the company organisation to update itself: new production and sale plans have to be adopted to maintain uninterrupted supply to customers.

Example In order to give a preliminary intuition of our system, 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). We use a simplified and incomplete syntax (w.r.t. the formal presentation of next section). Then we show how self-adaptation works when a fire incapacitates a factory. The global state contains for each of the two plants either OK or KO. When both plants are OK the global type is:

$$G_1 = \begin{cases} \mathsf{iS} \to \mathsf{iF} : (\text{item}, \text{amount}). \\ \mathsf{aS} \to \mathsf{aF} : (\text{item}, \text{amount}). \\ \mathsf{Ada} \to \{\mathsf{iS}, \mathsf{iF}, \mathsf{aS}, \mathsf{aF}\} : \mathit{check} \end{cases}$$

Each seller requires to the corresponding factory some products and 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:

$$G_2 = \begin{cases} \mathsf{Ada} \to \mathsf{Bob} : \text{contract}. \\ \mathsf{iS} \to \mathsf{iF} : (\text{item}, \text{amount}). \\ \mathsf{aS} \to \mathsf{Ada} : (\text{item}, \text{amount}). \\ \mathsf{Ada} \to \{\mathsf{iS}, \mathsf{iF}, \mathsf{aS}, \mathsf{Bob}\} : \mathit{check} \end{cases}$$

where Bob is in charge of rebuilding factories. In the symmetric case the global type G_3 is as expected. Finally, when both the factories are KO, Ada just closes down the business:

$$G_4 = \mathsf{Ada} \to \{\mathsf{iS}, \mathsf{aS}\} : \text{bye.end}$$

The processes in Table I are correct implementations for the monitors generated by projection from all the above global types. For instance, the monitor of Ada for G_2 is $\mathsf{Bob}!\text{Contract}.\mathsf{aS}?(\text{Item}, \text{Amount}).\{\mathsf{iS}, \mathsf{iF}, \mathsf{aS}, \mathsf{Bob}\}!\mathit{check}$, where ! represents output, ? represents input, Contract denotes the type of contract, etc. The Seller has two alternative behaviours. He can send on channel y (item, 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 uncapacitated, and by Bob, writing OK when he accomplished the rebuilding task. The adaptation function F in Ada code gives the new global type when applied to the pair (*state* iF, *state* aF), i.e.

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

TABLE I. PROCESSES FOR THE COMPANY EXAMPLE

$$\begin{array}{ll} F(\mathsf{OK}, \mathsf{OK}) = G_1 & F(\mathsf{OK}, \mathsf{KO}) = G_2 \\ F(\mathsf{KO}, \mathsf{OK}) = G_3 & F(\mathsf{KO}, \mathsf{KO}) = G_4 \end{array}$$

Notice that the senders and the receivers are omitted in processes and this allows a process to fill several different monitors. For instance, the process Seller can fill both 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 (OK, 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 chief. When process Bob writes OK for the American factory and the Italian factory is still OK, the adaptation reverses to the global type G_1. Then the American factory comes back into the scene.

Structure of the paper Sections II and III present the syntax of our calculus and the type system, respectively. The dynamic evolution is given in Section IV together with the main properties of the whole system. Related works are discussed in Section V.

II. A SELF-ADAPTIVE SYSTEM

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 [7] 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.

$$S ::= \mathsf{bool} \ \mid \ \mathsf{nat} \ \mid \ \ldots.$$

Definition 2.1: Global types are defined by:

$$\begin{array}{lcll} G & ::= & \mathsf{p} \to \Pi : \{\ell_i(S_i).G_i\}_{i \in I} & \mid \\ & & \mathsf{p} \to \Pi : \{\lambda_i\}_{i \in I} & \mid \quad \mathsf{end} \end{array}$$

In writing $\{\ell_i(S_i).G_i\}_{i \in I}$ 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

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

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

TABLE II. PROJECTION OF A GLOBAL TYPE ONTO A PARTICIPANT

$\mathsf{pa}(G)$ the set of all participants in G.

Global types can end 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. Note however that, given a global type, we can always insert adaptation flags in some points preserving also the original interaction protocol.

There is no recursion operator, but recursive protocols can be obtained through adaptation.

Notably we do not allow parallel composition of global types, which is quite common in the literature [5], [6], [8], [9]. As a matter of fact many papers [5], [6], [8] 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, that is orthogonal to the present development [9].

Monitors Monitors can be viewed as local types that are obtained as projections from global types onto individual participants, as in the standard approach of [5] and [10]. 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.

Definition 2.2: The set of *monitors* is defined by:

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

The constructs in the first line correspond to output and input actions. 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 type S_i, labelled 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, with an internal choice, can send, for each $i \in I$, a value of type S_i, distinguished by the label ℓ_i, and then continues as prescribed by $\mathscr{M}_i$. The constructs in the second line similarly fit with the sending and receiving of adaptation flags. The monitor end fits with all processes.

The projection of global types onto participants is given in Table II. A projection is undefined when two participants not involved in a choice have different projections in different branchings (condition $G_i \upharpoonright \mathsf{q} = G_j \upharpoonright \mathsf{q}$ 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 $\Pi \cup \{\mathsf{p}\} = \mathsf{pa}(G)$. I.e. all participants are involved in flag exchanges. 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 [11], [12], [13], [5], [10], [14], [7], [15], [16], [8], processes do not contain 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 and 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 has a unique channel. We convene to 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 the unique 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 2.3: Processes are defined by:

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

The syntax of processes is rather standard, but note that in the sending and receiving actions the involved participant 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 2.5) and the op operator represents an action on this global state, like for instance a "read" or "write" operation. We leave unspecified the kind of actions since we are interested only 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 III. Process types are mainly aimed at checking the matching between processes and monitors. Indeed it is convenient to include a type annotation in the syntax of the adaptation flag. The input flag $\mathsf{c}?(\lambda,\mathsf{T}).P$

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

TABLE III. THE MAPPINGS *lin* AND *lout*.

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 generate the new global type, which provides a new choreography for the system reconfiguration.

Network The sessions are initiated by the new constructor applied to global types (*session initiator*), denoted by $\mathsf{new}(G)$. In carrying on a multiparty interaction a process is always controlled by a monitor, which assures 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 some processes can 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 $\emptyset$. 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 $(\mathsf{p},\Pi,\lambda(G))$, indicating that the flag λ and the global type G are sent by participant p to all participants in Π. Queue concatenation, denoted by "$\cdot$", has $\emptyset$ as neutral element. A queue is λ*-free* if it contains no flag.

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 2.4: Networks are defined by:

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

System A system includes a network, a global state and a collection of processes together with their types (according to the typing rules of Section III). We use σ to range over global states and we denote by $\mathscr{P}$ the collection of processes with types. Since only networks and global states change at run time, we represent systems as their composition (via "$\|$"), without mentioning the process collection.

Definition 2.5: Systems are defined by:

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

III. PROCESS TYPES

Process types (called simply *types* where not ambiguous) describe process communication behaviours [11]. They have prefixes corresponding to sending and receiving of labels and flags (*input* and *output types*). 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 an union type. We do not allow intersection between input types with the same label, since we want choices to be unambiguous: the types following a same input prefix could be different and this would lead to a communication mismatch, as illustrated in Example 4.1. For the same reason we do not allow intersections between output types with the same label. Since we want to match types with monitors where internal choices are always taken by participants sending a label or a flag, we force unions to take as arguments output types (possibly combined by intersections or unions). We start with the more liberal syntax of pre-types. We formalise the above restrictions by means of two mappings from pre-types to sets of labels and flags (Table III) and then we define types.

Definition 3.1: 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.

Definition 3.2: A *type* is a pre-type satisfying the following constraints:

- 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. 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.

Typing rules are given in Table IV. We assume that expressions are typed by sorts, as usual. 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. 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 assures 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, $y!check(F)$ is replaced by $y!(check(F),\mathsf{T}_{\mathsf{Ada}})$, where $\mathsf{T}_{\mathsf{Ada}}$ is the type of the whole process Ada:

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

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

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

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

TABLE IV. TYPING RULES FOR PROCESSES

$$(!check) \wedge (!\text{Contract}.?(\text{Item}, \text{Amount}).!check) \wedge (!\text{bye}.\text{end})$$

The matching between process types and monitors (adequacy) is made rather flexible by using the *subtyping* relation defined in Table V. The intuitive meaning of subtyping is that a process with a smaller type has all the behaviours required by a bigger type and more. Therefore, end is the top type. Subtyping is monotone, for input/output prefixes, with respect to continuations and it follows the usual set theoretic inclusion of intersection and union.

We can exploit standard distributivity laws for intersection and union types, in order to verify the decidability of subtyping. By distributivity, any subtyping relation can be reduced to a set of subtyping relations, where we have only intersections on the left and only unions on the right. Moreover, since end is the top type, if end occurs in an intersection it can be erased, if end is the type on the left then the subtyping fails and, conversely, if end is the type on the right then the subtyping holds. Furthermore, in our context unions of input types are not types, then we have to only deal with subtyping relations in which the right side is either a union of output types or a single input type. Thus it is easy to prove that subtyping is decidable.

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.

Definition 3.3: 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:

$$|\mathsf{p}?\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I}| = \bigwedge_{i\in I} ?\ell_i(S_i).|\mathscr{M}_i|$$
$$|\Pi!\{\ell_i(S_i).\mathscr{M}_i\}_{i\in I}| = \bigvee_{i\in I} !\ell_i(S_i).|\mathscr{M}_i|$$
$$|\mathsf{p}?\{\lambda_i\}_{i\in I}| = \bigwedge_{i\in I} ?\lambda_i \qquad |\Pi!\{\lambda_i\}_{i\in I}| = \bigvee_{i\in I} !\lambda_i$$
$$|\mathsf{end}| = \mathsf{end}$$

For instance, the type $\mathsf{T}_{\mathsf{Ada}}$ defined above is adequate for the monitor of Ada discussed in the Introduction.

Adequacy is clearly decidable, being subtyping decidable.

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

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

where $=$ stands for $\leq$ and $\geq$.

TABLE V. SUBTYPING

IV. SAFE ADAPTATIONS AND COMMUNICATIONS

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 (though the new 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:

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

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 VI, where the treatment of recursions and conditionals is standard. In the rules for external choice we convene that 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 we omit the symmetric rules. 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. For example, $\mathsf{op}.\mathsf{s}[\mathsf{p}]?\ell_1(x).P_1 + \mathsf{s}[\mathsf{p}]?\ell_2(x).P_2 \xrightarrow{\mathsf{op}} \mathsf{s}[\mathsf{p}]?\ell_1(x).P_1 + \mathsf{s}[\mathsf{p}]?\ell_2(x).P_2 \xrightarrow{\mathsf{s}[\mathsf{p}]?\ell_2(v)} P_2$.

We assume a standard structural equivalence on networks, in which the parallel operator is commutative and associative. The structural equivalence erases any monitored process with end monitor, since it is idle:

$$\mathsf{op}.P \xrightarrow{\mathsf{op}} P \qquad \mu X.P \xrightarrow{\tau} P\{\mu X.P/X\}$$

$$\mathsf{s}[\mathsf{p}]?\ell(x).P \xrightarrow{\mathsf{s}[\mathsf{p}]?\ell(v)} P\{v/x\} \quad \mathsf{s}[\mathsf{p}]!\ell(\mathsf{e}).P \xrightarrow{\mathsf{s}[\mathsf{p}]!\ell(v)} P \quad \mathsf{e}\downarrow v$$

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

$$\mathsf{if}\ \ \mathsf{e}\ \ \mathsf{then}\ \ P\ \ \mathsf{else}\ Q \xrightarrow{\tau} P \quad \mathsf{e}\downarrow \mathsf{true},$$

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

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

TABLE VI. LTS OF PROCESSES

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

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

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

TABLE VII. EQUIVALENCE ON MESSAGE QUEUES

$$\mathsf{end}[P] \mid N \ \equiv\ N$$

For message queues, we need an equivalence for commuting independent messages and another one for splitting a message to multiple receivers, see Table VII. 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 VIII lists the reduction rules of the networks and Table IX 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 $\mathsf{pa}(G)$ of the participants in the global type G we need to find a process P_{p} with a type T_{p} in the collection $\mathscr{P}$ such that T_{p} is adequate for 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 assure the privacy of the communications in a session (as standard in session calculi [5]). 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 the processes which are associated to monitors. So we have left this choice arbitrary, putting as only condition 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 though queues. The type assignment system assures both that the type of P is adequate for $\mathscr{M}$ and that the type of P' is adequate for $\mathscr{M}'$. Following [15], [8], the agreement between monitors and processes is required; the novelty is that only the monitors define the senders and the receivers of messages.

The rules ADAINCONT and ADAINNEW of Table VIII 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 $\mathscr{P}$ a different process with a type adequate for $\mathscr{M}'$.

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 IX. Rule CXT is a standard contextual rule. Rule OP allows processes to read/modify the global state.

The more 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 $G\upharpoonright\mathsf{p}$ will be associated or not to the current process according to whether its type is adequate or not for $G\upharpoonright\mathsf{p}$, 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 assured 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 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 corresponding 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 American 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 this restriction allows some participants to finish their communications before performing an adaptation, while other participants have already self-adapted and then started the new communications.

We use $\longrightarrow^*$ with the usual meaning and we write $\longrightarrow_{\mathscr{P}}$, $\longrightarrow^*_{\mathscr{P}}$ when we want emphasise the use of $\mathscr{P}$ in rules INIT, ADAINNEW, ADAOUTNEW.

Example 4.1: We can show now the necessity of the conditions on local types given in Definition 3.2. For readability we omit $\{,\}$ in writing global types and monitors. Take the global type $G = 1 \to 2 : \ell(\mathsf{int}).2 \to 1 : \ell'(\mathsf{int}).\mathsf{end}$, whose projections on participants 1 and 2 are $\mathscr{M}_1 = 2!\ell(\mathsf{int}).2?\ell'(\mathsf{int}).\mathsf{end}$ and $\mathscr{M}_2 = 1?\ell(\mathsf{int}).1!\ell'(\mathsf{int}).\mathsf{end}$, respectively. Without the conditions of Definition 3.2, $\mathscr{P}$ could contain (P_1, T_1) and (P_2, T_2) where $P_1 = y!\ell(3).y?\ell'(x).\mathbf{0} + y!\ell(\mathsf{true}).\mathbf{0}$, $P_2 = y?\ell(x').y!\ell'(-x').\mathbf{0}$, $T_1 = !\ell(\mathsf{int}).?\ell'(\mathsf{int}).\mathsf{end}\wedge!\ell(\mathsf{bool}).\mathsf{end}$, and $\mathsf{T}_2 = ?\ell(\mathsf{int}).!\ell'(\mathsf{int}).\mathsf{end}$. The pre-type T_1 is not a type, since the intersection is between output types with the same label. Notice that T_1 and T_2 are adequate for $\mathscr{M}_1$ and $\mathscr{M}_2$, respectively. It is easy to verify that the network $\mathsf{new}(G)$ 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

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

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

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

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

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

TABLE VIII. NETWORK REDUCTION

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

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

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

TABLE IX. SYSTEM REDUCTION

T_1' and T_2 are adequate for $\mathscr{M}_1$ and $\mathscr{M}_2$, but the network $\mathsf{new}(G)$ smoothly terminates the computation. Note that T_1' is a type since it satisfies Definition 3.2, so there is no possible ambiguity on which branch of the external choice must be chosen.

In order to prove the correctness of our framework, we need to assign types to systems. 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 which occur free in the network) and Δ is a *session typing*. Session typings associate session channels to suitable types, that generalise process types. We introduce the notion of *consistency* of session typings, to assure that each pair of participants in a conversation performs mutual communications in a dual way. Since session typings represent the forthcoming communications, also typings can change by reducing systems. This can be formalised as in [5] by introducing the reduction of session typings, denoted by $\Longrightarrow$. We can state the subject reduction property of systems under the condition that they are typed by consistent session typings.

Theorem 4.2 (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 $\Delta \Longrightarrow^* \Delta'$.

As an immediate consequence of subject reduction we get *communication safety* according to [5], i.e. systems "never go wrong" since every delivered value matches the receiving monitored process and channel linearity is preserved.

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 4.3 (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 always (eventually) receive a message, and
2) every message in a queue will always (eventually) be received by an input monitored process.

The proof of progress relies on the observation that an initial system is well typed, so by subject reduction $\mathscr{S}'$ is well typed too. The completeness of $\mathscr{P}$ assures that all session participants are present also after an adaptation, and the consistency of session typings guarantees that communications are dual. Duality ensures that each message in the queue will have a receiver and each input will find a corresponding message in the queue.

V. 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 [4] provides a valuable discussion on this issue and an interesting classification of various approaches. For space reasons we focus here on the papers which are more related to the distinguishing features of our approach.

Adaptable processes In [17] 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 [18] by combining the constructors for adaptable processes of [17] with the session type system of [19] for the Boxed Ambient calculus [20]. Session behaviours are never disrupted by adaption 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. Dyadic sessions and synchronous communications are further differences with our calculus.

Adaptable choreographies The paper more similar to our is [21], 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 assuring 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 flags allow the progress property to be guaranteed without assuming such conditions.

The paper [22], building on [23] and [24], 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 [22] auxiliary communications are needed to assure that all participants take the same branch in conditionals, and new participants cannot be added by an adaptation. Moreover in [22] 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.

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 [25], [26]. The works that more influenced the present paper are [15], [8]. The calculus in those papers is a multi-party session calculus with assertions, and therefore it is much more expressive than our calculus. In fact the monitors in [15], [8] 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 [14]. The type syntax in that paper is more liberal than our, for example not requiring that labels in an intersection and in an union be different, so more processes can be typed. Anyway also in [14] the more interesting processes are 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 3.3, in this paper we give a subtyping for branchings and selections which is the opposite of that considered in [14]. Both subtypings have been largely used [12], [13], [16], [27], [28], [29], [6], [30], [31]. 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 $\vdash P \triangleright \{y : \mathsf{T}\}$, respectively. This is the reason why subtyping in [12] and in [31] is defined in opposite ways. Branching with less choices are smaller in the subtyping of [12] and bigger in the subtyping of [31]. Selections behave dually.

VI. Conclusion

We have presented a formal model of self-adaptation in multiparty sessions. The framework is based on self-adaptive monitors and global types.

Differently from approaches focusing on adaptation as code modification in software systems, our approach is choreography-centred (similar to [22] 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. Then, once the adaptation has been performed, all monitored processes behave correctly and interact with each other in a safe way.

As a main feature, we achieve a decentralised control of the adaptation and a notable flexibility in the dynamic system 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 system 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 are single-threaded, have no delimitations and can only operate on a single channel/session. This limitation can be addressed by extending the process language and its typing rules, without major consequences on the rest of the development.

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 useful remarks. 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] F. Zambonelli and M. Viroli, "From Service-Oriented Architectures to Nature-Inspired Pervasive Service Ecosystems," in *WOA'10*, ser. CEUR Workshop Proceedings, vol. 621. CEUR-WS.org, 2010.

[2] C. Ghezzi, M. Pradella, and G. Salvaneschi, "An Evaluation of the Adaptation Capabilities in Programming Languages," in *SEAMS'11*. ACM Press, 2011, pp. 50–59.

[3] H. Psaier, L. Juszczyk, F. Skopik, D. Schall, and S. Dustdar, "Run-time Behavior Monitoring and Self-Adaptation in Service-Oriented Systems," in *SASO'10*. IEEE Computer Society, 2010, pp. 164–173.

[4] R. Bruni, A. Corradini, F. Gadducci, A. Lluch-Lafuente, and A. Vandin, "A Conceptual Framework for Adaptation," in *FASE'12*, ser. LNCS, vol. 7212. Springer, 2012, pp. 240–254.

[5] K. Honda, N. Yoshida, and M. Carbone, "Multiparty Asynchronous Session Types," in *POPL'08*. ACM Press, 2008, pp. 273–284.

[6] M. Carbone, K. Honda, and N. Yoshida, "Structured Communication-Centered Programming for Web Services," *ACM Trans. Program. Lang. Syst.*, vol. 34, no. 2, pp. 8:1–8:78, Jun. 2012.

[7] P.-M. Deniélou and N. Yoshida, "Dynamic Multirole Session Types," in *POPL'11*. ACM Press, 2011, pp. 435–446.

[8] L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, and N. Yoshida, "Monitoring Networks through Multiparty Session Types," in *FMOODS/FORTE'13*, ser. LNCS, vol. 7892. Springer, 2013, pp. 50–65.

[9] G. Castagna, M. Dezani-Ciancaglini, and L. Padovani, "On Global Types and Multi-Party Sessions," *Log. Meth. Comp. Scie.*, vol. 8, pp. 1–45, 2012.

[10] L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida, "Global Progress in Dynamically Interleaved Multiparty Sessions," in *CONCUR'08*, ser. LNCS, vol. 5201. Springer, 2008, pp. 418–433.

[11] K. Honda, V. T. Vasconcelos, and M. Kubo, "Language Primitives and Type Disciplines for Structured Communication-based Programming," in *ESOP'98*, ser. LNCS, vol. 1381. Springer, 1998, pp. 22–138.

[12] S. Gay and M. Hole, "Subtyping for Session Types in the Pi Calculus," *Acta Inf.*, vol. 42, no. 2/3, pp. 191–225, 2005.

[13] S. J. Gay, "Bounded Polymorphism in Session Types," *Math. Struct. in Comp. Science*, vol. 18, no. 5, pp. 895–930, 2008.

[14] L. Padovani, "Session Types = Intersection Types + Union Types," in *ITRS'10*, ser. EPTCS, vol. 45, 2010, pp. 71–89.

[15] T.-C. Chen, L. Bocchi, P.-M. Deniélou, K. Honda, and N. Yoshida, "Asynchronous Distributed Monitoring for Multiparty Session Enforcement," in *TGC'11*, ser. LNCS, vol. 7173. Springer, 2012, pp. 25–45.

[16] L. Padovani, "Fair Subtyping for Multi-party Session Types," in *COORDINATION'11*, ser. LNCS, vol. 6721. Springer, 2011, pp. 127–141.

[17] M. Bravetti, C. Di Giusto, J. A. Pérez, and G. Zavattaro, "Adaptable Processes," *Log. Meth. Comp. Scie.*, vol. 8, no. 4, 2012.

[18] C. Di Giusto and J. A. Pérez, "Disciplined Structured Communications with Consistent Runtime Adaptation," in *SAC'13*. ACM Press, 2013, pp. 1913–1918.

[19] P. Garralda, A. B. Compagnoni, and M. Dezani-Ciancaglini, "BASS: Boxed Ambients with Safe Sessions," in *PPDP'06*. ACM Press, 2006, pp. 61–72.

[20] M. Bugliesi, G. Castagna, and S. Crafa, "Access Control for Mobile Agents: The Calculus of Boxed Ambients," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 1, pp. 57–124, 2004.

[21] G. Anderson and J. Rathke, "Dynamic Software Update for Message Passing Programs," in *APLAS'12*, ser. LNCS, vol. 7705. Springer, 2012, pp. 207–222.

[22] M. dalla Preda, I. Lanese, J. Mauro, M. Gabbrielli, and S. Giallorenzo. (2013) Safe Run-time Adaptation of Distributed Systems. [Online]. Available: http://www.cs.unibo.it/~lanese/publications/fulltext/safeadapt.pdf.gz

[23] I. Lanese, A. Bucchiarone, and F. Montesi, "A Framework for Rule-Based Dynamic Adaptation," in *TGC'10*, ser. LNCS, vol. 6084. Springer, 2010, pp. 284–300.

[24] M. dalla Preda, I. Lanese, J. Mauro, and M. Gabbrielli. (2013) Adaptive Choreographies. [Online]. Available: http://www.cs.unibo.it/~lanese/publications/adaptchor.pdf.gz

[25] G. L. Ferrari, E. Moggi, and R. Pugliese, "Guardians for Ambient-based Monitoring," *ENTCS*, vol. 66, no. 3, pp. 52–75, 2002.

[26] D. Gorla, M. Hennessy, and V. Sassone, "Security Policies as Membranes in Systems for Global Computing," *ENTCS*, vol. 138, no. 1, pp. 23–42, 2005.

[27] V. T. Vasconcelos, "Fundamentals of Session Types," *Inf. Comput.*, vol. 217, pp. 52–70, 2012.

[28] L. Padovani, "Fair Subtyping for Open Session Types," in *ICALP'13*, ser. LNCS, vol. 7966. Springer, 2013, pp. 373–384.

[29] R. Demangeon and K. Honda, "Full Abstraction in a Subtyped pi-Calculus with Linear Types," in *CONCUR'11*, ser. LNCS, vol. 6901. Springer, 2011, pp. 280–296.

[30] D. Kouzapas, N. Yoshida, and K. Honda, "On Asynchronous Session Semantics," in *FMOODS/FORTE'11*, ser. LNCS, vol. 6722. Springer, 2011, pp. 228–243.

[31] D. Mostrous, N. Yoshida, and K. Honda, "Global Principal Typing in Partially Commutative Asynchronous Sessions," in *ESOP'09*, ser. LNCS, vol. 5502. Springer, 2009, pp. 316–332.