An Event-Based Approach to Runtime Adaptation in Communication-Centric Systems

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

[1] I3S, UMR 7271, University of Nice Sophia Antipolis, Sophia Antipolis, Nice, France

[2] Johann Bernoulli Institute for Mathematics and Computer Science, University of Groningen, Groningen, The Netherlands

j.a.perez@rug.nl

Abstract. This paper presents a model of session-based concurrency with mechanisms for *runtime adaptation*. Thus, our model allows to specify communication-centric systems whose session behavior can be dynamically updated at runtime. We propose an *event-based* approach: adaptation requests, issued by the system itself or by its environment, are assimilated to events which may trigger runtime adaptation routines. Based on type-directed checks, these routines naturally enable the reconfiguration of processes with active sessions. We develop a type system that ensures *communication safety* and *consistency* properties: while the former guarantees absence of runtime communication errors, the latter ensures that update actions do not disrupt already established sessions.

1 Introduction

Context. Modern software systems are built as assemblies of heterogeneous artifacts which must interact following predefined protocols. Correctness in these *communication-centric* systems largely depends on ensuring that dialogues are consistent. *Session-based concurrency* is a type-based approach to ensure conformance of dialogues to prescribed protocols: dialogues are organized into units called *sessions*; interaction patterns are abstracted as *session types* [9], against which specifications may be checked.

As communication-centric systems operate on open infrastructures, *runtime adaptation* appears as a crucial feature to ensure continued system operation. Here we understand runtime adaptation as the dynamic modification of (the behavior of) the system in response to an exceptional event, such as, e.g., a varying requirement or a local failure. These events are not necessarily catastrophic but are hard to predict. As such, protocol conformance and dynamic reconfiguration are intertwined concerns: although the specification of runtime adaptation is not strictly tied to that of structured protocols, steps of dynamic reconfiguration have a direct influence in a system's interactive behavior.

We are interested in integrating forms of runtime adaptation into models of session-based concurrency. As a first answer to this challenge, in previous work [8] we extended a typed process framework for binary sessions with basic

DOI: 10.1007/978-3-319-33612-1_5

constructs from the model of *adaptable processes* [2]. In this work, with the aim of extending the applicability and expressiveness of the approach in [8], we propose adaptation mechanisms which depend on the state of the session protocols active in a given location. As a distinctive feature, we advocate an *event-based* approach: by combining constructs for *dynamic type inspection* and *non-blocking event detection* (as put forward by Kouzapas et al. [11,13]), adaptation requests, both internal or external to the location, can be naturally assimilated to events.

A Motivating Example. Here we consider a standard syntax for binary session types [9]:

$$\begin{array}{lll}\alpha, \beta ::= & ?(T).\beta & \text{input a value of typeT, continue as } \beta \\ & \mid\ !(T).\beta & \text{output a value of typeT, continue as } \beta \\ & \mid\ \&\{n_1{:}\alpha_1 \dots n_m{:}\alpha_m\} & \text{branching (external choice)} \\ & \mid\ \oplus\{n_1{:}\alpha_1 \dots n_m{:}\alpha_m\} & \text{selection (internal choice)} \\ & \mid\ \varepsilon \quad \mid \quad \mu t.\alpha \ \mid\ t & \text{terminated and recursive session}\end{array}$$

where T stands for both basic types (e.g., booleans, integers) and session types α. Also, $n_1, \dots, n_m$ denote *labels*. To illustrate session types, consider a buyer B and a seller S which interact as follows. First, B sends to S the name of an item and S replies back with its price. Then, depending on the amount, B either adds the item to its shopping cart or closes the transaction. In the latter case the protocol ends. In the former case B must further choose a paying method. From B's perspective, this protocol may be described by the session type $\alpha = !\mathsf{item}.\,?\mathsf{amnt}.\,\alpha_{\mathsf{pay}}$, where item and amnt are base types and

$$\alpha_{\mathsf{pay}} \;=\; \oplus\{\mathtt{addItem} : \oplus\{\mathtt{ccard} : \alpha_{\mathsf{cc}}\,,\, \mathtt{payp} : \alpha_{\mathsf{pp}}\}\,,\, \mathtt{cancel} : \varepsilon\}.$$

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

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

Thus, although both P and R implement α, their behavior is rather different, for they purchase different items using different payment methods (which are abstracted by processes P^{c} and R^{p}). Let us now analyze the situation for the seller S. To ensure protocol compatibility and absence of communication errors, the session type for S, denoted β, should be *dual* to α. This is written $\alpha \perp_{\mathsf{C}} \beta$. Intuitively, duality decrees that every action from B must be matched by a complementary action from S, e.g., every output of a string in α is matched by an input of a string in β. In our example, we let $\beta = ?\mathsf{item}.\,!\mathsf{amnt}.\,\beta_{\mathsf{pay}}$, where β_{pay} and a process implementation for S are as follows:

$$\beta_{\mathsf{pay}} = \&\{\mathtt{addItem} : \&\{\mathtt{ccard} : \beta_{\mathsf{cc}}\,,\, \mathtt{payp} : \beta_{\mathsf{pp}}\}\,,\, \mathtt{cancel} : \varepsilon\}$$
$$Q = y(i).\overline{y}(\mathit{price}(i)).y \triangleright \{\mathtt{addItem} : y \triangleright \{\mathtt{ccard} : Q^{\mathsf{c}} \parallel \mathtt{ppal} : Q^{\mathsf{p}}\} \parallel \mathtt{cancel} : \mathbf{0}\}$$

where *price* stands for an auxiliary function. Also, $\beta_{\mathtt{cc}}$ and $\beta_{\mathtt{pp}}$ are the duals of $\alpha_{\mathtt{cc}}$ and $\alpha_{\mathtt{pp}}$; they are realized by processes $Q_y^{\mathtt{c}}$ and $Q_y^{\mathtt{p}}$. The interaction of P and Q is defined using *session initialization* constructs: process $\overline{u}(x{:}\alpha).P$ denotes the *request* of a session of type α; dually, $u(x{:}\alpha).P$ denotes the *acceptance* of a session of type α. In both cases, u denotes a *(shared) name* used for synchronization. In our example, we may have

$$Sys = \overline{u}(x{:}\alpha).P \mid u(y{:}\beta).Q \longrightarrow (\nu\kappa)(P[\kappa^{+}/x] \mid Q[\kappa^{-}/y]) = S'$$

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

In this work, we are interested in ways of expressing and reasoning about the dynamic modification of session-typed processes such as P and Q above. Such modifications may be desirable to react to exceptional runtime conditions (say, an error) or to implement new requirements. For instance, the type below defines a new payment method for $\mathtt{S}$:

$$\beta_{\mathtt{gift}} \;=\; \&\{\mathtt{addItem} : \&\{\mathtt{giftc} : \beta_{\mathtt{gc}}\,,\, \mathtt{ccard} : \beta_{\mathtt{cc}}\,,\, \mathtt{payp} : \beta_{\mathtt{pp}}\}\,,\, \mathtt{cancel} : \varepsilon\}$$

Intuitively, $\beta_{\mathtt{gift}}$ *extends* $\beta_{\mathtt{pay}}$ with a new alternative on label $\mathtt{giftc}$. As such, it is safe to use a process implementing $\beta_{\mathtt{gift}}$ wherever a process implementing $\beta_{\mathtt{pay}}$ is required. The *safe substitution* principle that connects $\beta_{\mathtt{gift}}$ and $\beta_{\mathtt{pay}}$ is formalized by a *subtyping* relation on session types [7], denoted $\leq_{\mathtt{C}}$. In our example, we have $\beta_{\mathtt{pay}} \leq_{\mathtt{C}} \beta_{\mathtt{gift}}$.

In previous work [8] we studied how to update processes when sessions have not yet been established; this suffices to analyze runtime adaptation for processes such as Sys above. In this paper, we go further and address the runtime adaptation of processes such as S' above, which contain already established session protocols. As we would like to guarantee that adaptation preserves overall system correctness, a key challenge is ensuring that adaptation does not jeopardize such protocols. Continuing our example, let S'' be the process resulting from S' above, after the first step stipulated by α and β (i.e., an exchange of a value of type item). Intuitively, at that point, the buyer part of S' will have session type $?\mathsf{amnt}.\,\alpha_{\mathtt{pay}}$, whereas the seller part of S' will have session type $!\mathsf{amnt}.\,\beta_{\mathtt{pay}}$. Suppose we wish to modify at runtime the part of S'' realizing the buyer behavior. To preserve protocol correctness, a candidate new implementation must conform, up to $\leq_{\mathtt{C}}$, to the type $?\mathsf{amnt}.\,\alpha_{\mathtt{pay}}$; a process realizing any other type will fail to safely interact with the part of S'' implementing the seller. In [8] we defined the notion of *consistency* to formalize the correspondence between declared session protocols and the processes installed by steps of runtime adaptation. As we will see, consistency is still appropriate for reasoning about runtime adaptation of processes with active sessions.

Our Approach. Having motivated the context of our contributions, we move on to describe some technical details. We rely on a process language which extends session π-calculi with *locations*, *located processes*, and *update processes* [2]. We use *locations* as explicit delimiters for process behavior: these are transparent, possibly nested computation sites. Given a location loc and a process P, the *located process* $\mathrm{loc}[P]$ denotes the fact that P resides in loc (or, alternatively, that P has scope loc). This way, e.g., process

$$W = \mathrm{sys}\big[\,\mathrm{buyer}\big[\overline{u}(x{:}\alpha).P\big] \mid \mathrm{seller}\big[u(y{:}\beta).Q\big]\,\big]$$

represents an explicitly distributed variant of Sys above: the partners now reside in locations buyer and seller; location sys encloses the whole system. An *update process*, denoted $\mathrm{loc}\{U\}$, intuitively says that the behavior currently enclosed by loc should be replaced according to the adaptation routine U. Since a location may enclose one or more session channels, update processes allow for flexible specifications of adaptation routines. This way, e.g., one may specify an update on buyer that does not involve seller (and vice versa); also, a system-level adaptation could be defined by adding a process $\mathrm{sys}\{U_s\}$ in parallel to W, given an U_s that accounts for both buyer and seller behaviors.

The integration of runtime adaptation into sessions is delicate, and involves defining not only *what* should be the state of the system after adaptation but also *when* an adaptation step should be triggered. To rule out careless adaptation steps which jeopardize established protocols, communication and adaptation actions should be harmonized. As hinted at above, in previous work [8] we proposed admitting adaptation actions only when locations do not enclose running sessions. This is a simple solution that privileges communication over adaptation, in the sense that adaptation is enabled only when sessions are not yet active. Still, in realistic applications it may be desirable to give communication and adaptation a similar status. To this end, in this paper we admit the adaptation of locations with running sessions. We propose update processes $\mathrm{loc}\{U\}$ in which U is able to dynamically check the current state of the session protocols running in loc. In their simplest form, our update processes concern only one session channel and are of the shape

$$\mathrm{loc}\big\{\mathsf{case}\,x\,\mathsf{of}\;\{(x{:}\beta^i) : U_i\}_{i\in I}\big\}$$

where I is a finite index set, x denotes a channel variable, each β^i and U_i denotes a session type and an *alternative* (process) U_i, respectively. (We assume x occurs free in U_i.) The informal semantics for this construct is better understood by considering its interaction with a located process $\mathrm{loc}\big[Q\big]$ in which Q implements a session of type α along channel κ^p. The two processes may interact as follows. If there is a $j \in I$ such that types α and β^j "match" (up to $\leq_{\mathsf{C}}$), then there is a reduction to process $\mathrm{loc}\big[U_j[{}^{\kappa^p}\!/x]\big]$. Otherwise, if no β^j validates a match, then there is a reduction to process $\mathrm{loc}\big[Q\big]$, keeping the behavior of loc unchanged and consuming the update.

In general, update processes may define adaptation for locations enclosing more than one session channel. In the distributed buyer-seller example, the process below defines a runtime update which depends on the current state of the two channels at location sys:

$$U_{xy} = \text{sys}\left\{\mathsf{case}\, x, y\, \mathsf{of} \left\{ \begin{array}{c} (x{:}\alpha\,;\, y{:}\beta) : \text{buyer}[R] \mid \text{seller}[Q] \\ (x{:}\alpha_{\mathtt{pay}}\,;\, y{:}\beta_{\mathtt{pay}}) : \text{buyer}[P^*] \mid \text{seller}[Q^*] \end{array} \right\} \right\} \tag{1}$$

U_{xy} defines two possibilities for runtime adaptation. If the protocol has just been established (i.e., current types are α and β) then only the buyer is updated—its new behavior will be given by R above. If both item and price information have been already exchanged then implementations P^* and Q^*, compliant with types $\alpha_{\mathtt{pay}}$ and $\beta_{\mathtt{pay}}$, are installed.

Update processes rely on the protocol state at a given location to assess the suitability of adaptation routines. Our semantics for update relies on (a) *monitors* which store the current type for each running session; and (b) a type-directed test on the monitors enclosed in a given location. This test generalizes the `typecase` construct in [11].

While expressive, our typeful update processes by themselves do not specify *when* adaptation should be available. Even though update processes could be embedded within session communication prefixes (thus creating causal dependencies between communication and adaptation), such a specification style would only allow to handle exceptional conditions which can be fully characterized in advance. Other kinds of exceptional conditions, in particular contextual and/or unsolicited runtime conditions, are much harder to express by interleaving update processes within structured protocols.

To offer a uniform solution to this issue, we propose a *event-based* approach to trigger updates. We endow each location with a *queue* of adaptation requests; such requests may be internal or external to the location. In our example, an external request could be, e.g., a warning message from the buyer's bank indicating that an exchange with the bank is required before committing to the purchase with the seller.

Location queues are independent from session behavior. Their identity is visible to processes; they are intended as interfaces with other processes and the environment. To issue an adaptation request r for location loc, our process syntax includes *adaptation signals*, written $\overline{\text{loc}}(r)$. Similar to ordinary communication prefixes, these signals are orthogonal to sessions. Then, we may detect the presence of request r in the queue of loc using the *arrival predicate* $\mathsf{arrive}(\text{loc}, r)$ [11]. As an example, let $\mathtt{upd}_E$ denote an *external* adaptation request. To continuously check if an external request has been queued for sys, the process below combines process U_{xy} in (1) with arrival predicates, conditionals, and recursion:

$$U^*_{xy} = \mu\mathcal{X}.\mathsf{if}\ \mathsf{arrive}(\text{sys}, \mathtt{upd}_E)\ \mathsf{then}\ U_{xy}\ \mathsf{else}\ \mathcal{X} \tag{2}$$

We couple our process model for session-based concurrency and runtime adaptation with a type system that ensures the following key properties:

- *Safety*: well-typed programs do not exhibit communication errors (e.g., mismatched messages).

– *Consistency*: well-typed programs do not allow adaptation actions that disrupt already established sessions.

Safety is the typical guarantee expected from any session type discipline, here considered in a richer setting that combines session communication with runtime adaptation. In contrast, consistency is a guarantee unique to our setting: it connects the behavior of the adaptation mechanisms with the preservation of prescribed typed interfaces. We show that well-typed programs are safe and consistent (Theorem 3.6): this ensures that specified session protocols are respected, while forbidding incautious adaptation steps that could accidentally remove or disrupt the session behavior of interacting partners.

Organization. The rest of the paper is organized as follows. Next we present our event-based process model of session communication with typeful constructs for runtime adaptation (Sect. 2). Then, we present our session type system, which ensures safety and consistency for processes with adaptation mechanisms (Sect. 3). In Sect. 4 we discuss a process model of communication and adaptation with explicit compartments; it distills the main features of the model in Sect. 2. At the end, we discuss related works and draw some concluding remarks (Sect. 5). The appendix gives full sets of reduction and typing rules. Additional technical details and omitted definitions can be found in an online technical report [5].

2 The Process Model: Syntax and Semantics

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

Table 1 reports the syntax of expressions and processes. Processes include usual constructs for input, output, and labeled choice. Common forms of recursion, parallel composition, conditionals, and restriction are also included. As illustrated in Sect. 1, constructs for session establishment are annotated with a session type α, which is useful in derived static analyses. A prefix for closing a session, inherited from [8], is convenient to structure specifications. Variable x is bound in processes $\overline{u}(x{:}\alpha).P$, $u(x{:}\alpha).P$, and $k(x).P$. Binding for name and channel restriction is as usual. Also, recursion variable $\mathcal{X}$ is bound in process $\mu\mathcal{X}.P$. Given a process P, its sets of free/bound channels, names, variables, and recursion variables—noted $\mathsf{fc}(P)$, $\mathsf{fn}(P)$, $\mathsf{fv}(P)$, $\mathsf{fpv}(P)$, $\mathsf{bc}(P)$, $\mathsf{bn}(P)$, $\mathsf{bv}(P)$, and $\mathsf{bpv}(P)$, respectively—are as expected. We always rely on usual notions of α-conversion and (capture-avoiding) substitution, denoted $[{}^{k}\!/\!{}_{x}]$ (for channels) and $[{}^{P}\!/\!{}_{\mathcal{X}}]$ (for processes). We write $[{}^{k_1,\cdots,k_n}\!/\!{}_{x_1,\ldots,x_n}]$ to stand for an n-ary simultaneous substitution. Processes without free variables or free channels are called *programs*.

Table 1. Process syntax. Above, annotation α denotes a session type.

$e \ ::= v \mid x, y, z \mid k = k \mid a = a$	expressions
$\mid\ \mathsf{arrive}(\mathrm{loc}, r)$	arrival predicate
$P \ ::= \overline{u}(x : \alpha).P \mid u(x : \alpha).P \mid \mathsf{close}\,(k).P$	session request / acceptance / closure
$\mid\ \overline{k}(e).P \mid k(x).P$	data output /input
$\mid\ k \triangleleft n; P \mid k \triangleright \{n_i{:}P_i\}_{i \in I}$	selection / branching
$\mid\ \mu\mathcal{X}.P \mid \mathcal{X}$	recursion / rec. variable
$\mid\ P \mid P \mid \mathbf{0}$	parallel composition / inaction
$\mid\ (\nu\kappa)P \mid (\nu u)P \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	channel / name hiding / conditional
$\mid\ k\lfloor\alpha\rfloor$	session monitor
$\mid\ \mathrm{loc}[P]$	located process
$\mid\ \mathrm{loc}\big\{\mathsf{case}\,\widetilde{x}\,\mathsf{of}\,\{(x_1{:}\beta_1^i; \cdots ; x_m{:}\beta_m^i) : Q_i\}_{i\in I}\big\}$	typeful update process
$\mid\ \overline{\mathrm{loc}}(r) \mid \mathrm{loc}\lfloor\widetilde{r}\rfloor$	adaptation signal / queue

Up to here, the language is a synchronous π-calculus with sessions. Building upon *locations* $\mathrm{loc}, \mathrm{l}_1, \mathrm{l}_2, \ldots$, constructs for adaptation are: *located processes*, denoted $\mathrm{loc}[P]$; *update processes*, denoted $\mathrm{loc}\big\{\mathsf{case}\,x_1, \ldots, x_m\,\mathsf{of}\,\{(x_1{:}\beta_1^i; \cdots ; x_m{:}\beta_m^i) : Q_i\}_{i\in I}\big\}$; *(session) monitors*, denoted $\kappa^p\lfloor\alpha\rfloor$; *location queues*, denoted $\mathrm{loc}\lfloor\widetilde{r}\rfloor$; and *adaptation signals*, denoted $\overline{\mathrm{loc}}(r)$. Moreover, expressions include the *arrival predicate* $\mathsf{arrive}(\mathrm{loc}, r)$.

We now comment on these elements. *Located processes* and *update processes* have been motivated in Sect. 1. Here we just remark that update processes are assumed to refer to at least one variable x_i and to offer at least one alternative Q_i. Also, variables $x_1, \ldots, x_m$ are bound in $\mathrm{loc}\big\{\mathsf{case}\,x_1, \ldots, x_m\,\mathsf{of}\,\{(x_1{:}\beta_1^i; \cdots ; x_m{:}\beta_m^i) : Q_i\}_{i\in I}\big\}$; this process is often abbreviated as $\mathrm{loc}\big\{\mathsf{case}\,\widetilde{x}\,\mathsf{of}\,\{(x_1{:}\beta_1^i; \cdots ; x_m{:}\beta_m^i) : Q_i\}_{i\in I}\big\}$. Update processes generalize the `typecase` introduced in [11], which defines a case-like choice based on a single channel; in contrast, to specify adaptation for locations with multiple open sessions, our update processes define type-directed checks over one or more channels.

Update processes go hand-in-hand with *monitors*, runtime entities which keep the current protocol state at a given channel. We write $\kappa^p\lfloor\alpha\rfloor$ to denote the monitor which stores the protocol state α for channel κ^p. In [11], a similar construct is used to store in-transit messages in asynchronous communication. For simplicity, here we consider synchronous communication; monitors store only the current protocol state. This choice is aligned with our goal of identifying the core elements from the eventful session framework that are central in defining runtime adaptation (cf. Remark 3.7).

Location queues, not present in [11], handle adaptation requests, modeled as a possibly empty sequence of messages $\widetilde{r}$. Location queues enable us to give a unified treatment to adaptation requests, internal and external. Given $\mathrm{loc}\lfloor\widetilde{r}\rfloor$, it is worth observing that messages $\widetilde{r}$ are not related to communication as abstracted

Table 2. Reduction semantics: selected rules. Both α and β denote session types.

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

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

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

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

$$\langle\text{R:UREQ}\rangle \quad C\{\mathrm{loc}\lfloor\widetilde{r_1}\rfloor\} \mid D\{\overline{\mathrm{loc}}(r)\} \longrightarrow C\{\mathrm{loc}\lfloor\widetilde{r_1}\cdot r\rfloor\} \mid D\{\mathbf{0}\}$$

$$\langle\text{R:ARR1}\rangle \quad \frac{\widetilde{r} = r_1 \cdot \widetilde{r_0}}{C\{\mathsf{E}[\mathsf{arrive}(\mathrm{loc}, r_1)]\} \mid D\{\mathrm{loc}\lfloor\widetilde{r}\rfloor\} \longrightarrow C\{\mathsf{E}[\mathtt{true}]\} \mid D\{\mathrm{loc}\lfloor\widetilde{r_0}\rfloor\}}$$

$$\langle\text{R:ARR2}\rangle \quad \frac{(\widetilde{r} = r_2 \cdot \widetilde{r_0} \wedge r_1 \neq r_2) \vee \widetilde{r} = \epsilon}{C\{\mathsf{E}[\mathsf{arrive}(\mathrm{loc}, r_1)]\} \mid D\{\mathrm{loc}\lfloor\widetilde{r}\rfloor\} \longrightarrow C\{\mathsf{E}[\mathtt{false}]\} \mid D\{\mathrm{loc}\lfloor\widetilde{r}\rfloor\}}$$

$$\langle\text{R:UPD}\rangle \quad \frac{\begin{array}{c}\mathsf{fc}(P) = \{\kappa_1^p, \ldots, \kappa_m^p\} \qquad \forall j \in [1,..,m].(\kappa_j^p\lfloor\alpha_j\rfloor \in P) \\ (V = P) \bigvee \exists l.\big(\mathsf{match}_I(l,\ \{\alpha_1,\ldots,\alpha_m\},\ \{\beta_1^i,\ldots,\beta_m^i\}_{i\in I}) \wedge \\ V = Q_l\,[\kappa_1^p,\ldots,\kappa_m^p/\,x_1,\ldots,x_m]\big)\end{array}}{C\{\mathrm{loc}[P]\} \mid D\Big\{\mathrm{loc}\{\mathsf{case}\,\widetilde{x}\,\mathsf{of}\,\{(x_1{:}\beta_1^i;\cdots;x_m{:}\beta_m^i) : Q_i\}_{i\in I}\}\Big\} \longrightarrow C\{\mathrm{loc}[V]\} \mid D\{\mathbf{0}\}}$$

by session types. This represents the fact that we handle adaptation requests and structured session exchanges as orthogonal issues. An *adaptation signal* $\overline{\mathrm{loc}}(r)$ enqueues request r into the location queue of loc. To this end, as detailed below, the operational semantics defines synchronizations between adaptation signals and location queues. To connect runtime adaptation and communication, our language allows the coupling of update processes with the *arrival predicate on locations*, denoted $\mathsf{arrive}(\mathrm{loc}, r)$. Inspired by the `arrive` predicate in [11], this predicate detects if a message r has been placed in the queue of loc.

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

Semantics. The semantics of our language is given by a *reduction semantics*, the smallest relation generated by the rules in Table 5 (Appendix A). We write $P \longrightarrow P'$ for the reduction from P to P'. Reduction relies on a standard notion of structural congruence, denoted $\equiv$ (see [8, Def. 1]). It also relies on *evaluation* and *location* contexts:

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

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

Table 2 gives a selection of reduction rules; we comment on these rules below. The first four rules formalize session behavior within hierarchies of nested locations. Using duality for session types, denoted $\perp_{\mathsf{C}}$ (see [7] and Sect. 3), in rule $\langle \text{R:Open}\rangle$ the synchronization on a name u leads to establish a session on fresh channels κ^p and $\kappa^{\overline{p}}$; also, two monitors with the declared session types are created. Duality for polarities p is as expected: $\overline{+} = -$ and $\overline{-} = +$. Monitors are *local* by construction: they are created in the same contexts in which the session is established. Rule $\langle \text{R:Com}\rangle$ represents communication of a value: we require both complementary prefixes and that the monitors support input and output actions. After reduction, prefixes in processes and monitors are consumed. Similarly, rule $\langle \text{R:Sel}\rangle$ for labeled choice is standard, augmented with monitors. Rule $\langle \text{R:Clo}\rangle$ formalizes session termination, discarding involved monitors. The monitors in these three rules allow us to track the evolution of active session protocols.

The remaining rules in Table 2 define our event-based approach to runtime adaptation. Rule $\langle \text{R:UReq}\rangle$ treats the issue of an adaptation request r as a synchronization between a location queue and an adaptation signal. The queue and the signal may be in different contexts; this enables "remote" requests. Rules $\langle \text{R:Arr1}\rangle$ and $\langle \text{R:Arr2}\rangle$ resolve arrival predicates by querying the (possibly remote) queue $\widetilde{r}$. Rule $\langle \text{R:Upd}\rangle$ defines the typeful update of the current protocol state at loc, which is given by an indexed set of open sessions with their associated monitors. The rule attempts to match such protocol state with the first suitable option offered by an update process for loc. If there is no matching alternative the current protocol state at loc is kept unchanged. By an abuse of notation, we write $P_1 \in P$ to indicate that P_1 occurs in P, i.e., if $P = C[P_1]$ for some C. Formally, given an index set I over the update process, suitability with respect to the behavior at loc is defined by predicate match_I in Definition 2.1 below. Using subtyping $\leq_{\mathsf{C}}$ (see [7] and Sect. 3), the predicate holds for an $l \in I$ which defines a new protocol state.

In addition to the rules in Table 2, our semantics includes standard and/or self-explanatory treatments for reduction under evaluation contexts, parallel composition, located context, and restriction. Also, it accounts for applications of structural congruence, recursion and conditionals. The full set of rules is in Table 5 (Appendix A).

Definition 2.1 (Matching). *Given an index set I, session types $\alpha_1, \ldots, \alpha_m$, an indexed sequence of session types $\{\beta_1^i, \ldots, \beta_m^i\}_{i\in I}$, and an $l \in I$, we write*

$$\mathsf{match}_I(l,\, \{\alpha_1, \ldots, \alpha_m\},\, \{\beta_1^i, \ldots, \beta_m^i\}_{i\in I})$$

if and only if $\forall n < l.(\exists j \in [1..m].\ \beta_j^n \not\leq_{\mathsf{C}} \alpha_j) \wedge (\bigwedge_{h\in[1..m]} \beta_h^l \leq_{\mathsf{C}} \alpha_h)$.

Example 2.2. Recall process W given in the Introduction. According to our semantics:

$$\begin{aligned} W &\longrightarrow (\nu\kappa)\big(\mathrm{sys}\big[\,\mathrm{buyer}\big[P[\kappa^p/x] \mid \kappa^p\lfloor\alpha\rfloor\big] \mid \mathrm{seller}\big[Q[\kappa^{\overline{p}}/y] \mid \kappa^{\overline{p}}\lfloor\beta\rfloor\big]\,\big]\big) \\ &\longrightarrow^2 (\nu\kappa)\big(\mathrm{sys}\big[\,\mathrm{buyer}\big[P' \mid \kappa^p\lfloor\alpha_{\mathtt{pay}}\rfloor\big] \mid \mathrm{seller}\big[Q' \mid \kappa^{\overline{p}}\lfloor\beta_{\mathtt{pay}}\rfloor\big]\,\big]\big) \end{aligned}$$

Suppose that following an external request the seller must offer a new payment method. (a gift card). Precisely, we would like $\mathtt{S}$ to act according to the type $\beta_{\mathtt{gift}}$ given in Sect. 1. Let $\alpha_{\mathtt{gift}}$ be the dual of $\beta_{\mathtt{gift}}$. We then may define the following update process R^1_{xy}:

$$\mathrm{sys}\big\{\mathsf{case}\, x, y\, \mathsf{of}\, \{(x{:}\alpha_{\mathtt{pay}}\,;\, y{:}\beta_{\mathtt{pay}}) : \mathrm{buyer}\big[P' \mid x\lfloor\alpha_{\mathtt{gift}}\rfloor\big] \mid \mathrm{seller}\big[Q'' \mid y\lfloor\beta_{\mathtt{gift}}\rfloor\big]\}\big\}$$

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

3 Session Types for Eventful Runtime Adaptation

This section introduces a session type system for the process language of Sect. 2. Our main result (Theorem 3.6) is that well-typed programs enjoy both *safety* (absence of runtime communication errors) and *consistency* properties (update actions do not disrupt established sessions). Our development follows the lines of the typed framework in [8].

The syntax of session types (ranged over by $\alpha, \beta, \ldots$) has been already presented in the Introduction. We consider *basic types* (ranged over by $\tau, \sigma, \ldots$) and write $T, S, \ldots$ to range over τ, α. Therefore, although our process language copes with runtime adaptation, our type syntax is standard and retains the intuitive meaning of session types [9], which we now briefly recall. Type $?(\tau).\alpha$ (resp. $?(\beta).\alpha$) abstracts the behavior of a channel which receives a value of type τ (resp. a channel of type β) and then continues as α. Dually, type $!(\tau).\alpha$ (resp. $!(\beta).\alpha$) represents the behavior of a channel which sends a value of type τ and then continues as α. Type $\&\{n_1 : \alpha_1 \ldots n_m : \alpha_m\}$ describes a branching behavior: it offers m behaviors, and if the j-th alternative is selected then it behaves as described by type α_j ($1 \leq j \leq m$). In turn, type $\oplus\{n_1 : \alpha_1 \ldots n_m : \alpha_m\}$ describes the behavior of a channel which may select a single behavior among $\alpha_1, \ldots, \alpha_m$ and then continues as α_j. We use ε to type a channel with no communication behavior. Type $\mu t.\alpha$ describes recursive behavior; as usual, we consider recursive types under equi-recursive and contractive assumptions.

Along the paper we have informally appealed to *duality* and *subtyping* over session types (denoted $\perp_{\mathsf{C}}$ and $\leq_{\mathsf{C}}$, resp.). For the sake of space, we omit their full definitions; we just remark that since our session type structure is standard, we may rely on the (coinductive) definitions given by Gay and Hole [7], which are standard and well-understood.

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

Definition 3.1 (Interfaces). *We define an* interface *as the multiset whose underlying set of elements is* $\mathrm{I} = \{\mathsf{q}\, u{:}\alpha \mid \mathsf{q} \in \{\mathsf{lin}, \mathsf{un}\}\}$ *(i.e., a set of assignments from names to qualified session types). We use* $\mathcal{I}, \mathcal{I}', \ldots$ *to range over interfaces. We write* $dom(\mathcal{I})$ *to denote the set* $\{u \mid u : \alpha_{\mathsf{q}} \in \mathcal{I}\}$ *and* $\#_{\mathcal{I}}(a) = h$ *to mean that* a *occurs* h *times in* $\mathcal{I}$.

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

Definition 3.2 (Interface Ordering). *Given interfaces* $\mathcal{I}$ *and* $\mathcal{I}'$*, we write* $\mathcal{I} \sqsubseteq \mathcal{I}'$ *iff*

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

We now define our typing environments. We write q to range over qualifiers lin and un.

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

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

Table 3. Well-typed processes: selected rules.

$$\frac{}{\Theta, \mathrm{loc} : \mathcal{I} \vdash \mathrm{loc} \triangleright \mathcal{I}}\ \langle \text{T:LocEnv} \rangle \qquad \frac{}{\Gamma\,;\,\Theta \vdash k\lfloor \alpha \rfloor \triangleright k : \lfloor \alpha \rfloor ; \emptyset}\ \langle \text{T:Que} \rangle$$

$$\frac{}{\Gamma \vdash r_1 \triangleright msg}\ \langle \text{T:MSG} \rangle \qquad \frac{\Gamma \vdash \widetilde{r} \triangleright msg \quad \Gamma \vdash r_1 \triangleright msg}{\Gamma \vdash r_1 ; \widetilde{r} \triangleright msg}\ \langle \text{T:LOCQ} \rangle$$

$$\frac{\Theta \vdash \mathrm{loc} \triangleright \mathcal{I} \qquad \Gamma \vdash r \triangleright msg}{\Gamma\,;\,\Theta \vdash \mathsf{arrive}(\mathrm{loc}, r) \triangleright \mathsf{bool}}\ \langle \text{T:ARRIVE} \rangle \qquad \frac{\Gamma \vdash r \triangleright msg}{\Gamma\,;\,\Theta \vdash \overline{\mathrm{loc}}(r) \triangleright \emptyset ; \emptyset}\ \langle \text{T:Sig} \rangle$$

$$\frac{\Theta \vdash \mathrm{loc} \triangleright \mathcal{I} \qquad \Gamma\,;\,\Theta \vdash P \triangleright \Delta ; \mathcal{I}' \qquad \mathcal{I}' \sqsubseteq \mathcal{I}}{\Gamma\,;\,\Theta \vdash \mathrm{loc}[P] \triangleright \Delta ; \mathcal{I}'}\ \langle \text{T:Loc} \rangle \qquad \frac{\Gamma \vdash \widetilde{r} \triangleright msg}{\Gamma\,;\,\Theta \vdash \mathrm{loc}\lfloor \widetilde{r} \rfloor \triangleright \emptyset ; \emptyset}\ \langle \text{T:QLoc} \rangle$$

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

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

$$\frac{\Gamma\,;\,\Theta \vdash P \triangleright \Delta ; \mathcal{I} \quad k \notin \mathrm{dom}(\Delta)}{\Gamma\,;\,\Theta \vdash \mathsf{close}\,(k).P \triangleright \Delta, k : \varepsilon ; \mathcal{I}}\ \langle \text{T:Clo} \rangle \qquad \frac{\Gamma\,;\,\Theta \vdash P \triangleright \Delta_1 ; \mathcal{I}_1 \quad \Gamma\,;\,\Theta \vdash Q \triangleright \Delta_2 ; \mathcal{I}_2}{\Gamma\,;\,\Theta \vdash P \mid Q \triangleright \Delta_1 \cup \Delta_2 ; \mathcal{I}_1 \uplus \mathcal{I}_2}\ \langle \text{T:Par} \rangle$$

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

$$\frac{\Gamma\,;\,\Theta \vdash P \triangleright \Delta, \kappa^p : \langle\!\langle \alpha_1 \rangle\!\rangle, \kappa^{\overline{p}} : \langle\!\langle \alpha_2 \rangle\!\rangle ; \mathcal{I} \quad \alpha_1 \perp_{\mathsf{C}} \alpha_2}{\Gamma\,;\,\Theta \vdash (\nu \kappa)P \triangleright \Delta ; \mathcal{I}}\ \langle \text{T:CRes} \rangle$$

$$\frac{\Gamma\,;\,\Theta \vdash P \triangleright \Delta ; \mathcal{I} \quad \Delta \leq_{\mathsf{C}} \Delta' \quad \mathcal{I} \sqsubseteq \mathcal{I}'}{\Gamma\,;\,\Theta \vdash P \triangleright \Delta' ; \mathcal{I}'}\ \langle \text{T:SUB} \rangle \qquad \frac{\Gamma\,;\,\Theta \vdash P \triangleright \Delta ; \mathcal{I} \cup \mathcal{I}_u \quad u \notin \mathrm{dom}(\mathcal{I})}{\Gamma\,;\,\Theta \vdash (\nu u)P \triangleright \Delta ; \mathcal{I}}\ \langle \text{T:NRes} \rangle$$

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

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

meaning that, under environments Γ and Θ, process P has active sessions declared in Δ and interface $\mathcal{I}$. Selected typing rules are shown in Table 3; remaining rules can be found in Table 6 (Appendix B). Below we comment on some of the rules in Table 3: the rest are standard and/or self explanatory. Rule $\langle$T:Adapt$\rangle$ types update processes. Notice that the typing rule ensures that each process Q_i has exactly the same active sessions that those declared in the respective case. Also, we require that alternatives contain both processes and monitors. With $\mathcal{I}_j \sqsubseteq \mathcal{I}$ we guarantee that the process behavior does not "exceed" the expected behavior within the location. Rule $\langle \textsc{t:sub} \rangle$ takes care of subtyping both for typings Δ and interfaces. Rule $\langle \textsc{t:CRes} \rangle$ types channel restriction that ensures typing duality among partners of a session and their respective queues. Typing of queues is given by rule $\langle \textsc{t:Que} \rangle$ that simply assigns type $k : \lfloor \alpha \rfloor$ to queue $k\lfloor \alpha \rfloor$. Finally, rule $\langle \textsc{t:NRes} \rangle$ types hiding of service names, by simply removing their declarations from the interface $\mathcal{I}$ of the process. In the rule, $\mathcal{I}_u$ contains only declarations for u, i.e., $\forall v \neq u, v \notin \mathrm{dom}(\mathcal{I}_u)$.

Our type system enjoys the standard *subject reduction* property. We rely on *balanced* typings: Δ is balanced iff for all $\kappa^p : \alpha \in \Delta$ (resp. $\kappa^p : \lfloor \alpha \rfloor \in \Delta$) then also $\kappa^{\overline{p}} : \beta \in \Delta$ (resp. $\kappa^{\overline{p}} : \lfloor \beta \rfloor \in \Delta$), with $\alpha \perp_{\mathsf{C}} \beta$. The proof proceeds by induction on the last rule applied in the reduction; it adapts the one given in [8].

Theorem 3.3 (Subject Reduction). *If $\Gamma ; \Theta \vdash P \triangleright \Delta ; \mathcal{I}$ with Δ balanced and $P \longrightarrow Q$ then $\Gamma ; \Theta \vdash Q \triangleright \Delta' ; \mathcal{I}'$, for some $\mathcal{I}'$ and balanced Δ'.*

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

Definition 3.4 (κ-processes, κ-redexes, Errors). *A process P is a* κ-process *if it is a prefixed process with subject κ^p, i.e., P is one of the following:*

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

Process P is a κ-redex *if it contains the composition of exactly two κ-processes with opposing polarities. P is an* error *if $P \equiv (\nu\widetilde{\kappa})(Q \mid R)$ where, for some κ, Q contains $\underline{\text{either}}$ exactly two κ-processes that do not form a κ-redex $\underline{\text{or}}$ three or more κ-processes.*

Informally, a process P is called *consistent* if whenever it has a κ-redex then update actions do not destroy such a redex. Below, we formalize this intuition. Let us write $P \longrightarrow_{\mathrm{upd}} P'$ for any reduction inferred using rule $\langle \textsc{r:Upd} \rangle$. We then define:

Definition 3.5 (Safety, Consistency). *Let P be a process. We say P is* safe *if it never reduces into an error. We say P is* update-consistent *if and only if, for all P' and κ such that $P \longrightarrow^* P'$ and P' contains a κ-redex, if $P' \longrightarrow_{\mathrm{upd}} P''$ then P'' contains a κ-redex.*

We now state our main result; it follows as a consequence of Theorem 3.3.

Theorem 3.6 (Typing Ensures Safety and Consistency). *If $\Gamma ; \Theta \vdash P \triangleright \Delta ; \mathcal{I}$ with Δ balanced then program P is update consistent and safe.*

Remark 3.7 (Asynchronous Communication). We have focused on *synchronous* communication: this allows us to give a compact semantics, relying on a standard type structure. To account for asynchrony, we would require a runtime syntax for programs with queues for in-transit messages (values, sessions, labels). The type system must be extended to accommodate these new runtime processes. In our case, an extension with asynchrony would rely on the machinery defined in [11].

4 Discussion: A Compartmentalized Model of Communication and Adaptation

Given that the process model in Sect. 2 enables the interplay of communication and adaptation, how can we organize specifications to reflect a desirable separation of concerns? In ongoing work, with the aim of specifying systems at a high-level of abstraction, we have developed a model which defines *compartments* to isolate communication behavior and adaptation routines. Here we briefly describe this model, which is given in Table 4.

In a nutshell, programs of Sect. 2 are now organized into *systems*. A system G is the composition of a set of *applications* $A_1, \ldots, A_n$ each comprising three elements: a *behavior* R, a *state* S, and a *manager* $\mathcal{M}$. As a simple example of a system, we may consider the operating system of a smartphone, which is meant to manage a number of applications that may interact among them. Applications in our model can communicate between each other or exhibit intra-application communication. The behavior R is specified as a process; we distinguish between located processes representing service definitions from located processes which make use of such definitions. A reduction semantics (omitted) ensures that locations enclosing service definitions do not contain open (active) sessions. This may be convenient for defining adaptation strategies, since updates to service definitions may now be performed without concerns of disruption of active sessions. The state S collects session monitors and location queues and it is kept separate from R. As a simple example, the buyer-seller scenario given in Sect. 1 can be casted in our model as

$$\mathsf{byr}\big\langle \text{buyer}\big[\,\overline{u@\mathsf{slr}}(x:\alpha).P\big] \,;\, \mathsf{S_b} \,;\, \mathcal{M}_b\big\rangle \parallel \mathsf{slr}\big\langle \text{seller}\big[* \, u(y{:}\beta).Q\big] \,;\, \mathsf{S_s} \,;\, \mathcal{M}_s\big\rangle$$

That is, buyer and seller are implemented as separate applications, named byr and slr, respectively. Above, we have $\mathsf{S_b} = \text{buyer}\lfloor\epsilon\rfloor$ and $\mathsf{S_s} = \text{seller}\lfloor\epsilon\rfloor$.

While the manager $\mathcal{M}$ implements adaptation at the application (local) level, a *handler* $\mathcal{H}$ defines adaptation at the system (global) level. As we wish to describe communication behavior separately from adaptation routines, update processes are confined to handlers and managers. A manager is meant to react upon the arrival of an internal adaptation message $\mathtt{upd}_I$. As in Sect. 2, managers may act upon the issue of an internal update request $\mathtt{upd}_I$ for some location, whereas handlers may act upon the arrival of an external update request or an application upgrade request (denoted $\mathtt{upd}_E$ and $\mathtt{upg}$, respectively). A handler may either update or $\mathsf{upgrade}$ the behavior at some location loc within

Table 4. A compartmentalized model of communicating systems: syntax.

$$
\begin{aligned}
G &::= A \;\big|\; \mathcal{H} \;\big|\; (\nu\kappa)A \;\big|\; G_1 \parallel G_2 \;\big|\; \mathbf{0} \qquad A ::= a\langle R\,;\, \mathsf{S}\,;\, \mathcal{M}\rangle \\
\mathcal{H} &::= *\,\mathsf{if}\ \mathsf{arrive}(\mathrm{l}_1@a, \mathsf{upd}_E)\ \mathsf{then}\ \mathrm{l}_1\big\{\mathsf{case}\,\widetilde{x}\,\mathsf{of}\,\{(x_1{:}\beta_1^i;\cdots;x_m{:}\beta_m^i) : Q_i\}_{i\in I}\big\} \\
&\;\big|\;\; *\,\mathsf{if}\ \mathsf{arrive}(\mathrm{l}_1@a, \mathsf{upg})\ \mathsf{then}\ \mathrm{l}_1\{\!\{P\}\!\} \\
R &::= \mathrm{loc}\big[u(x{:}\alpha).P\big] \;\big|\; \mathrm{loc}\big[*\,u(x{:}\alpha).P\big] \;\big|\; \mathrm{loc}\big[P^-\big] \;\big|\; R_1 \mid R_2 \\
\mathsf{S} &::= \kappa^p\lfloor\alpha\rfloor \;\big|\; \mathrm{loc}\lfloor\widetilde{r}\rfloor \;\big|\; \mathsf{S}_1 \diamond \mathsf{S}_2 \;\big|\; \mathbf{0} \\
\mathcal{M} &::= *\,\mathsf{if}\ \mathsf{arrive}(\mathrm{l}_1, \mathsf{upd}_I)\ \mathsf{then}\ \mathrm{l}_1\big\{\mathsf{case}\,\widetilde{x}\,\mathsf{of}\,\{(x_1{:}\beta_1^i;\cdots;x_m{:}\beta_m^i) : Q_i\}_{i\in I}\big\} \;\big|\; \mathcal{M}_1 \circ \mathcal{M}_2 \;\big|\; \mathbf{0}
\end{aligned}
$$

application a; this is written loc@a. Upgrades are denoted $\mathrm{l}_1\{\!\{P\}\!\}$; they are a particular form of update intended for service definitions only. In Table 4 we write $*\mathsf{if}\ e\ \mathsf{then}\ P$ and $*u(x{:}\alpha).P$ as shorthands for persistent conditionals and services, respectively.

Our compartmentalized model induces specifications in which communication, runtime adaptation, and state (as in, e.g., asynchronous communication) are jointly expressed, while keeping a desirable separation of concerns. Notice that the differences between "plain" processes (as given in Sect. 2) and systems (as defined in Table 4) are mostly conceptual, rather than technical. In fact, the higher level of abstraction that is enforced by our model does not result in additional technicalities. We conjecture that a reduction-preserving translation of application-based specifications into processes does not exist—a main difficulty being, unsurprisingly, properly representing the separation between behavior and state. This difference in terms of expressiveness does not appear to affect the type system. In future work we plan to extend the typing discipline in Sect. 3 (and its associated safety and consistency guarantees) to systems.

5 Related Work and Concluding Remarks

Related Work. The combination of static typing and type-directed tests for dynamic reconfiguration is not new. For instance, Seco and Caires [14] study this combination for a calculus for object-oriented component programming. To the best of our knowledge, ours is the first work to develop this combination for a session process language. As already discussed, we build upon constructs proposed in [10–13]. The earliest works on eventful sessions, covering theory and implementation issues, are [10,12]. Kouzapas's PhD thesis [11] provides a unified presentation of the eventful framework, with case studies including event selectors (a building block in event-driven systems) and transformations between multithreaded and event-driven programs. At the level of types, the work in [11] introduces session set types to support the `typecase` construct. We use dynamic session type inspection only for runtime adaptation; in [11] `typecase` is part of the process syntax. This choice enables us to retain a standard session type syntax. Runtime adaptation of session typed processes—the main contribution of this paper—seems to be an application of eventful session types not previously identified.

Previous works on runtime adaptation for session types (binary and multiparty) include [1,3,8]. We have already commented on how our current approach enhances that in our previous work [8]. Both [1] and [3] study adaptation for multiparty communications, which already sets a substantial difference with respect to our work. In [3], a set of monitors which govern the behavior of participants are derived from a global specification. Self-adaptation for monitored processes is triggered by an external adaptation function, which is often left unspecified. As in our work, the operational semantics for adaptation in [3] uses (local) types and monitors; key differences include the use of type-directed checks for selecting adaptation routines that preserve consistency, and the use of events and queues to handle adaptation requests. The work [1] studies dynamic update for message passing programs; a form of consistency for updates over threads is ensured using multiparty session types, following an asynchronous communication discipline.

Concluding Remarks. Building upon [11], we have introduced an eventful approach to runtime adaptation of session typed processes. We identified the strictly necessary eventful process constructs that enhance and refine known mechanisms for runtime adaptation. Adaptation requests, both internal and external, are handled via event detectors and queues associated to locations. Our approach enables us to specify rich forms of updates on locations with running sessions; this represents a concrete improvement with respect to previous works [8]. We notice that expressing both internal and external exceptional events is useful in practice; for instance, both kinds of events coexist in BPMN 2.0 (see, e.g., [6, Chap. 4]). To rule out update steps that jeopardize running session protocols, we also introduced a type system that ensures communication safety and update consistency for session programs. We have also outlined a high-level model of structured interaction which organizes communication and adaptation components into a sensible structure.

Adaptation in our framework is "monotonic" or "incremental" in that changes always preserve/extend active session protocols, exploiting subtyping for enhanced flexibility. Interestingly, our framework can be modified so that arbitrary protocols are installed as a result of an update. One needs to ensure that the endpoints of a session are present in the same location: arbitrary updates are safe as long as both endpoints are simultaneously updated with dual protocols. To relax our framework in this way, we would need to modify definitions for session matching (Definition 2.1) and interface ordering (Definition 3.2).

In future work, we plan to further validate the constructs in our framework by revisiting the model of *supervision trees* (a mechanism for fault-tolerance in Erlang) that we gave in [4]. Other interesting topics for further development include accounting for *asynchronous* communication (cf. Remark 3.7) and extending our event-based approach to choreographic protocols; the framework in [3] may provide a good starting point.

Table 5. Reduction semantics: Full set of rules. Above, α and β denote session types.

$\langle\text{R:OPEN}\rangle$	$C\{u(x:\alpha).P\} \mid D\{\overline{u}(y:\beta).Q\} \longrightarrow (\nu\kappa)\big(C\{P[\kappa^p/x] \mid \kappa^p\lfloor\alpha\rfloor\} \mid D\{Q[\kappa^{\overline{p}}/y] \mid \kappa^{\overline{p}}\lfloor\beta\rfloor\}\big) \quad (\alpha \perp_{\mathsf{C}} \beta)$
$\langle\text{R:COM}\rangle$	$C\{\overline{\kappa}^{\,p}(v).P \mid \kappa^p\lfloor !(T).\alpha\rfloor\} \mid D\{\kappa^{\overline{p}}(x).Q \mid \kappa^{\overline{p}}\lfloor ?(T).\beta\rfloor\} \longrightarrow C\{P \mid \kappa^p\lfloor\alpha\rfloor\} \mid D\{Q[v/x] \mid \kappa^{\overline{p}}\lfloor\beta\rfloor\}$
$\langle\text{R:SEL}\rangle$	$C\{\kappa^p \triangleright \{n_j{:}P_j\}_{j\in J} \mid \kappa^p\lfloor \&\{n_j{:}\alpha_j\}_{j\in J}\rfloor\} \mid D\{\kappa^{\overline{p}} \triangleleft n_i; Q \mid \kappa^{\overline{p}}\lfloor\oplus\{n_j:\beta_j\}_{j\in J}\rfloor\} \longrightarrow C\{P_i \mid \kappa^p\lfloor\alpha_i\rfloor\} \mid D\{Q \mid \kappa^{\overline{p}}\lfloor\beta_i\rfloor\} \quad (i\in J)$
$\langle\text{R:CLO}\rangle$	$C\{\mathsf{close}\,(\kappa^p).P \mid \kappa^p\lfloor\varepsilon\rfloor\} \mid D\{\mathsf{close}\,(\kappa^{\overline{p}}).Q \mid \kappa^{\overline{p}}\lfloor\varepsilon\rfloor\} \longrightarrow C\{P\} \mid D\{Q\}$
$\langle\text{R:EVA}\rangle$	if $e \longrightarrow e'$ then $\mathsf{E}[e] \longrightarrow \mathsf{E}[e']$
$\langle\text{R:PAR}\rangle$	if $P \longrightarrow P'$ then $P \mid Q \longrightarrow P' \mid Q$
$\langle\text{R:RESN}\rangle$	if $P \longrightarrow P'$ then $(\nu a)P \longrightarrow (\nu a)P'$
$\langle\text{R:RESC}\rangle$	if $P \longrightarrow P'$ then $(\nu\kappa)P \longrightarrow (\nu\kappa)P'$
$\langle\text{R:STR}\rangle$	if $P \equiv P'$, $P' \longrightarrow Q'$, and $Q' \equiv Q$ then $P \longrightarrow Q$
$\langle\text{R:REC}\rangle$	$\mathsf{rec}\,\mathcal{X}.P \longrightarrow P[\mathsf{rec}\,\mathcal{X}.P/\mathcal{X}]$
$\langle\text{R:IFTRUE}\rangle$	$\mathsf{if}\ \mathtt{true}\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow P$
$\langle\text{R:IFFALSE}\rangle$	$\mathsf{if}\ \mathtt{false}\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow Q$
$\langle\text{R:UREQ}\rangle$	$C\{\text{loc}\lfloor\widetilde{r_1}\rfloor\} \mid D\{\overline{\text{loc}}(r)\} \longrightarrow C\{\text{loc}\lfloor\widetilde{r_1}\cdot r\rfloor\} \mid D\{\mathbf{0}\}$
$\langle\text{R:ARR1}\rangle$	$\dfrac{\widetilde{r} = r_1 \cdot \widetilde{r_0}}{C\{\mathsf{E}[\mathsf{arrive}(\text{loc}, r_1)]\} \mid D\{\text{loc}\lfloor\widetilde{r}\rfloor\} \longrightarrow C\{\mathsf{E}[\mathtt{true}]\} \mid D\{\text{loc}\lfloor\widetilde{r_0}\rfloor\}}$
$\langle\text{R:ARR2}\rangle$	$\dfrac{(\widetilde{r} = r_2 \cdot \widetilde{r_0} \wedge r_1 \neq r_2) \vee \widetilde{r} = \epsilon}{C\{\mathsf{E}[\mathsf{arrive}(\text{loc}, r_1)]\} \mid D\{\text{loc}\lfloor\widetilde{r}\rfloor\} \longrightarrow C\{\mathsf{E}[\mathtt{false}]\} \mid D\{\text{loc}\lfloor\widetilde{r}\rfloor\}}$
$\langle\text{R:UPD}\rangle$	$\dfrac{\begin{array}{c}\mathsf{fc}(P) = \{\kappa_1^p, \ldots, \kappa_m^p\} \qquad \forall j \in [1,..,m].(\kappa_j^p\lfloor\alpha_j\rfloor \in P) \\ (V = P) \bigvee \exists l.\big(\mathsf{match}_I(l,\ \{\alpha_1, \ldots, \alpha_m\},\ \{\beta_1^i, \ldots, \beta_m^i\}_{i\in I}) \wedge \\ V = Q_l\,[\kappa_1^p, \cdots, \kappa_m^p/\, x_1, \ldots, x_m]\big)\end{array}}{C\{\text{loc}[P]\} \mid D\Big\{\text{loc}\{\mathsf{case}\,\widetilde{x}\,\mathsf{of}\,\{(x_1{:}\beta_1^i; \cdots; x_m{:}\beta_m^i) : Q_i\}_{i\in I}\}\Big\} \longrightarrow C\{\text{loc}[V]\} \mid D\{\mathbf{0}\}}$

Acknowledgments. We are grateful to Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and the anonymous reviewers for useful remarks. This research was partially supported by COST Action IC1201: Behavioural Types for Reliable Large-Scale Software Systems.

A Reduction Semantics: Full Set of Rules

Table 5 gives the full set of reduction semantics rules.

Table 6. Additional typing rules.

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

$$\frac{}{\Gamma, x:\mathsf{bool} \vdash x \triangleright \mathsf{bool}}\ \langle\text{T:bVAR}\rangle \qquad \frac{}{\Gamma, x:\mathrm{name} \vdash x \triangleright \mathrm{name}}\ \langle\text{T:nVAR}\rangle$$

$$\frac{d = u \vee d = \kappa^p \vee d = x}{\Gamma \vdash d = d \triangleright \mathsf{bool}}\ \langle\text{T:EQ}\rangle \qquad \frac{\alpha \perp_{\mathsf{C}} \beta}{\Gamma, u:\langle\alpha_{\mathsf{q}},\beta_{\mathsf{q}}\rangle \vdash u \triangleright \langle\alpha_{\mathsf{q}},\beta_{\mathsf{q}}\rangle}\ \langle\text{T:SER}\rangle$$

$$\frac{}{\Gamma;\Theta \vdash \mathbf{0} \triangleright \emptyset;\emptyset}\ \langle\text{T:NIL}\rangle$$

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

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

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

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

$$\frac{\Gamma;\Theta\vdash e\triangleright\mathsf{bool} \qquad \Gamma;\Theta\vdash P\triangleright\Delta;\mathcal{I} \qquad \Gamma;\Theta\vdash Q\triangleright\Delta;\mathcal{I}}{\Gamma;\Theta\vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q\triangleright\Delta;\mathcal{I}}\ \langle\text{T:IF}\rangle$$

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

$$\frac{\Gamma;\Theta\vdash P\triangleright\Delta,k:\alpha_i;\mathcal{I}\qquad 1\le i\le m}{\Gamma;\Theta\vdash k\triangleleft n_i;P\triangleright\Delta,k:\oplus\{n_1:\alpha_1,\ldots,n_m:\alpha_m\};\mathcal{I}}\ \langle\text{T:SEL}\rangle$$

B Type System: Additional Typing Rules

Table 6 gives additional typing rules for the system in Sect. 3.

References

1. Anderson, G., Rathke, J.: Dynamic software update for message passing programs. In: Jhala, R., Igarashi, A. (eds.) APLAS 2012. LNCS, vol. 7705, pp. 207–222. Springer, Heidelberg (2012)
2. Bravetti, M., Di Giusto, C., Pérez, J.A., Zavattaro, G.: Adaptable processes. Logical Methods Comput. Sci. **8**(4:13), 1–71 (2012)
3. Coppo, M., Dezani-Ciancaglini, M., Venneri, B.: Self-adaptive monitors for multiparty sessions. In: PDP 2014, pp. 688–696. IEEE (2014)
4. Di Giusto, C., Pérez, J.A.: Session types with runtime adaptation: Overview and examples. In: PLACES. EPTCS, vol. 137, pp. 21–32 (2013)
5. Di Giusto, C., Perez, J.A.: An Event-Based Approach to Runtime Adaptation in Communication-Centric Systems. Research report, December 2014. https://hal.archives-ouvertes.fr/hal-01093090
6. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer, Berlin (2013)

7. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Inf. **42**(2–3), 191–225 (2005)
8. Di Giusto, C., Pérez, J.A.: Disciplined structured communications with disciplined runtime adaptation. Sci. Comput. Program. **97**, 235–265 (2015)
9. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
10. Hu, R., Kouzapas, D., Pernet, O., Yoshida, N., Honda, K.: Type-safe eventful sessions in Java. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 329–353. Springer, Heidelberg (2010)
11. Kouzapas, D.: A Study of Bisimulation Theory for Session Types. Ph.D. thesis, Imperial College London (2012)
12. Kouzapas, D., Yoshida, N., Honda, K.: On asynchronous session semantics. In: Bruni, R., Dingel, J. (eds.) FORTE 2011 and FMOODS 2011. LNCS, vol. 6722, pp. 228–243. Springer, Heidelberg (2011)
13. Kouzapas, D., Yoshida, N., Hu, R., Honda, K.: On asynchronous eventful session semantics. Math. Struct. Comput. Sci. **26**(2), 303–364 (2016)
14. Costa Seco, J., Caires, L.: Types for dynamic reconfiguration. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 214–229. Springer, Heidelberg (2006)