Disciplined Structured Communications with Consistent Runtime Adaptation

Cinzia Di Giusto  
CEA, LIST  
Saclay, France  
cinzia.digiusto@gmail.com

Jorge A. Pérez  
CITI and Department of Informatics  
FCT - New University of Lisbon, Portugal  
japerezp@gmail.com

ABSTRACT

Session types offer a powerful type-theoretic foundation for the analysis of structured communications, as commonly found in service-oriented systems. They are defined upon core programming calculi which offer only limited support for expressing adaptation and evolvability requirements. This is unfortunate, as service-oriented systems are increasingly being deployed upon highly dynamic infrastructures in which such requirements are central concerns. In previous work, we developed a process calculi framework of adaptable processes, in which concurrent processes can be replaced, suspended, or discarded at runtime. In this paper, we propose a session types discipline for a calculus with adaptable processes. Our framework offers an alternative for integrating runtime adaptation mechanisms in the analysis of structured communications. We show that well-typed processes enjoy consistency: communicating behavior is never interrupted by evolvability actions.

1. INTRODUCTION

Session types offer a powerful type-theoretic foundation for the analysis of complex scenarios of structured communications, as frequently found in service-oriented systems. They abstract communication protocols as basic interaction patterns, which are then statically checked against specifications in some core programming calculi—typically, a variant of the $\pi$-calculus. Introduced in [8, 9], session type theories have been extended in many directions—see [4] for a survey. Their practical relevance is witnessed by, e.g., their application in the verification of parallel systems [10].

In spite of these developments, we find that existing session-typed calculi do not explicitly support for reasoning about runtime adaptation. While channel mobility (delegation [12]) as supported by such calculi is often useful to model forms of dynamic reconfiguration, more general forms of adaptation/evolvability are not expressible or are hard to reason about. Runtime adaptation is an increasingly relevant issue nowadays, as distributed systems and applications are being deployed in open, highly dynamic infrastructures, such as cloud computing platforms. In such settings, runtime adaptation appears as a key feature to ensure continued system operation, reduce costs, and achieve business agility.

We thus observe a rather unfortunate discrepancy between (i) the evolvability capabilities of modern distributed systems in practice, and (ii) the forms of interaction available in the calculi upon which session types disciplines are defined.

In this paper, we propose an approach towards overcoming this discrepancy. We introduce a session types discipline for a language equipped with mechanisms for runtime adaptation. Rather than developing yet another session types discipline from scratch, we have deliberately preferred to build upon two existing lines of work. Our proposal results from combining the framework of adaptable processes, which we have developed together with Bravetti and Zavattaro in [1], with the main insights of the session type system put forward by Garralda et al. [6]. While adaptable processes represent an attempt for enhancing process calculi specifications with evolvability mechanisms, the work in [6] develops a session types theory for the Boxed Ambient calculus [2]. Despite these seemingly distant origins, our framework combines the most interesting ideas of both approaches into a simple yet expressive model of structured communications with explicit mechanisms for runtime adaptation.

We briefly describe our approach and results. Our process language includes a set of standard $\pi$-calculus constructs, extended with the located processes and the update processes introduced in [1]. Given a location $l$, a process $P$, and a context $Q(X)$ (i.e., a process with free occurrences of variable $X$), these processes are noted $l[P]$ and $l(Q(X))$, resp. They may interact so as to evolve into process $Q(P/X)$, which represents the update of process $P$ at $l$ with a reconfiguration routine (or built-in adaptation mechanism) embodied by $Q(X)$. Locations can be nested and are transparent: within $l[P]$, process $P$ can evolve autonomously, with the potential of interacting with some neighboring update process $l(Q(X))$, as just described. Hence, in our language communicating behavior coexists with update actions. This raises the need for disciplining both forms of interaction, respecting session types descriptions but also enforcing evolvability requirements. To this end, by observing that update corresponds to a form of (higher-order) process mobility, we draw inspiration from the session types in [6]. Indeed, such a type discipline ensures that session communications within Ambient hierarchies do not get interrupted by Ambient mobility steps.

We call this property session consistency.

While both Ambients and adaptable processes rely on nested located processes, Ambient mobility and evolvability steps are conceptually very different. In fact, Ambient mobility is only defined in a parent-child style, whereas located processes and update actions may interact independently of their relative position in the hierarchy induced by location nesting. This way, integrating our calculus for adaptable processes with the session types discipline in [6] roughly amounts to: (1) generalizing the operational semantics of [6] so as to account for adaptation in arbitrary process hier-
endowing the evolvability constructs with suitable annotations, describing the session behavior embodied in located and update processes; and (3) extending the typing judgments of \cite{9}, so as to be able to reason about process interfaces. Ultimately, this last step is what realizes a form of typeful adaptation, which contrasts with the untyped adaptation in \cite{1}. Well-typed processes in our framework satisfy basic correctness guarantees (formalized as a Subject Reduction result), which entails consistency for session-typed processes with runtime adaptation mechanisms.

This work is an initial step towards a theory of interaction in which processes—endowed with behavioral disciplines expressed as types—may dynamically evolve at runtime. We have focused on session consistency—a basic approach to discipline the interplay of evolvability steps and communication behavior. Issues such as deadlock-freedom, resource usage, and trustworthiness are also important—we plan to look into these in the future. Nevertheless, we believe that a major contribution of our paper is to address issues which, to our knowledge, have not yet been explored in the realm of session-typed processes.

Organization. Sect.2 presents our process language, and Sect.3 defines our session type system. In Sect.4 we state our main results (subject reduction and session consistency); an illustrative example is given in Sect.5. We discuss interface compatibility for update actions in Sect.6. Finally, Sect.7 discusses related work and Sect.8 collects some concluding remarks. An extended version of this paper, with proofs and additional examples, is available online.\cite{10}

2. THE PROCESS LANGUAGE

We extend standard session-typed languages \cite{9} with located processes and update actions. This allows us to explicitly represent adaptation mechanisms in models of structured communications. As in \cite{9}, our language is instrumented with some elements which support the static analysis. In particular, we annotate processes with session types—noted \(p_k\)—and interfaces (collections of session types)—denoted \(\Delta\). Both elements are introduced in Def.1.

Our syntax builds upon the following base sets: channels, ranged over \(c, d, \ldots\); names, ranged over \(a, b, x, \ldots\); locations, ranged over \(l, l', \ldots\); labels, ranged over \(n, n', \ldots\); constants (integers, booleans, names), ranged over \(k, k', \ldots\); process variables, ranged over \(X, X', \ldots\); recursion variables, ranged over \(y, Y, Y', \ldots\); and integers, ranged over \(j, h, \ldots\). We use \(u, u'\) to denote names and channels. Then, processes, ranged over \(P, Q, R, \ldots\) and expressions, ranged over \(e, e', \ldots\) are given by the grammar in Table 1.

We comment on the non-standard constructs in Table 1: intensions and conventions for the other constructs are as in \cite{9}. Processes open \(a(c : p_k)\) and close \((c)\) are used to establish and close a session on channel \(c\), resp. Once established, structured behavior on channels is possible. The exchange of expressions (with the expected simultaneous substitution \(\{h/k\}\)) is as usual; channel passing (delegation) is also supported. Thus, our language combines dynamic reconfiguration via channel passing with runtime adaptation via located and update processes, as we explain next.

A located process \(l^h[P]\) denotes a process \(P\) deployed at location \(l\). Inside \(l\), process \(P\) can evolve on its own and interact with external processes; we use \(l\) as a reference for a potential update actions (see below). In \(l^h[P]\), \(h\) stands for the number of active sessions in \(Q\): our operational semantics use this information to ensure that locations enclosing active sessions are not interrupted by update actions (consistency). We write \(P(X)\) to denote a process \(P\) with zero or more occurrences of process variable \(X\). Also, we write \(Q\{P/X\}\) to denote the process \(Q\) in which free occurrences of \(X\) are substituted with \(P\). Then, an update process \(l^h_\Delta\{P(X)\}\) represents a built-in adaptation mechanism, located at \(l\). In fact, as explained below, if the interface of \(Q\) and \(\Delta\) are compatible then processes \(l^h\{Q\}\) and \(l^h_\Delta\{P(X)\}\) may interact so as to evolve into \(Q\{P/X\}\). Thus, intuitively, process \(l^h_\Delta\{Q(X)\}\) behaves as a function which expects a process with interface compatible with \(\Delta\) and returns an adapted process with interface \(\Delta_2\).

Binders for the language are as follows: channel \(c\) is bound in open \(a(c : p_k)\) and \(\tilde{c}\) is bound in \(c(\tilde{c})\); \((P, Q)\) and \(P\{Q\}\) are all distinct). The set of free and bound channels and names of a process \(P\) is denoted by \(\text{fn}(P), \text{fc}(P), \text{bn}(P), \text{bc}(P)\), resp. as expected. We define \(\text{fu}(P) \triangleq \text{fn}(P) \cup \text{fc}(P)\). The treatment of recursion variables is also as customary. As for process variables, we assume update processes bind process variables in it. In all cases, we rely on usual notions of \(\alpha\)-conversion (noted \(\equiv_\alpha\)) and (capture-avoiding) substitution. We work only with closed processes.

The semantics of our language is given by a reduction semantics, denoted \(P \rightarrow P'\), the smallest relation on processes generated by the rules in Table 2. It relies on an evaluation relation on expressions (noted \(e \downarrow k\)) and on a structural congruence relation, defined as the smallest congruence generated by the following laws:

\[
\begin{align*}
(P \parallel Q) \triangleright R & \equiv R \triangleright (Q \parallel R) \\
(vu)^l & \equiv (vu)^l & \quad (vu)^{l_1}[(vu)^{l_2}] & \equiv (vu)^{l_1}[(vu)^{l_2}] \\
0 & \equiv 0 & \quad (vu)(vu') & \equiv (vu')(vu) \\
Q & \equiv Q \parallel ((vu)^0) & \quad (vu)(vu') & \equiv (vu')(vu) \\
Q & \equiv (vu)^0 & \quad (vu)(vu') & \equiv (vu')(vu) \\
Q & \equiv Q & \quad (vu)(vu') & \equiv (vu')(vu) \\

\end{align*}
\]

We write \(\rightarrow^*\) for the reflexive, transitive closure of \(\rightarrow\). As processes can be arbitrarily nested inside locations, in reduction rules we use (syntactic) contexts, i.e., processes with a hole \(\bullet\):

\[
C, D, E, \ldots : \equiv \bullet | l^h[C] | C \parallel P
\]

We assume the expected extension of \(\equiv\) to contexts. To ensure coherence of located processes along reduction, we define an operation over contexts which allows us to increase/decrease \(h\). Given a context \(C\), an integer \(j\), and \(s \in \{+, -\}\), we define:

\[
\bullet \vdash^n_j \bullet \equiv (l^n[C])^j \equiv l^n_j[(C)^j] \quad (C \parallel P)^j = (C)^j \parallel P
\]

We write \(C^-\) and \(C^+\) to stand for \(C^{-1}\) and \(C^{+1}\), resp. We now comment on rules (R:OPEN), (R:CLOSE), and (R:UPD) in Table 2.
(R:OPEN)
\[ E[C[\text{open} a (c : \rho_a), P]] \equiv D[\text{open} a (d : \tau_a), Q] \longrightarrow (\nu c') (E^c (C^c [P[c'/c]] \equiv D^c [Q[c'/d]])) \]

(R:I/O)
\[ E[C[\pi(c).P] \equiv D[c(\bar{x}).Q]] \longrightarrow E[C[P[\bar{k}/\bar{x}]] \equiv D[Q]] \quad (\bar{e} \downarrow \bar{k}) \]

(R:UPD)
\[ E[C[p^0(Q)] \equiv D[p^0(Q(X))] \quad (\text{with } \Delta_\emptyset \equiv \emptyset) \] \[ \longrightarrow E[C[P[\emptyset/Q(X)]] \equiv D[\emptyset]] \]

(R:PASS)
\[ E[C[c!d(P)] \equiv D[c!!d!(Q)] \longrightarrow E[C[P] \equiv D[Q]] \]

(R:SEL)
\[ E[C[\{n_i : P_i\}_{i \in I}] \equiv D[c \triangleq n_j, Q] \longrightarrow E[C[P_i] \equiv D[Q]] \quad (j \in I) \]

(R:REC)
\[ E[C[\text{rec}(Y : \Phi, \Delta), P)] \longrightarrow C[P[\text{rec}(Y : \Phi, \Delta), P/Y]] \]

(R:IFTr)
\[ C[if \; e \; \text{then} \; P \; else \; Q] \longrightarrow C[P] \quad (e \downarrow \text{true}) \]

(R:IFFa)
\[ C[if \; e \; \text{then} \; P \; else \; Q] \longrightarrow C[Q] \quad (e \downarrow \text{false}) \]

(R:CLOSE)
\[ E[C[\text{close}(c).P] \equiv D[\text{close}(c), Q]] \longrightarrow E^c (C^c [P] \equiv D^c [Q]) \]

(R:STR)
\[ \text{if } P \equiv P', \; P' \longrightarrow Q', \; \text{and } Q' \equiv Q \; \text{then } P \longrightarrow Q \]

(R:RES)
\[ \text{if } P \longrightarrow P' \; \text{then} \; (\nu u)P \longrightarrow (\nu u)P' \]

Table 2: Reduction Semantics

other rules are self-explanatory.

Rule (R:OPEN) defines session establishment by the interaction of two open prefixes on the same name and declaring dual session types (cf. Def. 9). When a session is initiated, the involved processes are restricted with a channel c', which we assume fresh. This way, channel restriction is only generated at runtime. Because of session initiation, the number of active sessions is increased by one in all enclosing contexts. Rule (R:CLOSE) is analogous: it decreases the active session annotation in all enclosing contexts. Let us write \( \Delta_{P} \) for the interface of process \( P \) (cf. Sect. 6). Rule (R:Upd) formalizes update actions. Two conditions are enforced:

1. Only located processes in which the number of open sessions is 0 can be updated. This is to avoid disrupting already established sessions.

2. The interface of the located process (\( \Delta_{P} \)) and the requirements of the update process (\( \Delta_{Q} \)) must be compatible—note \( \Delta_1 \equiv \Delta_{Q} \). For the sake of simplicity, in the remainder we assume \( \equiv \). Sect. 10 discusses more flexible definitions.

An update (or evolvability) step is then realized by substituting Q (the current behavior of the located process) into P(X). This is a form of objective update, as the located process does not contain information on future update actions: it reduces autonomously until it is adapted by an update process in its surrounding context.

3. THE TYPE SYSTEM

Our type discipline builds upon the one given in [6]. As such, the type syntax is divided into three categories: basic types, which handle expressions; pre-session types describing a channel’s ability for sending/receiving messages and offering/selecting behaviors; session types representing well-formed session-typed behavior; and channel types, which can be a session type or a used session type \( \perp \).

We extend the judgments of [6] with an interface \( \Delta \) —the collection of all session types declared in a given process. As we have seen, interfaces are key to the operational definition of update actions. To define the interface of processes with recursion, our session types are qualified: they can be linear (lin) or unrestricted (un). While lin is used for those session types to be used just once, un annotates those session types intended to feature a persistent behavior—roughly, these are types assigned to sessions declared in the context of a recursive process. Formally, we have:

\[ \Delta ::= \emptyset \mid \Delta, (\rho, k) \]

Table 3: Types and Typing Environments

\[ \Phi ::= \emptyset \mid \Phi, c : \omega \]

Table 4: Dual of pre-session types

\[ \Theta ::= \emptyset \mid \Theta, Y : \Phi, \Delta \mid \Theta, X : \Delta \]

Table 3: Types and Typing Environments

Table 4: Dual of pre-session types

Our type syntax contains the usual session types constructs, with their standard meanings. We use \( t, t', \ldots \) to range over type variables; also, we require recursive types to be contractive, i.e., not containing a subexpression of the form \( \mu t, \mu t_{1}, \ldots, \mu t_{n}, t \). Binder \( \mu \) gives rise to standard notions of free/bound type variables; \( \text{fv}(\zeta) \) denotes the free type variables in \( \zeta \). We consider session types modulo folding/unfolding of recursive types. We now introduce the central notion of duality for (pre-session) types.

Definition 1. Basic types (\( \kappa \)), pre-session types (\( \zeta \)), session types (\( \rho_{0}, \sigma_{0} \)), and channel types (\( \omega \)) are as in Table 3.

Our type syntax contains the usual session types constructs, with their standard meanings. We use \( t, t', \ldots \) to range over type variables; also, we require recursive types to be contractive, i.e., not containing a subexpression of the form \( \mu t, \mu t_{1}, \ldots, \mu t_{n}, t \). Binder \( \mu \) gives rise to standard notions of free/bound type variables; \( \text{fv}(\zeta) \) denotes the free type variables in \( \zeta \). We consider session types modulo folding/unfolding of recursive types. We now introduce the central notion of duality for (pre-session) types.

Definition 2 (Duality). Given a pre-session type \( \zeta \), its dual (noted \( \overline{\zeta} \)) is inductively defined in Table 4.

We comment on the typing environments defined in Table 3. An interface \( \Delta \) is a set of pairs \( (\rho, k) \) where \( \rho \) is a session type and
\( k \) is either an integer \( j > 0 \) or \( \infty \). The intention is to capture the session types declared in a process, the number of times a type is declared, and whether a type is linear or unrestricted. In \( \Delta \), types are recorded at most once; several occurrences of the same type are captured by the second component of the pair. The union of an interface and a pair, noted \( \Delta \uplus \langle \rho, k \rangle \), is defined as follows:

\[
\Delta \uplus \langle \rho, k \rangle = \begin{cases} 
\Delta' \cup \{\{\rho, k+1\}\} & \text{if } \exists \Delta', \Delta = \Delta' \cup \{(j, \rho)\} \\
\Delta' \cup \{(j, \rho)\} & \text{otherwise}
\end{cases}
\]

Recall that \( \infty + 1 = \infty \). We sometimes write \( \Delta \uplus \rho \) to stand for \( \Delta \uplus \{\rho, 1\} \) (if \( q = \text{lin} \)) or \( \Delta \uplus \{\rho, \infty\} \) (if \( q = \text{un} \)). The extension of \( \Delta \) to pairs \( \langle \rho, k \rangle \) to interfaces (denoted \( \Delta_{\rho} \uplus \Delta_{\rho} \)) is as expected. Letting \( \alpha \) stand for session prefixes (expression/channel input and output, session selection and close), the interface of a process \( P \), denoted \( \Delta_{\rho} \), is inductively defined in Table 5.

Set \( \Phi \) collects information on already active sessions: it records a channel name and its associated session type. While \( \Gamma \) is a first-order environment which maps expressions with basic types \( \kappa \), the higher-order environment \( \Theta \) stores the type of process/recursion variables. Given these environments, a type judgment is of form

\[
\Gamma; \Theta \vdash P :: \Phi; \Delta
\]

meaning that, under environments \( \Gamma \) and \( \Theta \), process \( P \) has active sessions declared in \( \Phi \) and interface \( \Delta \). We then have:

**Definition 3.** A process is well-typed if it can be typed using the rules in Tables 6 and 7.

We comment on some typing rules. Rule (T:OPEN) types the process open \( (c : \rho_0).P \). Observe how channel \( c \) becomes bound and so it is removed from \( \Phi \); simultaneously, \( \Delta \) is extended with the now complete (non-active) session \( \rho_0 \). Rule (T:LOC) checks that \( h \) corresponds to the number of running sessions in \( P \), i.e., the cardinality of \( \Phi \). Rule (T:UPD) requires that the type of \( P(X) \) is \( \Delta_2 \) provided that \( X \) is of type \( \Delta_1 \). This way, the rule guarantees that the process resulting from an update action has the declared interfaces. Rule (T:PAR) checks that only matching processes can be put in parallel. This is obtained via the merge operator \( \triangleright \), which ensures session type compatibility via duality.

**Definition 4.** We define \( \omega_1 \triangleright \omega_2 = \bot \) if \( \{\omega_1, \omega_2\} = \{\rho, \bar{\rho}\} \) and \( \omega_1 \triangleright \omega_2 = \text{undefined otherwise} \). Then:

\[
\Phi_1 \triangleright \Phi_2 = \{c : \omega_1 \triangleright \omega_2 | c : \omega_1 \in \Phi_1 \text{ and } c : \omega_2 \in \Phi_2\} \cup \{c : \omega_1 | c : \omega_1 \in \Phi_1 \text{ and } c \notin \text{dom} (\Phi_2)\} \cup \{c : \omega_2 | c : \omega_2 \in \Phi_2 \text{ and } c \notin \text{dom} (\Phi_1)\}
\]

Rule (T:CLOSE) extends \( \Phi \) with a new channel with an empty session type. Rules (T:PVAR) and (T:RVAR) handle process and recursion variables, resp. Rule (T:WEAK) handles all those processes in which \( (\nu c)P \equiv P \) holds, and (T:CRESC) types channel restriction (the expected rule for name restriction is elided). While rules (T:CAT) and (T:THR) handle delegation, rules (T:IN), (T:OUT), (T:TRA), and (T:SEL) are standard.

### 4. Session Consistency by Typing

We now investigate session consistency: this is to enforce a basic discipline on the interplay of communicating behavior (i.e., session interactions) and evolvability behavior (i.e., update actions). We say that sessions are consistent when they are not disrupted by an evolvability step. That is, performing an update action does not affect the behavior of active sessions. To define consistency, we extend and adapt notions given in [6]. Process \( P \) is said to be communicating over channel \( c \) if it is one of the following:

\[
\begin{align*}
1. \quad & (c(x)).P' \\
2. \quad & \tau(v).P' \\
3. \quad & (c'!)d.P' \\
4. \quad & c!(d).P' \\
5. \quad & c \triangleright \{n_1 : P_1, \ldots , n_k : P_k\} \\
6. \quad & c \triangleright \{n_1 : P_1, \ldots , n_k : P_k\}
\end{align*}
\]

Two communicating processes are dual (on session \( c \)) if they are, respectively, of the forms (1) and (2), (3) and (4), (5) and (6), or a pair of (7), as given above. Let us write \( P_{(c)} \) and \( P_{(c)}' \) for two dual
This result follows from Thm. 1 by observing that enabling update actions only for located processes without active sessions (cf. rule (r:UPD)), essentially rules out the possibility of updating a location containing a communicating process \( P_{\text{loc}} \), as defined above. Indeed, our type system ensures that the annotations enabling update actions are correctly assigned and maintained along reduction.

### 5. A DISTRIBUTED SERVICE SCENARIO

As an example, let us consider a simple scenario of distributed, interacting services (a client \( C \) and a service \( S \)), conveniently represented as located processes:

\[
\text{Sys} \triangleq \Gamma \quad \text{open} (c : \nu \sigma). \left( \text{r} (u, p), c \triangleright n_1 ; Q_1 \|^* n_1 ; Q_2 \right) . \text{close} (c) \\
\text{S} \triangleq \text{open} (s : \sigma). \left( s (u, p) \|^* n_1 , P_1 . \text{close} (s) \right)
\]

and where \( R \) represents the platform in which \( S \) is deployed. Sys may proceed by establishing a session of type \( \sigma \) between \( S \) and \( C \) (realized by \( P_1 \) and \( Q_1 \)). Suppose that \( R \) stands for an adaptation routine for \( S \), defined as an update process:

\[
\text{R} \triangleq r'' \left\{ \text{r}''^1 (\text{S'} \|^* \text{R} \|^* \text{T}) \right\}
\]

Thus, an update action on \( r \) reconfigures the distributed structure of \( S \): in a single step, the monolithic service \( S \) is replaced by a more flexible implementation in which \( S' \) (located at \( r_1 \)) first establishes a session with \( C \) and then delegates it to \( T \) (located at \( r_2 \)). This update action is transparent to \( C \); it is possible because the interface of \( S \) (i.e., \( \sigma \)) coincides with the annotation of \( R \). Also, observe that interface \( \Delta \) contains entries for session types \( \nu (\sigma) \) and \( \nu (? \sigma) \), which are declared in \( S' \) and \( T \), resp.

This example can be easily extended with infinite behavior (e.g., with \( S \) and \( R \) as persistent services) and with multiple clients. The above scenario already illustrates how our framework extends the expressiveness of session-typed languages with located and update processes. Our type system not only ensures correct communication, but also enforces a compatibility relation between \( C \) and \( S \) by relying on Cor. 1. We know that well-typedness implies session consistency, i.e., an update action on \( r \) will not occur if \( C \) and \( S \) have already initiated a session.

### 6. ON INTERFACE COMPATIBILITY

The reduction semantics of Sect. 4 is parametric with respect to a compatibility relation \( \simeq \) on interfaces \( \Delta \). We now informally discuss some alternatives; a formal treatment of interface compatibility is left for future work.

Until now, we assumed \( \simeq \triangleq \triangleq \), therefore requiring that interfaces should be exactly the same. This definition may be overly restrictive in specifications. A more flexible definition can be obtained by exploiting subtyping. Given types \( \rho, \sigma \), we say that \( \rho \) is a subtyping of \( \sigma \) (noted \( \rho \leq \sigma \)) if, intuitively, any process of type \( \rho \) can safely be used in a context where a process of type \( \sigma \) is expected. Subtyping for session types has been studied in [7, 11]; it arises from subset relations on basic types (as in, e.g., \( \mathbb{R} \subseteq \mathbb{R} \)) and from branching and selection constructs. Assuming \( \rho \leq \sigma \), for
all \( i \in \{1, \ldots, m\} \) and \( k \geq 0 \), we have:

\[
\&\{n_1 : \rho_1, \ldots, n_k : \rho_n\} \leq \&\{n_1 : \sigma_1, \ldots, n_k : \sigma_{m+k}\} \\
\oplus\{n_1 : \rho_1, \ldots, n_k : \rho_{n+k}\} \leq \oplus\{n_1 : \sigma_1, \ldots, n_k : \sigma_m\}
\]

Moreover, \( \leq \) is co-variant for input prefixes and contra-variant for outputs. This way, a compatibility relation based on \( \leq \) could be defined by means of a bijection \( f : \Delta_1 \rightarrow \Delta_2 \), such that for all \( \rho \in \Delta_1 \), we have \( \rho \leq f(\rho) \). Another compatibility relation could rely on the type equivalence induced by \( \leq \); types \( \sigma \) and \( \sigma' \) are equivalent if \( \sigma \leq \sigma' \) and \( \sigma' \leq \sigma \) hold. (This can be formalized by adopting an equi-recursive convention for recursive types.) Other definitions for \( = \) could be obtained by enriching \( \Delta[p] \) with information on the distributed structure (location nesting) in \( P \). This would allow to combine structural and behavioral considerations when reasoning about interfaces. In general, incorporating these definitions of compatibility would require (minor) adjustments in our type system (precisely, in rules such as \((T:\text{LOC})\) and \((T:\text{UPD})\)).

7. RELATED WORK

To our knowledge, ours is the first attempt to incorporate rich adaptation/evolvability constructs into a session-typed process language. Related to our work are efforts on formal models for service-oriented systems with constructs such as exceptions and compensations (e.g., \[4\][3]). In our view, such notions represent only a particular instance of adaptation/evolvability scenarios. That is, while exceptions/compensations are typically conceived for handling exceptional, catastrophic events (such as errors), in our view, runtime adaptation in modern distributed systems aims at covering more general events, not necessarily catastrophic. As an example, consider elasticity in cloud-based applications, i.e., the ability such applications have to acquire/release computing resources based on user demand. Although elasticity triggers system adaptation, it does not represent a catastrophic event; rather, it represents an acceptable (yet hard to predict) state of the system. Because of this conceptual difference, we do not have a clear perspective as to how known models of exceptions/compensations can be used to express the adaptation capabilities that are expressible in our framework.

As already discussed, our approach has been greatly influenced by \[6\][1]. Nevertheless, there are several significant differences between our framework and those works. (1) Unlike the system in \[6\], our framework supports channel passing (delegation). As delegation is already useful to represent forms of dynamic reconfiguration, its interplay with update actions is very appealing. (2) We have extended typing judgments in \[6\] with interfaces \( \Delta \), which are central to characterize located processes which can be safely updated. (3) While in \[1\] adaptable processes are defined for a fragment of CCS, here we consider them within a typed \( \pi \)-calculus. (4) Adaptation steps in \[1\] are completely unrestricted. Here we have considered annotated versions of the constructs in \[1\]: they offer a more realistic account of update actions, as they are supported by runtime conditions based on session types.

8. CONCLUDING REMARKS

We have proposed a framework for reasoning about runtime adaptation in the context of structured communications. More precisely, we introduced a session types discipline for an extended \( \pi \)-calculus, in which channel mobility is enhanced with update actions on located processes. Our approach consisted in extending and integrating two existing lines of work, on abstractions for evolvability in process calculi \[1\] and on session types for mobile calculi \[6\]. We focused on statically ensuring consistency, i.e., a correct interplay between session behavior and evolvability steps.

A main motivation for our work is the observation that while paradigms such as service-oriented computing are increasingly popular among practitioners, formal models based on them—such as reasoning techniques based on session types—fail to capture distinctive aspects of such paradigms. Here we have aimed at addressing, for the first time, one of such aspects, namely runtime adaptation. In our view, it represents an increasingly important concern when analyzing the behavior of communicating systems in open, context-aware computing infrastructures. As future work, we plan to investigate stronger notions of correctness for our adaptable, session-typed processes, in particular deadlock-freedom. Also, following \[1\] and the discussion in Sect. \[6\] we would like to integrate subtyping into our type system, and to obtain algorithmic characterizations of notions such as type duality and interface compatibility.

Acknowledgments. We thank António Ravara, Fabrizio Montesi, and the anonymous reviewers for their useful and detailed remarks. This work was partially supported by the French project ANR-11-INSE-0007-REVER, and by the Portuguese Foundation for Science and Technology (FCT) through the Carnegie Mellon Portugal Program—grants INTERFACES NGN-44 / 2009 and CITI.

9. REFERENCES