Session Types for Dynamically Evolvable Communicating Systems

Cinzia Di Giusto\textsuperscript{1} and Jorge A. Pérez\textsuperscript{2}

\textsuperscript{1} Université d’Evry - Val d’Essonne, IBISC, Evry, France
\textsuperscript{2} CITI and Departamento de Informática, FCT - Universidade Nova de Lisboa, Portugal

Abstract

In recent work \cite{6}, we have developed a session types discipline for adaptable processes—a core language for concurrency in which located processes may be stopped, duplicated, or discarded at runtime. Our aim is to understand whether known techniques for the static analysis of structured communications scale up to the challenging context of context-aware, adaptable distributed systems, in which disciplined interaction and dynamic reconfiguration are intertwined concerns. Our work is an initial step towards this general goal; it is also driven by the observation that analyses of structured communications could be more effective if grounded on process languages enhanced to capture emerging phenomena in distributed systems. We focus on dynamic reconfiguration, a crucial issue in, e.g., the cloud-based infrastructures in which distributed applications are deployed nowadays. In this short note, we summarize and illustrate the framework in \cite{6}.

1 Introduction

The type-theoretic foundations provided by session types have proved successful in the analysis of complex communication scenarios. This is witnessed by, e.g., the several extensions and enhancements of early session types proposals \cite{9} with common concepts/idioms in practical distributed programming. Interestingly, while often such extensions have appealed to increasingly sophisticated type structures (which feature, e.g., subtyping \cite{8}, dependent/indexed types \cite{4}, and kinding \cite{3}), the associated modeling language—the polyadic $\pi$-calculus—has remained essentially the same. This “asymmetric” development of session types and process languages should not appear as a surprise: because of its canonicity and expressiveness, the $\pi$-calculus represents a rather natural choice for representing concurrent systems which interact by following precise—yet intricate—behavioral patterns.

We are interested in one such patterns, namely dynamic reconfiguration: it allows to represent the suspension, replacement, or abortion of running interacting processes. As such, it can be useful to model and reason about important mechanisms in modern distributed systems, such as code mobility, online software update, failure recovery (as in, e.g., constructs for exceptions/compensations), and scaling (i.e., the ability of acquiring/releasing computing resources based on runtime demand). These mechanisms typically have a global effect over the system and are meant to be executed atomically. Hence, they appear difficult to implement as $\pi$-calculus specifications: it is not obvious how name passing/scope extrusion—the central abstraction vehicles of the $\pi$-calculus—can be used to model this kind of global reconfiguration primitives while retaining atomicity and an adequate level of abstraction for reasoning.

Based on these limitations, and with the aim of setting a formal framework for reasoning about communicating systems with dynamic reconfiguration, with Bravetti and Zavattaro we have developed a framework of adaptable processes \cite{1}. Adaptable processes extend usual process calculi with two constructs: located processes $l[P]$ and update processes $l\{U\}$, where $P$ is a process and $U$ is a context (a process with zero or more occurrences of a variable $X$). While located processes explicitly represent distributed interaction, update processes provide a built-in adaptation mechanism for located processes. More precisely, by synchronizing on name $l$, processes $l[P]$ and $l\{U\}$ may reduce to $U[P]$—the process
which results from substituting occurrences of $X$ in $U$ with $P$. This way, we obtain a mechanism for dynamic process reconfiguration which is performed atomically in a single reduction.

In recent work [6], we have started to explore the integration of a session-typed discipline into a $\pi$-calculus with located and update processes. The intention is to understand whether session types scale up to the challenging context of context-aware, adaptable distributed systems, in which disciplined communication and dynamic reconfiguration are intertwined concerns. Indeed, processes in our language may evolve either by the usual forms of synchronization but also by performing evolvability actions. Hence, besides showing that well-typed processes respect session types along computation, we have proved that the interplay of communicating and evolvability actions is consistent, i.e., an update action for a located process can only take place if such a process is not already involved in an active session. That is, consistency ensures that evolvability steps do not interfere with the session-typed protocols. Technically, our work builds upon the approach of Garralda et al. [7] on session types for Ambients, and extends it to the case of adaptable processes which run in arbitrary, nested locations.

In this short note, we summarize the main elements of the framework in [5] and illustrate it with an example. We also mention some topics for future work. The reader is referred to [1, 6] for details and motivation on untyped and typed adaptable processes, respectively.

2 A Framework of Disciplined, Adaptable Processes

Syntax and Semantics. We consider a minor extension of the usual polyadic $\pi$-calculus for binary session types (cf. [9]). Besides expected base sets for names $a, b, x, \ldots$, channels $c, d, \ldots$, labels $n, n', \ldots$, and constants $k, k', \ldots$, our process syntax relies on $\Delta$-calculus for binary process variables, ranged over $X, X', \ldots$; recursion variables, ranged over $Y, Y', \ldots$; and integers, ranged over $j, h, \ldots$. We use $u, u'$ to denote names and channels. Processes are thus given by the following grammar:

$$
\text{Expressions } e ::= k \mid e_1 + e_2 \mid e_1 - e_2 \mid \ldots
$$

$$
\text{Processes } P ::= \text{open } a(c : \rho_n).P \mid \tau(\overline{c}).P \mid c(\overline{x}).P \mid P \parallel P \\
| c^{\triangleright} \{n_1 : P_1, \ldots, n_k : P_k\} \mid c^{\triangleleft}\nu.P \mid c^{\downarrow}(d).P \mid c^{\uparrow}(d).P \\
| \text{rec}(Y : \Phi : \Delta).P \mid Y \mid \text{if } e \text{ then } P \text{ else } Q \mid \text{close}(c).P \mid (\nu a)P \mid 0 \\
| l^h[P] \ (h \geq 0) \mid X \mid l^\Delta_{\Delta_2}(P(X))
$$

The first four lines present an almost standard session-typed language with session passing (delegation) [10]. Differences are in presence of an explicit prefix for session closing, denoted close$(c)$, and in the session opening construct, which is annotated with a session type $\rho_n$. While the former is useful to count the number of active sessions in a given located process, the latter is related to the fact that session types in a global computing scenario represent decentralized service offerings. These differences were first proposed by Garralda et al. [7] in the context of session types for the Ambient calculus.

Constructs related to adaptable processes are given in the last line. We have located processes $l^h[P]$ and update processes $l^\Delta_{\Delta_2}(P(X))$, where $P(X)$ denotes a process with one or more occurrences of a variable $X$. Locations are transparent: hence, in $l^h[P]$ process $P$ may evolve autonomously until it is updated by an update action on $l$. One objective of our type system is then to discipline such update actions, so as to avoid updating processes which contain open sessions. To this end, processes are annotated: while in $l^h[P]$ integer $h$ denotes the number of active sessions in $P$, in $l^\Delta_{\Delta_2}(P(X))$ both $\Delta_1$ and $\Delta_2$ denote type-based interfaces—roughly, collections of session types. A process $l^\Delta_{\Delta_2}(P(X))$ will be able to update a located process with interface (compatible with) $\Delta_1$ thus resulting in a (reconfigured) process with interface $\Delta_2$. These annotations are central to the operational semantics, which is defined as a reduction relation. Fig. 1 reports some selected reduction rules; we use $C, D, E$ to denote (syntactic) contexts which represent the nested structure of located processes.
The semantics relies on a structural congruence relation (omitted), which handles scope extrusion for channels and names in the usual way. Rules (R:OPEN) and (R:CLOSE), which formalize session establishment and termination, resp., use two operations over contexts, denoted $C^+$ and $C^-$. Informally, $C^+$ (resp. $C^-$) denotes the context obtained from $C$ by increasing (resp. decreasing) the annotation $h$ in all of its located processes. Rule (R:UpD) captures the essence of our notion of consistency: a (located) process $Q$ can be updated only if (i) it contains no active sessions, and (ii) its interface (denoted $\Delta_Q$ in the rule) is compatible with the update process. This is denoted $\Delta_Q \vdash \Delta_1$. For the sake of generality, this compatibility relation is left unspecified.

**Type System and Main Results.** We have deliberately aimed at retaining a type structure close to standard session types presentations. Assuming a set of basic types, ranged over $\kappa, \kappa', \ldots$, we have:

| Session types $\zeta$ | ::= $\epsilon | \!\!(\tilde{k}).\zeta | \!\!(k).\zeta | \!\!(\sigma).\zeta | \!\!(\sigma).\zeta | t | \mu t.\zeta$ |
|-----------------------|-------------------------------------------------------------|
| Closed pre-session types $\rho$ | $\equiv \{\zeta | \text{fsv}(\zeta) = \emptyset\}$ |
| Session types $\omega$ | $\equiv \rho_\emptyset | \bot$ |

Session types are annotated with a linear or unrestricted qualification, denoted $q$: this is necessary to type process recursion. To formalize typing environments, we use interfaces $\Delta$, which are collections of closed pre-session types $\rho$. Intuitively, these represent available but not yet active sessions. 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. This way, a type judgment is of form

$$\Gamma; \Theta \vdash P : \Phi; \Delta$$

stating that, under environments $\Gamma$ and $\Theta$, process $P$ has active sessions declared in $\Phi$ and interface $\Delta$. We only present typing rules for adatable and update processes; see [6] for a full account.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Type System</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T:OPEN)</td>
<td>$\Gamma; \Theta \vdash P : \Phi; \Delta \quad h = #{c \mid c : \omega \in \Phi}$</td>
</tr>
<tr>
<td>(T:UPD)</td>
<td>$\Gamma; \Theta \vdash P \to \Phi; \Delta \quad h = #{c \mid c : \omega \in \Phi}$</td>
</tr>
</tbody>
</table>

While rule (T:OPEN) checks that the annotation for located processes corresponds to the number of open sessions, rule (T:UPD) ensures that the annotations for update processes correspond to the interface of the variable involved, and to that offered by the process.

We now summarize the main properties of well-typed processes. Some auxiliary definitions are required. Process $P$ is said to be communicating over channel $c$ if it sends (resp. receives) a value/session on $c$, if it makes a selection (resp. offers a choice), or if it closes a session on $c$. Then, two communicating processes are dual on session $c$ if they are complementary to each other, i.e., if one sends, the
other receives on c; if one makes a selection, the other chooses; or if they are closing the same session c. We write $Q(c)$ and $Q^*(c)$ for two dual communicating processes on c. Also, let $\rightarrow$, $\rightarrow'$ stand for an update action, i.e., a reduction inferred using rule (R:UPD) (possibly with the interplay of restriction/structural congruence rules). A session on channel c is consistent in a process P if, for all $P',P''$ such that

$$P \rightarrow \nu{E}(E'[Q(c)] || D[Q^*(c)])$$

and $P' \rightarrow P''$, then $\exists E',C',D'$ such that $P'' \equiv (\nu{E})(E'[C'[Q(c)] || D'[Q^*(c)])$. Hence, intuitively, consistency formalizes the fact that reductions corresponding to update actions do not disrupt the behavior of communicating processes in already active sessions. They can only involve parts of the system not engaged into active sessions.

In [6] we have shown that all sessions in our well-typed processes are consistent. This result follows from the subject reduction theorem below: i.e., well-typedness is preserved by reduction.

**Theorem 1 (Subject reduction).** If $\Gamma; \Theta \vdash P :: \Phi; \Delta$ and $P \rightarrow Q$ then one of the following holds:

1. $\Gamma; \Theta \vdash Q :: \Phi; \Delta'$
2. $\Gamma; \Theta \vdash Q :: \Phi; \Delta'$, for some $\Delta'$;
3. $\Gamma; \Theta \vdash Q :: \Phi, c :: \bot; \Delta'$, for some $\Delta' \subset \Delta$ and channel c.

The above items correspond to the different possibilities for reduction. The main consequence of Thm.1 is the absence of communication errors for well-typed processes. It also allows us to prove that every session that is established along the evolution of a process is consistent. We can indeed state:

**Corollary 1 (Consistency by Typing).** Suppose $\Gamma; \Theta \vdash P :: \Phi; \Delta$ is a well-typed process. Then every session $p_q \in \Delta$ is consistent.

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 $P_q$, as defined above. Indeed, our type system ensures that the annotations enabling update actions are correctly assigned and maintained along reduction.

### 3 Example: Dynamic Update in Workflow Applications

We now discuss a simple model of a workflow application, which extends the example given in [1]. Our model combines the main features of adaptable processes (nested locations and update processes) with delegation, the well-known instrument to represent reconfiguration in session-typed processes.

Briefly, a workflow is a conceptual unit that describes how a number of activities coordinate to achieve a given task. A workflow-based application usually consists of a workflow runtime engine that contains a number of workflows running concurrently on top of it; a workflow base library on which activities may rely on; and of a number of runtime services, which are typically application dependent. Exploiting nested, located processes, we may abstract a workflow application as the following process:

$$App \triangleq wfa^0[we^0[|W_1 \parallel W_k \parallel wb^0[|BL] ] \parallel S_1 \parallel \cdots \parallel S_j ]$$

where the application is modeled as an located process $wfa$ which contains a workflow engine (located at we) and a number of runtime services $S_1,\ldots,S_j$. In turn, the workflow engine contains a number of workflows $W_1,\ldots,W_k$, a process $WE$ (which represents the engine’s behavior and is left unspecified), and a located process $wb$ representing the base library (also left unspecified). A single workflow $W_i$ is composed of a number of activities: it is modeled as a located process $wi$ containing a process $WL_i$. 


—representing the workflow’s logic—, and \( n \) activities, which are located process on names \( a_j \):

\[
W_i \triangleq w^0_i [WL_i \parallel \prod_{j=1}^{n} a_j|P_j]}
\]

We focus on the \( i \)-th workflow. Suppose that it has two activities only, located on names \( r \) and \( t \). That is, \( W_i \triangleq w^0_i [WL_i \parallel r^0_i|P_1] \parallel t^0_i|P_2] \), with \( P_1 \) defined as

\[
P_1 \triangleq \text{open} a(s : \sigma).s(u, p).s \triangleq n_1.Q[s].\text{close}(s)
\]

and where \( \sigma \) describes the session on \( s \) (i.e., the input, the selection, and the unspecified behavior represented by \( Q[s] \)). We now illustrate a simple reconfiguration policy for \( r^0_i|P_1 \), expressible in our language. Suppose \( WL_i \triangleq Rec \), where \( Rec \) is a reconfiguration policy for activity \( r \), defined as follows:

\[
\begin{align*}
Rec & \triangleq \text{if } e \text{ then } r^0_i \sigma \{v^0_1[X] \} \text{ else } r^0_j \sigma \{r^0_i|P' \parallel r^0_j|P''\} \text{ where:} \\
P' & \triangleq \text{open} a(s : \sigma).\text{open} b(d : !(\sigma)).d!\langle s \rangle.\text{close}(d) \\
P'' & \triangleq \text{open} b(e : ?(\sigma)).e?\langle s \rangle.\text{open}\langle s \rangle.\text{close}(s) \text{ close}(e)
\end{align*}
\]

Thus, \( Rec \) formalizes a reconfiguration policy that depends on the boolean value \( e \) evaluates to: if \( e \) reduces to true, then the behavior at activity \( r \) will be kept unchanged, but it will be relocated to name \( v \):

\[
W_i \longrightarrow^* w^0_i [v^0_1|P_1] \parallel t^0_i|P_2]
\]

Otherwise, if \( e \) evaluates to false, an update action on \( r \) will reconfigure the distributed structure of \( P_1 \):

\[
W_i \longrightarrow^* w^0_i [r^0_i|P' \parallel r^0_j|P''] \parallel t^0_i|P_2]
\]

Thus, in a single step, the monolithic service \( P_1 \) will be replaced with a more flexible implementation in which \( P' \) (located at \( r_1 \)) first establishes a session and then delegates it to \( P'' \) (located at \( r_2 \)). This update action is transparent to any user of the first activity of \( W_i \); it is possible because the interface of \( P_1 \) (i.e., \( \sigma \)) coincides with the annotation of the update process. Also, observe that interface \( \Delta \) contains entries for session types \( !(\sigma) \) and \( ?(\sigma) \), which are declared in \( P' \) and \( P'' \), respectively. Finally, notice that in (1) there is an unbalance in the number of open and close prefixes located at \( r_1 \) and \( r_2 \) which may appear puzzling. This unbalance is due to delegation; upon session establishment associated annotations are treated consistently thanks to Rule (R:PASS) in Fig. [1]

Clearly, more interesting reconfiguration policies can be expressed in our framework. The above scenario already illustrates how our framework extends the expressiveness of session-typed languages with dynamic reconfiguration. Typing not only ensures correct communications between \( P \) and its partner on channel \( s \); by relying on Cor. [1] we know that well-typedness also implies session consistency, i.e., an update action on \( r \) performed by \( Rec \) will not occur if \( P_1 \) has already established a session.

4 Concluding Remarks and Future Work

We have presented the main elements for a simple integration of session types into adaptable processes, a process calculi framework for specifying interacting processes which may be suspended, updated, or discarded at runtime. Our work appears to be the first attempt in addressing the integration of dynamic evolution issues in models of communicating systems in which interaction is described by session types. As we have explained above, the work summarized here is only an initial step towards that goal; there are several open issues we intend to explore in future work.
- We would like to investigate progress (deadlock freedom) for adaptable, session-typed processes. This includes adapting to our setting known session type systems for ensuring progress [5,2], but also understanding whether the information added by such systems (e.g., orderings on session events) can be combined with appropriate update actions to prevent/overcome deadlocked sessions at runtime.

- Our nested, transparent locations may be too “open” for some applications. In fact, we would like to have ways of enforcing secure updates, so as to protect located processes from unintended evolvability actions—as in, for instance, update actions triggered by unauthorized users, or update actions based on permissions for replacing, extending, or destroying the behavior of a located process.

- In this presentation, we have refrained from enriching the typing system with information related to updates. We think such an extension would not be difficult, and could offer a way to better control update actions. This way, for instance, one could decree (and statically check) that update actions occur only in selected parts of a communication protocol.

Acknowledgments We are very grateful to the anonymous reviewers: their comments and remarks were most useful in improving the presentation of this short note. This research was supported by the Portuguese Foundation for Science and Technology (FCT) under grants SFRH / BPD / 84067 / 2012 and CITI, and by French project ANR BLAN 0307 01 - SYNBIOTIC.

References


