Session Types for Orchestrated Interactions[*]

Franco Barbanera

Dipartimento di Matematica e Informatica
University of Catania
`barba@dmi.unict.it`

Ugo de'Liguoro

Dipartimento di Informatica
University of Torino
`ugo.deliguoro@unito.it`

In the setting of the π-calculus with binary sessions, we aim at relaxing the notion of duality of session types by the concept of *retractable compliance* developed in contract theory. This leads to extending session types with a new type operator of "speculative selection" including choices not necessarily offered by a compliant partner. We address the problem of selecting successful communicating branches by means of an operational semantics based on orchestrators, which has been shown to be equivalent to the retractable semantics of contracts, but clearly more feasible. A type system, sound with respect to such a semantics, is hence provided.

1 Introduction

Contracts [15, 16, 11, 7] and session types [13, 19] are both intended as abstractions representing interaction protocols among concurrent processes. In both theories, interaction is modeled by message exchange along channels, abstractly represented by input/output actions indexed over channel names. Also the used formalisms all stem from CCS and its variants, possibly extended by some value passing mechanism. The resemblance is even tighter when considering "session contracts" [1, 7], where only internal and external choices among contracts prefixed by pairwise distinct output viz. input actions are respectively allowed.

Contracts versus Session Types. In spite of similarities, these theories stem from rather different concepts. In case of contracts, the input/output behaviour of a participant to a conversation is formalised as whole by a term of an appropriate process algebra; contract theory then focuses on the "compliance" relation, holding when two or more protocols are such that, whenever there is an action by a participant that is expected to be performed, the symmetric namely dual one is made available by some other participant. Restricting to the binary case, we say that a "server" protocol is compliant with a "client" one if all actions issued by the latter are matched by the respective co-actions by the former, possibly until the client reaches a successful state.

Session types are a type system for a dialect of Milner's π-calculus. Like with typed π-calculus, judgments associate to a process a "typing" that pairs channel names with the types of the values that can be transmitted through the channels; since π-terms are allowed to communicate channel names as well, channel types are among the types of exchanged values. Differently from ordinary π-calculus types, session types are regular trees of value types, that can be session types as well. In this way a single type can describe the flow of data through each channel that do not need to have all the same type; also input/output communication actions are distinguished by their types. When a "session" is opened by two processes in parallel, a new private channel is created - the session channel - that is shared by the processes; if the two "end-points", namely the respective occurrences of the session channel in the

---

participant processes, are typed by *dual* types - roughly interchanging input and output types - then the interaction will be error free at run-time. Observe that not only the typing has to be checked against the process structure, which is not considered in contract theory, but also the very same process can issue several sessions at the same time, and session channels can be exchanged among processes. Therefore, even in the simpler setting of binary sessions, namely with channel names connecting two processes at a time, difficulties arise from the possible nesting of several sessions, and their ability to communicate across the boundary of a single session.

Orchestrated Compliance. Compliance being a rather restrictive requirement, more liberal constraints have been proposed in the literature, among which are "orchestrated" compliance [18] and "retractable" contracts [4]. According to the orchestrated model the interactions between a client and a server are mediated by a third process - the *orchestrator* - ruling interactions by allowing certain actions and co-actions only, possibly buffering a bounded number of messages on both sides. In the retractable model instead, actions are classified into irretractable (unaffectible) and retractable (affectible) ones. The concept is that, while irretractable actions by a participant have to be matched with their duals by the other compliant participant, retractable actions are just tried and possibly retracted, in case of communication failure, to issue some other action instead. Although these two models are different, it has been shown in [3] that, by restricting to certain orchestrators that allow just synchronous communications, contracts that are deemed compliant in both models are the same. Moreover, it is possible to provide an algorithm which synthesizes an orchestrator out of two retractable contracts if they are compliant, or fails otherwise (see [6] and section 6 below).

Orchestrated interaction instead of duality. In this paper we address the issue of adapting the idea of retractable contracts to session types. More precisely we see session types as contracts, and propose to replace the otherwise restrictive notion of type duality by the relation of retractable or equivalently orchestrated compliance. To better illustrate the point, let us consider a process P in parallel with a system Q, that has to choose how to interact in a session with Q by selecting one of several alternatives. Both P and Q are equipped with specification of their behaviours, so that it is known in advance that at least one alternative is actually successful, but not necessarily all of them are. Now there are two possible ways of guaranteeing P to successfully complete the interaction with Q:

- either $P \mid Q$ is run on a computational infrastructure that, in case of a synchronization failure, allows to roll back to some previous choice point $P' \mid Q'$, and to try a different branch of the interaction;

- or, when checking the compliance of the specifications, it is statically computed which are the safe choices, if any, *before* running $P \mid Q$, so that they can be stored into a mediating process.

The reason for preferring the second approach is clearly apparent, as it limits the backtracking to static type checking, while avoiding it at run-time. In fact, once types of opposite end-points have been recognized compliant up to retractability of certain choices (that are kept distinct syntactically from unretractable ones), and an orchestrator f has been sinthesized, the very same orchestrator can serve as guidance in the interaction on the session channel. In particular, by putting the orchestrator f in parallel with the processes holding the end-points, it can be used to drive the proper choices at run time. If k is the session channel, we write the resulting session by:

$$(\nu k)(\langle k\rangle f \mid P \mid Q)$$

representing that the interaction over the private session channel k is ruled by f.

The orchestrator is obviously an abstraction that allows for many different implementations. Among the possible ones there are:

1. a communication infrastructure responsible of running the orchestrator;

2. the two partners agree on an orchestrator that it is used on both sides as a communication interface.

We do not further discuss implementation issues that would lead us outside the scope of the present paper.

Speculative selection. In session-types formalism the type $\oplus\{l_1{:}S_1,\ldots,l_n{:}S_n\}$ describes the protocol consisting in selecting the label l_i to be sent as output, and continuing as specified in S_i; this is the consequence of an internal choice that is transparent to the other participant in the session, that is expected to be able to react to all $l_1,\ldots,l_n$. The dual is the branching type $\&\{l_1{:}S'_1,\ldots,l_n{:}S'_n\}$, expecting a label among $l_1,\ldots,l_n$ as input to continue according to the respective continuation. To these we add a new type constructor written

$$\boxplus\{l_1{:}S_1,\ldots,l_n{:}S_n\}$$

that we dub *speculative selection type*. The intended meaning of speculative selection is: try selecting labels among $l_1,\ldots,l_n$ until an l_i is found such that $l_i : S'_i$ is in the corresponding branching type, and S_i and S'_i are compliant. Observe that the speculative selection type has no dual, so that we cannot use the notion of duality in the system.

As a running example suppose that a *Client* is willing to establish a session with a movie-*Provider* and to behave on her channel end according to the following session type:

$$\mathsf{ClientSess} \;=\; ![\texttt{String}].\oplus\{\textsc{buy}{:}\oplus\{\textsc{uhd}{:}\mathsf{S},\textsc{hd}{:}\mathsf{S}\},\ \textsc{rent}{:}\boxplus\{\textsc{uhd}{:}\mathsf{S},\textsc{hd}{:}\mathsf{S},\textsc{sd}{:}\mathsf{S},\textsc{ld}{:}\mathsf{S}\}\}$$

where $\mathsf{S} \;=\; ![\texttt{String}].\&\{\textsc{ok}{:}?[\texttt{Url}],\textsc{no}{:}\texttt{end}\}$. Accordingly the client will send on the session channel a login information (an element of the ground type `String`) and then will internally decide whether she intends to BUY or to RENT a movie. In the former case, she will further decide whether to buy an ultra-high-definition (UHD) or a high-definition (HD) movie. In the latter case, instead, it is stated in the speculative selection type $\boxplus\{\textsc{uhd}{:}\mathsf{S},\textsc{hd}{:}\mathsf{S},\textsc{sd}{:}\mathsf{S},\textsc{ld}{:}\mathsf{S}\}$ that she will proceed according to four possible failure-amenable choices: renting an ultra-high-definition (UHD), a high-definition (HD), a standard-definition (SD) or a low-definition (LD) movie. In all cases, she will proceed according to type S by sending the string with the title of the movie, and either receiving the URL from which the movie can be downloaded, if available (availability corresponds to the reception of OK), or ending the session if it is not (NO).

From the previous discussion, if the *Client* behaves on her end-point of the session channel according to the type ClientSess, she can safely interact with the *Provider* in case the latter behaves on the other end-point of the session channel according to the following session type ProvSess:

$$\mathsf{ProvSess} \;=\; ?[\texttt{String}].\&\{\textsc{buy}{:}\&\{\textsc{uhd}{:}\mathsf{S}',\textsc{hd}{:}\mathsf{S}'\},\ \textsc{rent}{:}\&\{\textsc{hd}{:}?[\texttt{Nat}].\mathsf{S}',\textsc{sd}{:}\mathsf{S}',\textsc{ld}{:}\mathsf{S}'\}\}$$

where $\mathsf{S}' \;=\; ?[\texttt{String}].\oplus\{\textsc{ok}{:}![\texttt{URL}],\textsc{no}{:}\texttt{end}\}$.

Now this client/server interaction succeeds in case the client actually buys a movie. If the client intends to rent a movie, instead, the choice UHD will produce a synchronization failure, since no ultra-high-definition movies are for rent on that server. Also the choice HD of a movie to be rent would produce a syncronization failure, since then the server requires a membership code (as described by HD:?[Nat].S'). But this is not all the story, as the remaining two possibilities still lead to success, so that we insist that these two participants can agree at least in part. Indeed to prevent synchronization failures the *Client* will be instructed at run-time by the orchestrator to select either SD or LD choice only while renting a movie. The orchestrator is computed when the session between the client and the server is tried and possibly opened, hence before it is started. An orchestrator for the example is:

$$\mathsf{f} \;=\; \bullet.\big((\textsc{buy}.(\textsc{uhd}.\mathsf{f}' + \textsc{hd}.\mathsf{f}')) + (\textsc{rent}.(\textsc{sd}.\mathsf{f}' \oplus \textsc{ld}.\mathsf{f}'))\big)$$

where $f' = \bullet.((\textsc{ok}.\bullet)+\textsc{no})$. This means that on the session channel between *Client* and *Provider*, the orchestrator f first enables an input/output interaction ($\bullet$) and then either ($+$) a BUY or a RENT branching. In case of RENT, the orchestrator internally decides ($\oplus$) to force either an SD or LD choice; this can be left open as both of them are "safe", namely do not lead to (not even future) synchronization failures.

Observe that also the following two orchestrators can successfully drive the interaction between *Client* and *Provider*:

$$f_1 = \bullet.((\textsc{buy}.(\textsc{uhd}.f'+\textsc{hd}.f'))+\textsc{rent}.\textsc{sd}.f'), \quad f_2 = \bullet.((\textsc{buy}.(\textsc{uhd}.f'+\textsc{hd}.f'))+\textsc{rent}.\textsc{ld}.f').$$

As a matter of fact, our calculus and its type system do consider any safe orchestrator for a session interaction. In actual implementations, however, one could be interested in limiting the nondeterminism in a session exclusively to that exposed by the two partners. This involves considering *deterministic* orchestrators only, like f_1 and f_2 above, that is with no $\oplus$ inside.

Adding priorities. For the semantics of the type $\boxplus\{\textsc{uhd}{:}\mathsf{S},\textsc{hd}{:}\mathsf{S},\textsc{sd}{:}\mathsf{S},\textsc{ld}{:}\mathsf{S}\}$ the actual ordering of the labels is immaterial; therefore the actual choice stored in the orchestrator is up to the synthesis algorithm. In practice, however, the choice of either f_1 or f_2 should not be randomly determined, rather it might reflect a particular policy representable at type level. A simple way is to provide a priority ordering among the failure-amenable choices, that can be expressed by the following modified syntax:

$$\boxplus\langle\!\langle\textsc{ld}{:}\mathsf{S},\textsc{sd}{:}\mathsf{S},\textsc{hd}{:}\mathsf{S},\textsc{uhd}{:}\mathsf{S}\rangle\!\rangle$$

where $\langle\!\langle\ \rangle\!\rangle$ is now an ordered list. So the above type expresses that the option LD is the one liked best, but in case of failure, SD is the second preferred choice, and so on. The priorities represented by the above speculative selection type force the orchestrator f_2 to be synthesized.

We shall show that it is possible to synthesize the deterministic orchestrator which reflects the priorities described in the speculative selection types. Moreover, with an extra computational effort, the priority-ordering policy can be lowered down and described at process level.

According to the compliance relation, in a session only the client's communicating actions are guaranteed to be matched by corresponding actions on the server's side. Therefore there might be some pending communications on the server side even in case the client has successfully completed. This fact produces particular deadlocks which do not show up in ordinary session-based calculi and type systems, as these are based on the notion of duality. We show how to get rid of such stuck states by adding suitable reduction rules. The type system can in fact be proved to be sound w.r.t. the new reductions, preventing typed process from reaching these peculiar stuck states.

In order to focus on the main concepts, in the present paper we do not treat recursion, that can be easily added in a fairly standard way.

Structure of the paper. In Section 2 we define types, orchestrators and the relation of orchestrated compliance. The syntax of the calculus and the type system are treated in Section 3. In Section 4 the operational semantics is defined and the subject reduction property is proved, obtaining error freeness of typed processes as a corollary. The particular deadlocks due to the use of the compliance relation instead of duality is dealt with in Section 5, whereas in Section 6 we restrict orchestrators to deterministic ones, and make them implement a priority selection policy which can be described either at type or process level. Section 7 contains the conclusions and suggests possible extensions.

2   Session types and orchestrated compliance

First we introduce session types following [13] but for recursion (omitted for sake of simplicity) and for the new type $\boxplus\{l_i{:}S_i\}_{i\in I}$ for speculative selection, corresponding to retractable choice in [4].

Definition 2.1 (Types).

$$
\begin{array}{rcll}
G & := & \texttt{Nat} \mid \texttt{Bool} \mid \ldots & \text{ground types}
\end{array}
$$

$$
\begin{array}{rcll}
S & := & & \text{session types}
\end{array}
$$

	end	*terminated*	
	$?[G].S$	*value input*	$\mid\ ![G].S$ *value output*
	$?[S_1^p].S_2$	*session input*	$\mid\ ![S_1^p].S_2$ *session output*
	$\oplus\{l_i{:}S_i\}_{i\in I}$	*selection*	$\mid\ \&\{l_i{:}S_i\}_{i\in I}$ *branching*
	$\boxplus\{l_i{:}S_i\}_{i\in I}$	*speculative selection*	

$$
\begin{array}{rcll}
T & := & G \mid S^p & \text{I/O types}
\end{array}
$$

where $p \in \{+,-\}$. Moreover, the labels l_i's belong to a countable set of labels $\mathscr{L}$ and are pairwise distinct in branching, selection and speculative-selection types.

The syntax of *orchestrators* is inspired by that in [18] and [6]. Main differences with respect to [18] and [6] are that we do not have buffers, here unnecessary because we do not model asynchronous communications; besides, our orchestrators can introduce some nondeterminism in orchestrated interactions (see Definition 4.2).

Definition 2.2 (Orchestrators). *We define the set* Orch *of orchestrators, ranged over by* f, g,..., *as the terms generated by the following grammar:*

$$
\begin{array}{rcll}
\textsf{f},\textsf{g} & ::= & \mathbb{1} & \textit{idle} \\
 & \mid & \bullet.\textsf{f} & \textit{I/O prefix} \\
 & \mid & l.\textsf{f} & \textit{selection prefix} \\
 & \mid & l.\textsf{f} + l'.\textsf{g} \quad (l \neq l') & \textit{external choice} \\
 & \mid & l.\textsf{f} \oplus l'.\textsf{g} \quad (l \neq l') & \textit{internal choice}
\end{array}
$$

where $l,l' \in \mathscr{L}$.

We write $\sum_{i\in I} l_i.\textsf{f}_i$ (resp. $\oplus_{i\in I} l_i.\textsf{f}_i$) for $l_1.\textsf{f}_1 + \cdots + l_n.\textsf{f}_n$ (resp. $l_1.\textsf{f}_1 \oplus \cdots \oplus l_n.\textsf{f}_n$), where $I = \{1,\ldots,n\} \neq \emptyset$ and the l_i's are pairwise distinct. If I is a singleton then $\sum_{i\in I} l_i.\textsf{f}_i$ ($\oplus_{i\in I} l_i.\textsf{f}_i$) is just a selection prefix.

Definition 2.3 (Orchestrated compliance). *The relation* $\textsf{f} : S \dashv S'$ *among the orchestrator* f *and session types S,S' is the least one such that:*

1. $\mathbb{1} : \text{end} \dashv S$, *for any S,*

2. *if* $\textsf{f} : S \dashv S'$ *then* $\bullet.\textsf{f} : ?[G].S \dashv ![G].S'$ *and* $\bullet.\textsf{f} : ![G].S \dashv ?[G].S'$ *for any G,*

3. *if* $\textsf{f}_i : S_i \dashv S'_i$ *for all $i \in I$ then, for any set of indexes J,*
   $\sum_{i\in I} l_i.\textsf{f}_i : \oplus\{l_i{:}S_i\}_{i\in I} \dashv \&\{l_j{:}S'_j\}_{j\in I\cup J}$ *and* $\sum_{i\in I} l_i.\textsf{f}_i : \&\{l_j{:}S_j\}_{j\in I\cup J} \dashv \oplus\{l_i{:}S'_i\}_{i\in I}$,

4. *if* $H \subseteq I \cap J$ *with $H \neq \emptyset$ and $\textsf{f}_h : S_h \dashv S'_h$ for all $h \in H$ then*
   $\oplus_{h\in H} l_h.\textsf{f}_h : \boxplus\{l_i{:}S_i\}_{i\in I} \dashv \&\{l_j{:}S'_j\}_{j\in J}$ *and* $\oplus_{h\in H} l_h.\textsf{f}_h : \&\{l_j{:}S_j\}_{j\in J} \dashv \boxplus\{l_i{:}S'_i\}_{i\in I}$.

We say that S and S' are **compliant***, written $S \dashv S'$, if* $\textsf{f} : S \dashv S'$ *for some* f.

Proposition 2.4. *Given S,S' it is decidable whether $S \dashv S'$. Moreover if $S \dashv S'$ then an orchestrator* f *such that* $\textsf{f} : S \dashv S'$ *is computable.*

Proof. By induction over the structure of S, S'.   □

Remark 2.5. *Both Definition 2.3 and Proposition 2.4 easily extend to the case of (contractive) recursive types and orchestrators (see [6], where orchestrated compliance is defined for "session contracts" instead of types).*

Remark 2.6. *In the theory of session contracts [2, 7], compliance does correspond to the composition of duality and subtyping. Such a correspondence does transfer also to session types in a very general sense, as shown in [8]. In our setting, even if a relation of subtyping can be obtained out of a restriction of the subcontract relation defined in [5], the above mentioned correspondence looks unrealistic since, as pointed out in [4], there exists no natural notion of duality[1] for retractable contracts.*

Example 2.7. Extending the running example in the Introduction, we include the higher-order features of session types. We assume that the *Client* of the movie-*Provider* has to pay for the buyed/rented movie. For the payment, if the movie is available, the *Provider* throws to the *Client* a session channel (typed by the session PAY) that can be used to pay using several possible cards. The new versions of ClntSess and ProvSess are now, respectively:

$$\mathsf{ClntSess} = ![\mathtt{String}].\oplus\{\textsc{buy}{:}\oplus\{\textsc{uhd}{:}\mathsf{S}, \textsc{hd}{:}\mathsf{S}\}, \textsc{rent}{:}\boxplus\{\textsc{uhd}{:}\mathsf{S}, \textsc{hd}{:}\mathsf{S}, \textsc{sd}{:}\mathsf{S}, \textsc{ld}{:}\mathsf{S}\}\}$$

where $\mathsf{S} = ![\mathtt{String}].\oplus\{\textsc{ok}{:}?[\mathtt{PAY}^-].?[\mathtt{Url}], \textsc{no}{:}\mathtt{end}\}$

and

$$\mathsf{ProvSess} = ?[\mathtt{String}].\&\{\textsc{buy}{:}\&\{\textsc{uhd}{:}\mathsf{S}', \textsc{hd}{:}\mathsf{S}'\}, \textsc{rent}{:}\&\{\textsc{hd}{:}?[\mathtt{Nat}].\mathsf{S}', \textsc{sd}{:}\mathsf{S}', \textsc{ld}{:}\mathsf{S}'\}\}$$

where $\mathsf{S}' = ?[\mathtt{String}].\oplus\{\textsc{ok}{:}![\mathtt{PAY}^-].![\mathtt{Url}], \textsc{no}{:}\mathtt{end}\}$

Here we assume that the payment always succeedes and the *Client* is not cheating. The safe interaction between *Client* and the *Provider* is guaranteed by ClntSess ⊣ ProvSess. In fact, for instance, it can be checked that:

$$\mathsf{g} : \mathsf{ClntSess} \dashv \mathsf{ProvSess}$$

where $\mathsf{g} = \bullet.((\textsc{buy}.(\textsc{uhd}.\mathsf{g}' + \textsc{hd}.\mathsf{g}')) + (\textsc{rent}.(\textsc{sd}.\mathsf{g}' \oplus \textsc{ld}.\mathsf{g}')))$ with $\mathsf{g}' = \bullet.((\textsc{ok}.\bullet.\bullet) + \textsc{no})$.

For the payment the *Provider* establishes a session with the *Bank*, hence acting as a client. The *Provider*'s end of the corresponding session channel is typed by:

$$\mathsf{bankCustSess} = ![\mathtt{Amount}].\mathsf{PAY}$$

where $\mathsf{PAY} = \boxplus\{\textsc{diners}{:}![\mathtt{ccNumber}], \textsc{mcard}{:}![\mathtt{ccNumber}], \textsc{visa}{:}![\mathtt{ccNumber}]\}$

(notice that PAY is precisely the type of the channel-end that the *Client* receives from the movie-*Provider* during the interaction described by ClntSess).

Hence, once the *Provider-Bank* session is established, the *Provider* sends the cost of the movie and then delegates the actual payment to the *Client*. The polarity '−' in the channel-send of ProvSess indicates that the delegated channel's end is sent by the applicant of the session. Therefore the receiver has to act as such [2].

On the delegated channel the *Client* is allowed to pay either by using a DINERS, a MASTERCARD or a VISA, but it is not guaranteed that all of them will be actually available.

We also assume that once the *Provider-Bank* session is established, the *Bank* behaves on its channel's end according to the type

$$\mathsf{BankSess} = ?[\mathtt{Amount}].\&\{\textsc{discover}{:}\mathsf{bS}, \textsc{mcard}{:}\mathsf{bS}, \textsc{visa}{:}\mathsf{bS}, \textsc{aexpr}{:}\mathsf{bS}\}$$

where $\mathsf{bS} = ?[\mathtt{ccNumber}].![\mathtt{TransIDnum}]$. Observe that the *Bank* can accept any of the DISCOVER,

---

[1] Duality for retractable session contracts can be immediately recovered by extending the formalism of [4] with speculative input choices, as done in [5]. Such extension, however, do not seem to have a clear session-type counterpart.

[2] The polarities of two corresponding channel-send and channel-receive are always the same, since they both refer to the same channel's end; in particular, in the present example the polarity is '−' also in the channel-receive of ClntSess.

MCARD, VISA or AEXPR cards, but does not accept DINERS. Moreover, after receiving the credit-card number, the *Bank* does issue the identifier of the transaction (an element of the ground type `TransIDnum`), notwhithstanding it is not requested by the `bankCustSess` session type. Nonetheless the safeness of the interaction with the *Bank* is guaranteed by the fact that `bankCustSess ⊣ BankSess`. In fact, for instance, it can be checked that

$$\mathsf{h} : \mathsf{bankCustSess} \dashv \mathsf{BankSess}$$

where $\mathsf{h} = \bullet.((\mathrm{MCARD}.\bullet) \oplus (\mathrm{VISA}.\bullet))$.

3 Calculus and type assignment

We assume to have a countable set $\mathscr{K}$ of *channel names*, ranged over by $k, k', \ldots$ In a session, we refer to the process performing the session-opening `request` as "the client", while the process which `accepts` it is referred to as "the server".

We distinguish among user-defined and run-time processes. While the former represent the code of concurrent programs, the latter formalise the state of the system at run-time. Following [12, 19], clients' and servers' channel-ends are identified by means of polarities $-$ and $+$ respectively, ranged over by $p, q, \ldots$ Moreover, we define $k^{\overline{+}} = k^-$ and $k^{\overline{-}} = k^+$.

Definition 3.1 (Processes). *The set of (user-defined)* processes *is defined as the set of the* closed *expressions generated by the following grammar:*

P, Q	:=	$\mathbf{0}$	*terminated*	$\mid (P \mid Q)$	*parallel*
		$\mid \mathtt{request}_S(k)P$	*session request*	$\mid \mathtt{accept}_S(k)P$	*session accept*
		$\mid k![e].P$	*value send*	$\mid k?(x).P$	*value receive*
		$\mid \mathtt{throw}\,k[k'].P$	*channel send*	$\mid \mathtt{catch}\,k(k').P$	*channel receive*
		$\mid k \triangleleft l.P$	*selection*	$\mid k \triangleright \{l_i{:}P_i\}_{i \in I}$	*branching*
		$\mid k \triangleleft [l_i{:}P_i]_{i \in I}$	*speculative selection*		

where $k, k' \in \mathscr{K}$, I is non-empty and finite, and the labels in branching and speculative selection, all belonging to $\mathscr{L}$ (as the one in selection), are pairwise distinct.

Let P be any user defined process. Run-time processes *are defined by the grammar*

$$R ::= (\nu k)(\langle k \rangle \mathsf{f} \mid P) \mid (\nu k)(\langle k \rangle \mathsf{f} \mid R) \mid (P \mid R) \mid (R \mid P) \mid (R \mid R')$$

where channels can be polarized channels*, i.e. channel names can be decorated with polarities. In $(\nu k)(\langle k \rangle \mathsf{f} \mid P)$ and $(\nu k)(\langle k \rangle \mathsf{f} \mid R)$ the operator (νk) is a binder of k^p, $k^{\overline{p}}$ and k.*

In session type systems the compatibility of processes establishing a session lays on the notion of duality of the respective types, that are associated to a port name which is the same on both sides. To duality we have replaced the notion of compliance. It is hence not straightforward to use port names for establishing a session between a client and a suitable server, since there is not a unique type for the possible servers of a given client. So, in order to get a calculus as general and simple as possible, we equip the `request` and `accept` operators with their types, without any reference to port names.

Most process actions are similar to those of the calculus in [13], but for the new *speculative selection*. A process $k \triangleleft [l_i{:}P_i]_{i \in I}$ is able to send on channel k any of the labels l_i's and to proceed afterwards as P_i. The process is aware that some of the synchronizations on the l_i's could led to a failure and an orchestrator is hence expected to drive the choice.

The processes *Provider*, *Client* and *Bank* of the running example can be described as follows (where also conditional processes are used and where only the parts concerning rental are described).

Example 3.2. Let **b** be a boolean expression representing the decision of whether buying or renting a movie.

$$Client = \texttt{request}_{\mathsf{ClntSess}}(k)k![\text{loginfo}].\texttt{if } \mathbf{b} \texttt{ then } k{\triangleleft}\text{BUY}.C_{\text{B}} \texttt{ else } k{\triangleleft}\text{RENT}.C_{\text{R}}$$

where

$$C_{\text{R}} = k{\triangleleft}[\text{UHD}.C', \text{HD}.C', \text{SD}.C', \text{LD}.P'] \qquad C' = k![\text{"}zootropolis\text{"}].k{\triangleright}\{\text{OK}: C'_{\text{OK}}, \text{NO}:\mathbf{0}\}$$
$$C'_{\text{OK}} = \texttt{catch}\, k(k').k'{\triangleleft}[\text{DINERS}:Q, \text{MCARD}:Q, \text{VISA}:Q] \qquad Q = k'![1234].k?(url).Watch$$

$$Provider = \texttt{accept}_{\mathsf{ProvSess}}(k)k?(x).k{\triangleright}\{\text{BUY}:P_{\text{B}}, \text{RENT}:P_{\text{R}}\}$$

where

$$P_{\text{R}} = k{\triangleright}\{\text{UHD}.P', \text{HD}.P', \text{SD}.P', \text{LD}.P'\} \qquad P' = k?(y).\texttt{if } \text{available}(y) \texttt{ then } k{\triangleleft}\text{OK}.P'_{\text{OK}} \texttt{ else } k{\triangleleft}\text{NO}$$
$$P'_{\text{OK}} = \texttt{request}_{\mathsf{bnkCustSess}}(k')k'![\text{amount}(y)].\texttt{throw}\, k[k'].k![\text{url}(y)]$$

$$Bank = \texttt{accept}_{\mathsf{BankSess}}(k')k'?(x).k'{\triangleright}\{\text{DISCOVER}:B, \text{MCARD}:B, \text{VISA}:B, \text{AEXPR}:B\}$$

where $B = k'?(cc).k'![\text{IDtrans}(x, cc)]$

The Type System The following type system is the "more liberal" system in [19], where duality is replaced by the relation of orchestrated compliance. There are two kinds of judgments: the first one is $\Gamma \vdash e : G$, where e is an expression of ground type; the second one is $\Gamma \Vdash P \triangleright \Delta$, where P is either a user-defined or a run-time process. The *context* Γ is a finite set of typings $x : G$ of expression variables; the *process-typing* (henceforth just *typing*) Δ is a finite set of typings $k^p : S$, where k^p is a polarised channel name, and S a session type. By $dom(\Gamma)$ and $dom(\Delta)$ we mean the set of variables or channel names that are typed, respectively, in Γ and Δ. Variables and (polarized) channel names are pairwise distinct both in Γ and Δ as usual; note that $k^+ \neq k^-$.

Definition 3.3 (Type System). *The rules of the type system are in Figure 1. In rule* [INACT-T] *a typing Δ is* completed *if for any $k^p \in dom(\Delta)$ we have $k^p : \text{end} \in \Delta$. In rule* [CONC-T] *the typing $\Delta \cdot \Delta'$ is the union of the typings Δ and Δ' provided that $dom(\Delta) \cap dom(\Delta') = \emptyset$, it is undefined otherwise; in the latter case the rule does not apply.*

The rules of Figure 1 are similar to those of [19], but for the following. In rules [ACC-T], [REQ-T] port names are not considered; consequently contexts Γ just contains expression variables and we do not have restrictions over port names. In [19] rule [CRES-T] is

$$\begin{array}{c} [\text{CRES-T}] \\ \dfrac{\Gamma \Vdash P \triangleright \Delta \cdot k^- : S \cdot k^+ : \overline{S}}{\Gamma \Vdash (\nu k)P \triangleright \Delta} \end{array}$$

where $\overline{S}$ is the dual of S. In the premise we have the typing $\Delta \cdot k^- : S \cdot k^+ : S'$ instead, with the side condition $\mathsf{f} : S \dashv S'$ yielding the orchestrator appearing in the process in the conclusion.

It is not difficult to check the parallel composition of the processes of running example is a typable user-defined process, in particular $\emptyset \Vdash Provider \mid Client \mid Bank \triangleright \emptyset$.

4 Operational semantics and error freeness

Because of the presence of orchestrators in session interactions, the notion of structural congruence has to be handled with some extra care than in usual calculi with session-types.

$$
\begin{array}{cc}
\text{[INACT-T]} & \text{[CONC-T]} \\[2pt]
\dfrac{\Delta \text{ completed}}{\Gamma \Vdash \mathbf{0} \triangleright \Delta} & \dfrac{\Gamma \Vdash P \triangleright \Delta \quad \Gamma \Vdash Q \triangleright \Delta'}{\Gamma \Vdash P \mid Q \triangleright \Delta \cdot \Delta'}
\end{array}
$$

$$
\begin{array}{cc}
\text{[ACC-T]} & \text{[REQ-T]} \\[2pt]
\dfrac{\Gamma \Vdash P\{k^{+}/k\} \triangleright \Delta \cdot k^{+} : S}{\Gamma \Vdash \mathtt{accept}_S(k)P \triangleright \Delta} & \dfrac{\Gamma \Vdash P\{k^{-}/k\} \triangleright \Delta \cdot k^{-} : S}{\Gamma \Vdash \mathtt{request}_S(k)P \triangleright \Delta}
\end{array}
$$

$$
\begin{array}{cc}
\text{[REC-T]} & \text{[SEND-T]} \\[2pt]
\dfrac{\Gamma, x : G \Vdash P \triangleright \Delta \cdot k^{p} : S}{\Gamma \Vdash k^{p}?(x).P \triangleright \Delta \cdot k^{p} : \textbf{?}[G].S} & \dfrac{\Gamma \vdash e : G \quad \Gamma \Vdash P \triangleright \Delta \cdot k^{p} : S}{\Gamma \Vdash k^{p}![e].P \triangleright \Delta \cdot k^{p} : ![G].S}
\end{array}
$$

$$
\begin{array}{cc}
\text{[CAT-T]} & \text{[THR-T]} \\[2pt]
\dfrac{\Gamma \Vdash P\{k'^{q}/k'\} \triangleright \Delta \cdot k^{p} : S_2 \cdot k'^{q} : S_1}{\Gamma \Vdash \mathtt{catch}\, k^{p}(k').P \triangleright \Delta \cdot k^{p} : \textbf{?}[S_1^{q}].S_2} & \dfrac{\Gamma \Vdash P \triangleright \Delta \cdot k^{p} : S_2}{\Gamma \Vdash \mathtt{throw}\, k^{p}[k'^{q}].P \triangleright \Delta \cdot k^{p} : ![S_1^{q}].S_2 \cdot k'^{q} : S_1}
\end{array}
$$

$$
\begin{array}{cc}
\text{[BR-T]} & \text{[SEL-T]} \\[2pt]
\dfrac{\forall i \in I \supseteq J \ \ \Gamma \Vdash P_i \triangleright \Delta \cdot k^{p} : S_i}{\Gamma \Vdash k^{p} \triangleright \{l_i : P_i\}_{i \in I} \triangleright \Delta \cdot k^{p} : \&\{l_j : S_j\}_{j \in J}} & \dfrac{\Gamma \Vdash P \triangleright \Delta \cdot k^{p} : S_j \quad j \in I}{\Gamma \Vdash k^{p} \triangleleft l_j.P \triangleright \Delta \cdot k^{p} : \bigoplus\{l_i : S_i\}_{i \in I}}
\end{array}
$$

$$
\begin{array}{c}
\text{[SSEL-T]} \\[2pt]
\dfrac{\forall i \in I \ \ \Gamma \Vdash P_i \triangleright \Delta \cdot k^{p} : S_i}{\Gamma \Vdash k^{p} \triangleleft [l_i : P_i]_{i \in I} \triangleright \Delta \cdot k^{p} : \boxplus\{l_i : S_i\}_{i \in I}}
\end{array}
$$

$$
\begin{array}{cc}
\text{[CRES-T]} & \text{[CRES'-T]} \\[2pt]
\dfrac{\Gamma \Vdash P \triangleright \Delta \cdot k^{-} : S_1 \cdot k^{+} : S_2 \quad \mathsf{f} : S_1 \dashv S_2}{\Gamma \Vdash (\nu k)(\langle k \rangle \mathsf{f} \mid P) \triangleright \Delta} & \dfrac{\Gamma \Vdash P \triangleright \Delta \quad k^{+}, k^{-} \notin dom(\Delta)}{\Gamma \Vdash (\nu k)P \triangleright \Delta}
\end{array}
$$

Figure 1: The type system.

Definition 4.1 (Structural Congruence). *The* structural congruence $\equiv$ *is the least congruence over run-time processes such that:*

1. $R \equiv R'$ *if R' is obtained from R by alphabetical change of bound channel names, avoiding name clashes,*

2. $(\nu k)(\langle k \rangle \mathsf{f} \mid Q \mid (\nu k')(\langle k' \rangle \mathsf{g} \mid Q' \mid R)) \equiv (\nu k')(\langle k' \rangle \mathsf{g} \mid (\nu k)(\langle k \rangle \mathsf{f} \mid Q \mid Q') \mid R)$ *if $k \notin \mathrm{FC}(R), k' \notin \mathrm{FC}(Q)$,*

3. *if X, Y, Z are either user-defined or run-time processes that are not a named orchestrator, then:* $(X \mid Y) \equiv (Y \mid X)$ *and* $(X \mid (Y \mid Z)) \equiv ((X \mid Y) \mid Z)$.

Definition 4.2 (Operational semantics). *The operational semantics of processes is described by the reduction rules listed in Figure 2.*

This operational semantics extends that in [13] by taking into account the new operator and the necessity for interactions of being orchestrated.

[LINK] If the session type S is compliant with S' by means of the orchestrator f, the session-opening request $\mathtt{request}_S(k)P$ can be accepted by $\mathtt{accept}_{S'}(k)Q$. A new channel is created for the opened session. The '$-$' end is owned by the process who requested the opening (the client) whereas the '$+$' end is owned by the other one (the server). To connect the orchestrator f to the opened session, f is labelled with the channel name k. This forces its orchestration actions to act only on synchronizations over the channel k.

[LINK]
$$\text{request}_S(k)P \mid \text{accept}_{S'}(k)Q \longrightarrow (\nu k)(\ \langle k \rangle f \mid P\{k^-/k\} \mid Q\{k^+/k\}\ ) \qquad \text{if } f : S \dashv S'$$

[ORCHCOMM]
$$(\nu k)(\ \langle k \rangle \bullet.f \mid k^p![e].P \mid k^{\overline{p}}?(x).Q\ ) \longrightarrow (\nu k)(\ \langle k \rangle f \mid P \mid Q\{v/x\}\ ) \qquad \text{if } e \downarrow v$$

[ORCHDELEG]
$$(\nu k)(\ \langle k \rangle \bullet.f \mid \text{throw}\, k^p[k'^q].P \mid \text{catch}\, k^{\overline{p}}(k').Q\ ) \longrightarrow (\nu k)(\ \langle k \rangle f \mid P \mid Q\{k'^q/k'\}\ )$$

[ORCHSEL]
$$(\nu k)(\ \langle k \rangle \textstyle\sum_{h \in H} l_h.f_h \mid k^p \triangleleft l_c.P \mid k^{\overline{p}} \triangleright \{l_i:Q_i\}_{i \in I}\ ) \longrightarrow (\nu k)(\ \langle k \rangle f_c \mid P \mid Q_c\ ) \qquad \text{if } c \in H \cap I$$

[ORCHSSEL]
$$(\nu k)(\ \langle k \rangle \textstyle\bigoplus_{h \in H} l_h.f_h \mid k^p \triangleleft [l_j : P_j]_{j \in J} \mid k^{\overline{p}} \triangleright \{l_i:Q_i\}_{i \in I}\ ) \longrightarrow (\nu k)(\ \langle k \rangle f_c \mid P_c \mid Q_c\ ) \quad \text{if } c \in H \cap I \cap J$$

[PAR]
$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$

[SCOP]
$$\frac{P \longrightarrow P'}{(\nu k)P \longrightarrow (\nu k)P'}$$

[STR]
$$\frac{Q \equiv P \longrightarrow P' \equiv Q'}{Q \longrightarrow Q'}$$

Figure 2: Operational Semantics

[ORCHCOMM], [ORCHDELEG], [ORCHSEL] In these rules the orchestrator enables the communication of a value, the communication of a channel, and the selection of a label, respectively. In rule [ORCHCOMM] the side condition $e \downarrow v$ reads: expression e evaluates to the value v.

[ORCHSSEL] In presence of a speculative selection, i.e. a number of choices possibly leading to synchronization failures, the role of the orchestrator is to suggest one among the safe choices. Notice that, in case the cardinality of $H \cap I \cap J$ is strictly greater than one, this rule is nondeterministic. In actual implementation it is reasonable to expect the orchestrator not to add nondeterminism to the system. We show in Section 6 how this can be obtained by interpreting $[l_j : P_j]_{j \in J}$ in $k^p \triangleleft [l_j : P_j]_{j \in J}$ as a priority list and how is it possible to synthesize, out of S and S' in [LINK], the orchestrator that suggests the safe choice possessing the highest priority, if any.

[PAR], [SCOP], [STR] These rules are standard.

Example 4.3. We can now see in Figure 3 the evolution of the user-defined process of our example

$$\textit{Client} \mid \textit{Provider} \mid \textit{Bank}$$

We assume that the client decides to rent an available movie.

The type system guarantees type-checked processes to be free from (a version of) the standard synchronization errors of session-type-based calculi, which now involve also orchestrators and that we dub *orchestrated synchronization errors*. It also prevent (a class of) errors due to the absence of orchestration action and that we dub *vacuous orchestration errors*.

Example 4.4. Let us see some examples of errors that cannot actually occur in typeable processes (see Lemma 4.6 below).

$(\nu k)(k^+![e].R \mid k^-![e'].R' \mid \langle k \rangle f)$ This process is stuck. It cannot be typed since the types for k^+ and k^- should have, respectively, the form $![G].S_1$ and $![G'].S_2$. Rule [CRES-T] cannot be applied to type the whole process, since, by Definition 2.3, these types are not compliant (even in case $G = G'$).

$(\nu k)(k^+ \triangleleft l.R \mid k^+ \triangleright \{l_j:R_j\}_{j \in J} \mid \langle k \rangle \sum_{i \in I} l_i.f_i)$ This stuck process cannot be typed since rule [CRES-T] requires the compliant types to be assigned to polarized channels with different polarities.
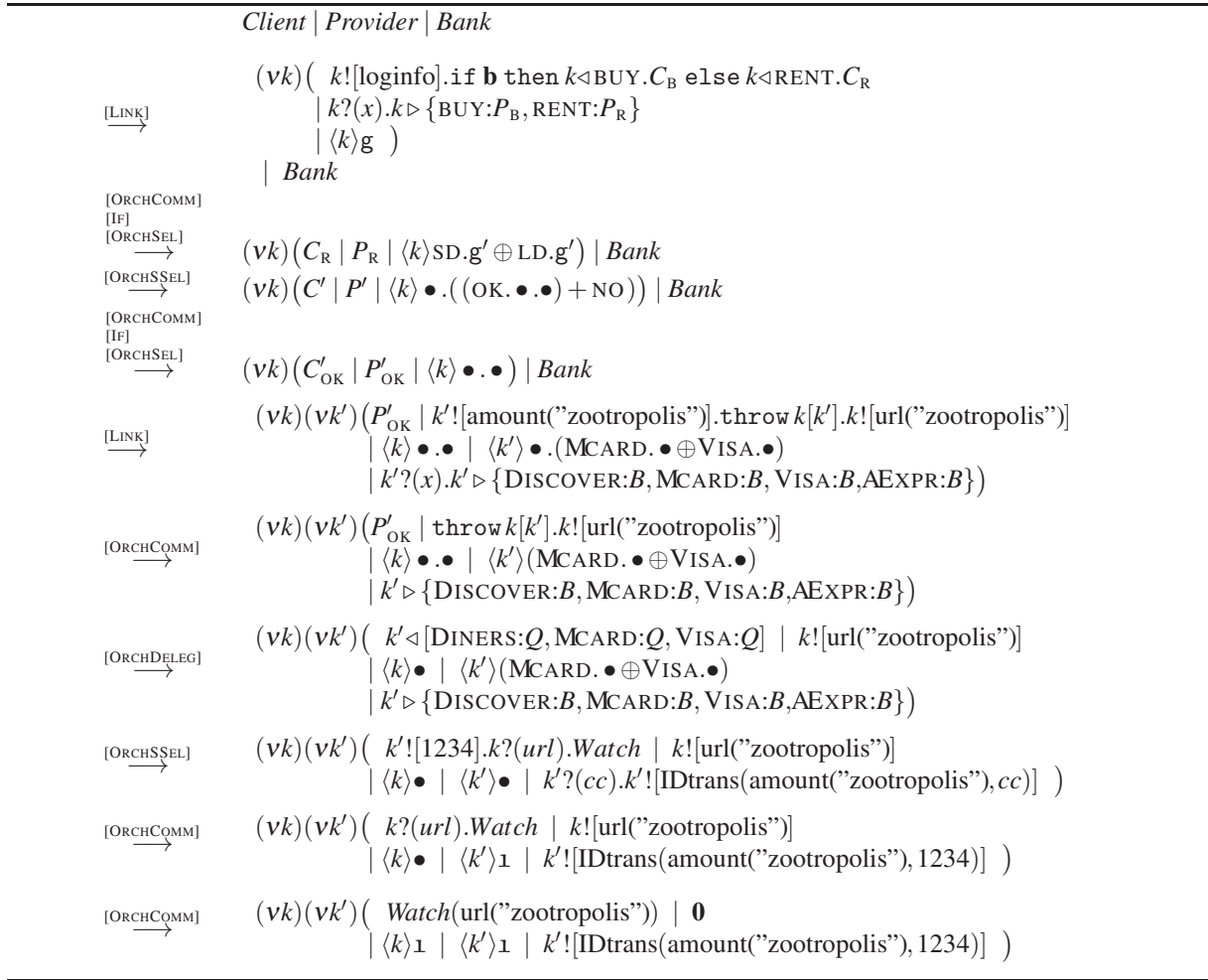
$$Client \mid Provider \mid Bank$$

$$(\nu k)\big(\; k![loginfo].\texttt{if } \mathbf{b} \texttt{ then } k \triangleleft BUY.C_B \texttt{ else } k \triangleleft RENT.C_R$$

[LINK] $\xrightarrow{\hspace{1cm}}$
$$\mid k?(x).k \triangleright \{BUY{:}P_B, RENT{:}P_R\}$$
$$\mid \langle k \rangle g \;\big)$$
$$\mid Bank$$

[ORCHCOMM]
[IF]
[ORCHSEL] $\xrightarrow{\hspace{1cm}}$
$$(\nu k)\big(C_R \mid P_R \mid \langle k \rangle SD.g' \oplus LD.g'\big) \mid Bank$$

[ORCHSSEL] $\xrightarrow{\hspace{1cm}}$
$$(\nu k)\big(C' \mid P' \mid \langle k \rangle \bullet.((OK.\bullet.\bullet) + NO)\big) \mid Bank$$

[ORCHCOMM]
[IF]
[ORCHSEL] $\xrightarrow{\hspace{1cm}}$
$$(\nu k)\big(C'_{OK} \mid P'_{OK} \mid \langle k \rangle \bullet.\bullet\big) \mid Bank$$

[LINK] $\xrightarrow{\hspace{1cm}}$
$$(\nu k)(\nu k')\big(P'_{OK} \mid k'![amount("zootropolis")].\texttt{throw } k[k'].k![url("zootropolis")]$$
$$\mid \langle k \rangle \bullet.\bullet \mid \langle k' \rangle \bullet.(MCARD.\bullet \oplus VISA.\bullet)$$
$$\mid k'?(x).k' \triangleright \{DISCOVER{:}B, MCARD{:}B, VISA{:}B, AEXPR{:}B\}\big)$$

[ORCHCOMM] $\xrightarrow{\hspace{1cm}}$
$$(\nu k)(\nu k')\big(P'_{OK} \mid \texttt{throw } k[k'].k![url("zootropolis")]$$
$$\mid \langle k \rangle \bullet.\bullet \mid \langle k' \rangle (MCARD.\bullet \oplus VISA.\bullet)$$
$$\mid k' \triangleright \{DISCOVER{:}B, MCARD{:}B, VISA{:}B, AEXPR{:}B\}\big)$$

[ORCHDELEG] $\xrightarrow{\hspace{1cm}}$
$$(\nu k)(\nu k')\big(\; k' \triangleleft [DINERS{:}Q, MCARD{:}Q, VISA{:}Q] \mid k![url("zootropolis")]$$
$$\mid \langle k \rangle \bullet \mid \langle k' \rangle (MCARD.\bullet \oplus VISA.\bullet)$$
$$\mid k' \triangleright \{DISCOVER{:}B, MCARD{:}B, VISA{:}B, AEXPR{:}B\}\big)$$

[ORCHSSEL] $\xrightarrow{\hspace{1cm}}$
$$(\nu k)(\nu k')\big(\; k'![1234].k?(url).Watch \mid k![url("zootropolis")]$$
$$\mid \langle k \rangle \bullet \mid \langle k' \rangle \bullet \mid k'?(cc).k'![IDtrans(amount("zootropolis"), cc)] \;\big)$$

[ORCHCOMM] $\xrightarrow{\hspace{1cm}}$
$$(\nu k)(\nu k')\big(\; k?(url).Watch \mid k![url("zootropolis")]$$
$$\mid \langle k \rangle \bullet \mid \langle k' \rangle 1 \mid k'![IDtrans(amount("zootropolis"), 1234)] \;\big)$$

[ORCHCOMM] $\xrightarrow{\hspace{1cm}}$
$$(\nu k)(\nu k')\big(\; Watch(url("zootropolis")) \mid \mathbf{0}$$
$$\mid \langle k \rangle 1 \mid \langle k' \rangle 1 \mid k'![IDtrans(amount("zootropolis"), 1234)] \;\big)$$

Figure 3: Reductions example

$(\nu k)(k^+![e].R \mid k^-?(x).R' \mid \langle k \rangle \sum_{i \in I} l_i.f_i)$ This process is stuck since the orchestrator does not enable the input/output syncronization. It cannot be typed either. In fact the types for k^+ and k^- should have, respectively, the form $![G].S_1$ and $?[G].S_2$, and rule [CRES-T] cannot be applied since, by Definition 2.3, it is impossible to have $\sum_{i \in I} l_i.f_i :![G].S_1 \dashv ?[G].S_2$.

$(\nu k)(k^-![e].R \mid R' \mid \langle k \rangle 1)$ Since any reduction is necessarily driven by an orchestrating action, there is no possibility for the process $k^-![e].R$ to progress. Unlike the previous examples, this sort of deadlock depends exclusively on the lack of orchestration actions.

Definition 4.5 (Errors).

i) A k^p-process is a run-time process term whose first action involves the channel k, namely a process having one of the following forms:

$k^p![e].P, \quad k^p?(x).P, \quad \texttt{throw } k^p[k'^q].P, \quad \texttt{catch } k^p(k'^q).P, \quad k^p \triangleleft l.P, \quad k^p \triangleright \{l_i{:}P_i\}_{i \in I}, \quad k^p \triangleleft [l_i{:}P_i]_{i \in I};$

ii) A potential k-redex is a process term formed by the parallel composition of one k^p-processe, one k^q-processe, for some p and q, and one k-named orchestrator;

iii) A process P is an orchestrated synchronization error (orch-synch error, for short) if it contains a potential k-redex which does not reduces;

iv) A process R is a vacuous-orchestration error *if it contains a subterm R' such that*
$$R' = \langle k \rangle 1 \mid R'' \quad \text{where } R'' \text{ is a } k^- \text{-process;}$$

v) An error *is either an orchestrated synchronization error or a vacuous-orchestration error.*

The type system guarantees that a typable process cannot be an error.

Lemma 4.6. *Let $\Gamma \Vdash R \triangleright \Delta$. Then R is not an error.*

By means of the Subject Reduction property we show that a typable initial process never reduces to an error.

Concerning the typing of expressions, we assume the standard property that if $\Gamma \vdash e : G$ and $e \downarrow v$ then $\Gamma \vdash v : G$. Then the following technical lemma, to be used for the subject reduction property (Theorem 4.9), is proved by an easy induction on derivations:

Lemma 4.7. *If $\Gamma \vdash e : G$, $\Gamma, x : G \Vdash R \triangleright \Delta$, and $e \downarrow v$ then $\Gamma \Vdash R\{v/x\} \triangleright \Delta$.*

Before proceeding with the Subject Reduction property, we show that typability is invariant with respect to the structural congruence formalized in Definition 4.1. The presence of orchestrators makes this usually fairly standard proof more subtle to handle.

Lemma 4.8. *If $\Gamma \Vdash P \triangleright \Delta$ and $P \equiv Q$ then $\Gamma \Vdash Q \triangleright \Delta$.*

Proof. By induction over the definition of $\equiv$. We illustrate the case 4.1.2, where we have:
$$(\nu k)(\langle k \rangle f \mid P \mid (\nu k')(\langle k' \rangle g \mid Q \mid R)) \equiv (\nu k')(\langle k' \rangle g \mid (\nu k)(\langle k \rangle f \mid P \mid Q) \mid R)$$
and $k \notin \text{FC}(R), k' \notin \text{FC}(P)$. The derivation of $\Gamma \Vdash (\nu k)(\langle k \rangle f \mid P \mid (\nu k')(\langle k' \rangle g \mid Q \mid R)) \triangleright \Delta$ ends by:

$$
\cfrac{
  \cfrac{
    \Gamma \vdash P \triangleright \Delta_2 \qquad
    \cfrac{
      \cfrac{
        \cfrac{\Gamma \vdash Q \triangleright \Delta_5 \quad \Gamma \vdash R \triangleright \Delta_6}{\Gamma \vdash Q \mid R \triangleright \Delta_4} \qquad g : S_3 \dashv S_4
      }{\Gamma \vdash (\nu k')(\langle k' \rangle g \mid Q \mid R) \triangleright \Delta_3}
    }{\Gamma \vdash P \mid (\nu k')(\langle k' \rangle g \mid Q \mid R) \triangleright \Delta_1} \qquad f : S_1 \dashv S_2
  }{}
}{\Gamma \vdash (\nu k)(\langle k \rangle f \mid P \mid (\nu k')(\langle k' \rangle g \mid Q \mid R)) \triangleright \Delta}
$$

where $\Delta_1 = \Delta \cdot k^- : S_1 \cdot k^+ : S_2$ for some S_1, S_2; $\Delta_1 = \Delta_2 \cdot \Delta_3$; $\Delta_4 = \Delta_3 \cdot k'^- : S_3 \cdot k'^+ : S_4$ for some S_3, S_4; finally $\Delta_4 = \Delta_5 \cdot \Delta_6$. Let's set $\Delta \setminus k = \Delta \setminus \bigcup_{S,S'} \{k^p : S, k^{\overline{p}} : S'\}$. Then such a derivation can be rearranged as follows:

$$
\cfrac{
  \cfrac{
    \cfrac{
      \cfrac{\Gamma \vdash P \triangleright \Delta_2 \quad \Gamma \vdash Q \triangleright \Delta_5}{\Gamma \vdash P \mid Q \triangleright \Delta_2 \cdot \Delta_5} \qquad f : S_1 \dashv S_2
    }{\Gamma \vdash (\nu k)(\langle k \rangle f \mid P \mid Q) \triangleright (\Delta_2 \cdot \Delta_5) \setminus k} \qquad \Gamma \vdash R \triangleright \Delta_6
  }{\Gamma \vdash (\nu k)(\langle k \rangle f \mid P \mid Q) \mid R \triangleright ((\Delta_2 \cdot \Delta_5) \setminus k) \cdot \Delta_6} \qquad g : S_3 \dashv S_4
}{\Gamma \vdash (\nu k')(\langle k' \rangle g \mid (\nu k)(\langle k \rangle f \mid P \mid Q) \mid R) \triangleright (((\Delta_2 \cdot \Delta_5) \setminus k) \cdot \Delta_6) \setminus k'}
$$

To see that this is a correct derivation, let us assume that $k' \notin dom(\Delta_2)$, because $k' \notin \text{FC}(P)$ is the side condition of case 4.1.2. Now $\Delta_2 \cdot \Delta_5$ is defined since $\Delta_5 \setminus k' = \Delta_3$, and we know that $\Delta_2 \cdot \Delta_3 = \Delta_1$ is defined. On the other hand if $k'' \in dom(\Delta_6)$ then $k'' \notin dom(\Delta_5)$ because $\Delta_5 \cdot \Delta_6 = \Delta_4$ is defined; also if $k'' \neq k$ then $k'' \notin dom(\Delta_2)$ since $dom(\Delta_6) \subseteq dom(\Delta_3) \setminus \{k'\}$ and Δ_2 and Δ_3 are compatible. It follows that

$((\Delta_2 \cdot \Delta_5) \setminus k) \cdot \Delta_6$ is defined. Finally, assuming without loss of generality that $k \notin dom(\Delta_6)$ as we know that $k \notin \text{FC}(R)$, we conclude:

$$
\begin{aligned}
\Delta &= (\Delta_2 \cdot (\Delta_5 \cdot \Delta_6) \setminus k') \setminus k && \text{by the first derivation} \\
&= (\Delta_2 \cdot \Delta_5 \cdot \Delta_6) \setminus k' \setminus k && \text{since } k' \notin dom(\Delta_2) \\
&= ((\Delta_2 \cdot \Delta_5) \setminus k \cdot \Delta_6 \setminus k) \setminus k' \\
&= (((\Delta_2 \cdot \Delta_5) \setminus k) \cdot \Delta_6) \setminus k' && \text{since } k \notin dom(\Delta_6)
\end{aligned}
$$

$\square$

Since no typable processes is an orch-synch error (Lemma 4.6), the following theorem property guarantees (see Corollary 4.10) that no error can appear during the evolution of a typable user-defined process.

Theorem 4.9 (Subject reduction). *If $\Gamma \Vdash P \rhd \Delta$ and $P \longrightarrow Q$ then $\Gamma \Vdash Q \rhd \Delta$.*

Proof. By induction over the definition of $P \longrightarrow Q$. In case of rule [LINK] we have:

$$\texttt{request}_S(k)P \mid \texttt{accept}_{S'}(k)Q \longrightarrow (\nu k)(\langle k \rangle \mathsf{f} \mid P\{k^-/k\} \mid Q\{k^+/k\})$$

for some f such that $\mathsf{f} : S \dashv S'$. On the other hand by hypothesis and the shape of the rules, we have the derivation:

$$
\frac{
\dfrac{\Gamma \Vdash P\{k^-/k\} \rhd \Delta \cdot k^- : S}{\Gamma \Vdash \texttt{request}_S(k)P \rhd \Delta} \qquad
\dfrac{\Gamma \Vdash Q\{k^+/k\} \rhd \Delta \cdot k^+ : S'}{\Gamma \Vdash \texttt{accept}_{S'}(k)Q \rhd \Delta'}
}{
\Gamma \Vdash \texttt{request}_S(k)P \mid \texttt{accept}_{S'}(k)Q \rhd \Delta \cdot \Delta'
}
$$

where $\Delta \cdot \Delta'$ has to be defined (namely $dom(\Delta) \cap dom(\Delta') = \emptyset$). This implies that:

$$(\Delta \cdot k^- : S) \cdot (\Delta' \cdot k^+ : S') = \Delta \cdot \Delta' \cdot k^- : S \cdot k^+ : S'$$

is defined as well, so that using the fact that $\mathsf{f} : S \dashv S'$, we obtain the following derivation:

$$
\frac{
\dfrac{\Gamma \Vdash P\{k^-/k\} \rhd \Delta \cdot k^- : S \quad \Gamma \Vdash Q\{k^+/k\} \rhd \Delta \cdot k^+ : S'}{\Gamma \Vdash P\{k^-/k\} \mid Q\{k^+/k\} \rhd \Delta \cdot \Delta' \cdot k^- : S \cdot k^+ : S'}
}{
\Gamma \Vdash (\nu k)(\langle k \rangle \mathsf{f} \mid P\{k^-/k\} \mid Q\{k^+/k\}) \rhd \Delta \cdot \Delta'
}
$$

In case of rule [ORCHCOMM] we have:

$$(\nu k)(\langle k \rangle \bullet .\mathsf{f} \mid k^p![e].P \mid k^{\overline{p}}?(x).Q) \longrightarrow (\nu k)(\langle k \rangle \mathsf{f} \mid P \mid Q\{v/x\}).$$

Let us suppose without loss of generality that $p = -$ so that $\overline{p} = +$. By hypothesis we have a derivation ending by:

$$
\frac{
\dfrac{\Gamma \Vdash k^-![e].P \rhd \Delta'' \cdot k^- : S \quad \Gamma \Vdash k^+?(x).Q \rhd \Delta''' \cdot k^+ : S'}{\Gamma \Vdash k^-![e].P \mid k^+?(x).Q \rhd \Delta \cdot k^- : S \cdot k^+ : S'}
}{
\Gamma \Vdash (\nu k)(\langle k \rangle \bullet .\mathsf{f} \mid k^p![e].P \mid k^{\overline{p}}?(x).Q) \rhd \Delta
}
$$

for some S, S' such that $\bullet.\mathsf{f} : S \dashv S'$, and Δ'', Δ''' such that $\Delta = \Delta'' \cdot \Delta'''$. From the derivability of $\Gamma \Vdash \vdash k^-![e].P \rhd \Delta'' \cdot k^- : S$ we deduce that $\Gamma \vdash e : G$ for some ground G, $\Gamma \Vdash P \rhd \Delta'' \cdot k^- : S''$ and that $S = ![G].S''$ for some S''. Similarly we know that $\Gamma \vdash x : G'$ for some G', $\Gamma, x : G' \Vdash Q \rhd \Delta''' \cdot k^+ : S'''$ and that $S' = ?[G'].S'''$.

Now from this and the fact that $\bullet.\mathsf{f} : ![G].S'' \dashv ?[G'].S'''$ we infer by Definition 2.3 that G and G' must be the same and that $\mathsf{f} : S'' \dashv S'''$. Hence we have the derivation:

$$\frac{\dfrac{\Gamma \Vdash P \rhd \Delta'' \cdot k^- : S'' \quad \Gamma \Vdash Q\{v/x\} \rhd \Delta''' \cdot k^+ : S'''}{\Gamma \Vdash P \mid Q\{v/x\} \rhd \Delta \cdot k^- : S'' \cdot k^+ : S'''}}{\Gamma \Vdash (\nu k)(\langle k \rangle \mathsf{f} \mid P \mid Q\{v/x\}) \rhd \Delta}$$

where $\Gamma \Vdash Q\{v/x\} \rhd \Delta''' \cdot k^+ : S'''$ follows by $\Gamma \vdash e : G$, derivability of $\Gamma, x : G \Vdash Q \rhd \Delta''' \cdot k^+ : S'''$, that $e \downarrow v$ and Lemma 4.7.

In case of [ORCHDELEG] we have:

$$(\nu k)(\langle k \rangle \bullet.\mathsf{f} \mid \mathtt{throw}\, k^p[k'^q].P \mid \mathtt{catch}\, k^{\overline{p}}(k').Q) \longrightarrow (\nu k)(\langle k \rangle \mathsf{f} \mid P \mid Q\{k'^q/k'\})$$

Then by hypothesis there exists the derivation:

$$\frac{\dfrac{\Gamma \Vdash P \rhd \Delta' \cdot k^p : S_1}{\Gamma \Vdash \mathtt{throw}\, k^p[k'^q].P \rhd \Delta' \cdot k'^q : S' \cdot k^p :![S'^q]S_1} \quad \dfrac{\Gamma \Vdash Q\{k'^r/k'\} \rhd \Delta'' \cdot k'^r : S'' \cdot k^{\overline{p}} : S_2}{\Gamma \Vdash \mathtt{catch}\, k^{\overline{p}}(k').Q \rhd \Delta'' \cdot k^{\overline{p}} :?[S''^r]S_2}}{\dfrac{\Gamma \Vdash \mathtt{throw}\, k^p[k'^q].P \mid \mathtt{catch}\, k^{\overline{p}}(k').Q \rhd \Delta \cdot k'^q : S' \cdot k^p :![S'^q]S_1 \cdot k^{\overline{p}} :?[S''^r]S_2}{\Gamma \Vdash (\nu k)(\langle k \rangle \bullet.\mathsf{f} \mid \mathtt{throw}\, k^p[k'^q].P \mid \mathtt{catch}\, k^{\overline{p}}(k').Q) \rhd \Delta \cdot k'^q : S'}}$$

where $\Delta' \cdot \Delta'' = \Delta$; further, assuming w.l.o.g. that $p = +$ and hence that $\overline{p} = -$, we have $\bullet.\mathsf{f} :![S'^q]S_1 \dashv ?[S''^r]S_2$, which implies that $S'' = S'$, $r = q$, and $\mathsf{f} : S_1 \dashv S_2$. Therefore there exists the derivation:

$$\frac{\dfrac{\Gamma \Vdash P \rhd \Delta' \cdot k^p : S_1 \quad \Gamma \Vdash Q\{k'^q/k'\} \rhd \Delta'' \cdot k'^q : S' \cdot k^{\overline{p}} : S_2}{\Gamma \Vdash P \mid Q\{k'^q/k'\} \rhd \Delta \cdot k'^q : S' \cdot k^p : S_1 \cdot k^{\overline{p}} : S_2}}{\Gamma \Vdash (\nu k)(\langle k \rangle \mathsf{f} \mid P \mid Q\{k'^q/k'\}) \rhd \Delta \cdot k'^q : S'}$$

where the fact that $\Delta \cdot k'^q : S' \cdot k^p : S_1 \cdot k^{\overline{p}} : S_2$ is well defined follows by the fact that $\Delta \cdot k'^q : S' \cdot k^p : ![S'^q]S_1 \cdot k^{\overline{p}} :?[S''^r]S_2$ is such.

Let us consider the case [ORCHSSEL]:

$$(\nu k)(\langle k \rangle \oplus_{h \in H} l_h.\mathsf{f}_h \mid k^p \lhd [l_j : P_j]_{j \in J} \mid k^{\overline{p}} \rhd \{l_i : Q_i\}_{i \in I}) \longrightarrow (\nu k)(\langle k \rangle \mathsf{f}_c \mid P_c \mid Q_c)$$

where $c \in H \cup I \cup J$. By hypothesis we have a derivation ending by:

$$\frac{\Gamma \Vdash k^p \lhd [l_j : P_j]_{j \in J} \mid k^{\overline{p}} \rhd \{l_i : Q_i\}_{i \in I} \rhd \Delta \cdot k^p : \boxplus\{l_j : S_j\}_{j \in J} \cdot k^{\overline{p}} : \&\{l_i : S'_i\}_{i \in I}}{\Gamma \Vdash (\nu k)(\langle k \rangle \oplus_{h \in H} l_h.\mathsf{f}_h \mid k^p \lhd [l_j : P_j]_{j \in J} \mid k^{\overline{p}} \rhd \{l_i : Q_i\}_{i \in I}) \rhd \Delta}$$

with the side condition $\oplus_{h \in H} l_h.\mathsf{f}_h : \boxplus\{l_j : S_j\}_{j \in J} \dashv \&\{l_i : S'_i\}_{i \in I}$. By Definition 2.3 this implies that $H \subseteq J \cap I$ is a non empty set such that for all $h \in H$ it holds $\mathsf{f}_h : S_h \dashv S'_h$. The premise of the last inference in the above derivation must be derived by rule [CONC-T] from

$$\frac{\forall j \in J \quad \Gamma \Vdash P_j \rhd \Delta' \cdot k^p : S_j}{\Gamma \Vdash k^p \lhd [l_j : P_j]_{j \in J} \rhd \Delta' \cdot k^p : \boxplus\{l_j : S_j\}_{j \in J}} \quad \text{and} \quad \frac{\forall i \in I \quad \Gamma \Vdash Q_i \rhd \Delta'' \cdot k^{\overline{p}} : S'_i}{\Gamma \Vdash k^{\overline{p}} \rhd \{l_i : Q_i\}_{i \in I} \rhd \Delta'' \cdot k^{\overline{p}} : \&\{l_i : S'_i\}_{i \in I}}$$

where $\Delta = \Delta' \cdot \Delta''$ is defined. Now since $c \in H \subseteq J \cap I$, we know that $\mathsf{f}_c : S_c \dashv S'_c$ so that from the above we obtain the derivation:

$$\frac{\Gamma \Vdash P_c \rhd \Delta' \cdot k^p : S_c \quad \Gamma \Vdash Q_c \rhd \Delta'' \cdot k^{\overline{p}} : S'_c}{\dfrac{\Gamma \Vdash P_c \mid Q_c \rhd \Delta \cdot k^p : S_c \cdot k^{\overline{p}} : S'_c}{\Gamma \Vdash (\nu k)(\langle k \rangle \mathsf{f}_c \mid P_c \mid Q_c) \rhd \Delta}}$$

 The case of rule [ORCHSEL] is similar and simpler. Finally rules [PAR], [SCOP] and [STR] follow by induction and Lemma 4.8. □

Corollary 4.10 (Error freeness). *If P is a user-defined process such that $\Gamma \Vdash P \triangleright \Delta$ for some Γ and Δ, and R is a run-time process such that $P \xrightarrow{*} R$, then R is not an error.*

Notice that, whereas our type system prevents deadlocks like the last one in Example 4.4, it cannot prevent deadlocks due to the presence of subterms like $(\nu k)(k^+![e].R \mid R' \mid \langle k \rangle \mathbb{1})$, where R' is not a k^--process (see Example 5.1 and Definition 5.2 below). This sort of deadllocks are intrinsecally due to the asymmetric nature of our compliance relation. They can be avoided either by using a less general compliance relation (namely a symmetric restriction of the present one), or by extending the operational semantics, as we shall do in the next section.

5 ⊣-dependent deadlocks and clean-up reductions

Since in our setting session compatibility, unlike e.g duality, is asymmetric (compliance is a "client biased" relation), we have to face additional problems related to peculiar deadlock states that we dub ⊣-dependent deadlocks. These are due to actions on the server side that can remain unmatched after the client has reached a successful state, as shown in the following example.

Example 5.1. Let us consider the following processes:

$$P = \text{request}_{S_a}(k)k![4].\text{request}_{S_b}(k')k'![\text{True}] \qquad Q = \text{accept}_{S_c}(k)k?(x).k![x].\text{accept}_{S_d}(k')k'?(y)$$

where $\quad S_a = ![\text{Nat}] \quad S_b = ![\text{Bool}] \quad S_c = ?[\text{Nat}].![\text{Nat}] \quad S_d = ?[\text{Bool}]$

It is not difficult to see that the system $(P \mid Q)$ does type check and that its evolution proceeds as follows: The first session can be opened, since $\bullet.\mathbb{1} : S_a \dashv S_c$, and we get

$$(\nu k)(\ \langle k \rangle \bullet .\mathbb{1} \mid k^-![4].\text{request}_{S_b}(k')k'![\text{True}] \mid k^+?(x).k^+![x].\text{accept}_{S_d}(k')k'?(y)\ )$$

Now the orchestrator enables the input/output interaction on channel k, letting the system evolve to

$$(\nu k)(\ \langle k \rangle \mathbb{1} \mid \text{request}_{S_b}(k')k'![\text{True}] \mid k^+![4].\text{accept}_{S_d}(k')k'?(y)\ )$$

The behaviour described by S_a has now been completed, but the $k^+![4]$ operation on the server side now prevents the system to progress.

We call ⊣-dependent deadlock a stuck state like the one described in the above example.

Definition 5.2. *A process P is a ⊣-dependent deadlock if $P \not\rightarrow$ and it contains a subterm R of the form*
$$R = (\nu k)(\langle k \rangle \mathbb{1} \mid R' \mid R'') \quad \text{where } R' \text{ is a } k^+\text{-process.}$$

In order to avoid the above sort of deadlock states without modifying our notion of compliance, we extend the operational semantics with some extra reductions for orchestrators like $\langle k \rangle \mathbb{1}$, i.e. the orchestrator that succesfully completed all the actions requested by the client on channel k. In particular the extra reductions formalized in Figure 4 allow an orchestrator $\langle k \rangle \mathbb{1}$ to vacuously satisfy all the synchronization actions on channel k coming from the server, namely those having k^+ in their prefix. We call them **clean-up reductions**

By adding the clean-up reductions to the operational semantics it is not difficult to check that the subject reduction property (Theorem 4.9) still holds. Out of that we get the following extension of Corollary 4.10.

Corollary 5.3 (Error freeness). *If P is a user-defined process such that $\Gamma \Vdash P \triangleright \Delta$ for some Γ and Δ, and R is a run-time process such that $P \xrightarrow{*} R$ (where also clean-up reductions are considered) then R is neither an error nor a ⊣-dependent deadlock.*

[ORCHCLNUP1]
$(vk)(\langle k \rangle \mathbf{1} \mid \pi.R' \mid R) \longrightarrow (vk)(\langle k \rangle \mathbf{1} \mid R' \mid R)$ where $\pi \in \{k^+![e], k^+?(x), k^+ \lhd l\}$

[ORCHCLNUP2]
$(vk)(\langle k \rangle \mathbf{1} \mid k^+ \rhd \{l_i : R'_i\}_{i \in I} \mid R) \longrightarrow (vk)(\langle k \rangle \mathbf{1} \mid R'_c \mid R)$ if $c \in I$

[ORCHCLNUP3]
$(vk)(\langle k \rangle \mathbf{1} \mid k^+ {}_{\mathrm{prty}} \lhd [l_i : Q_i]_{i \in I} \mid R') \longrightarrow (vk)(\langle k \rangle \mathbf{1} \mid R'_c \mid R)$ if $c \in I$

Figure 4: Clean-up reductions

Let us see how the $\dashv$-dependent deadlock is avoided in Example 5.1.

Example 5.4. Once we get to

$$(vk)(\langle k \rangle \mathbf{1} \mid \mathtt{request}_{S_b}(k')k'![\mathrm{True}] \mid k^+![4].\mathtt{accept}_{S_d}(k')k'?(y))$$

the reduction ORCHCLNUP1 allows $\langle k \rangle \mathbf{1}$ to vacuously satisfy the action$k^+![4]$. So, by such a rule, the system can evolve to

$$(vk)(\langle k \rangle \mathbf{1} \mid \mathtt{request}_{S_b}(k')k'![\mathrm{True}] \mid \mathtt{accept}_{S_d}(k')k'?(y))$$

and then, since $\bullet.\mathbf{1} : S_b \dashv S_d$, it can progress to $(vk)(\langle k \rangle \mathbf{1} \mid (vk')(\langle k' \rangle \bullet.\mathbf{1} \mid k'![\mathrm{True}] \mid k'?(y)))$ and finally to $(vk)(\langle k \rangle \mathbf{1} \mid (vk')(\langle k' \rangle \mathbf{1} \mid \mathbf{0} \mid \mathbf{0}))$.

Remark 5.5. *As previously mentioned, the necessity of clean-up reductions is due to the asymmetric nature of the compliance relation. $\dashv$-dependent deadlocks would not appear in typed terms if we forced compliance to be symmetric, for instance by replacing item 1 of Definition 2.3 with $\mathbf{1} : \mathrm{end} \dashv \mathrm{end}$.*

6   Deterministic-orchestrators and priority choices

As recalled in the introduction and discussed in e.g. [18, 6], the orchestration process should not exhibit any internal nondeterminism. In the present section, we stand by such a viewpoint and consider only deterministic orchestrators.

Definition 6.1 (Deterministic orchestrators).
An orchestrator $f \in \mathrm{Orch}$ is deterministic *if it does not contain any occurrence of the $\oplus$ operator.*

Deterministic orchestrators can be used to select exactly one safe option in a construct like $k \lhd [l_i : P_i]_{i \in I}$ according to a given priority ordering among the failure-amenable options $\{l_i\}_{i \in I}$. The priority ordering can be explicitly specified in the speculative types, which we interpret now as *speculative types with priorities*.

Definition 6.2 (Types with priorities). *We modify the set S of session types in Def. 2.1 by replacing $\boxplus\{l_i : S_i\}_{i \in I}$ by $\boxplus\langle\langle l_1 : S_1, \ldots, l_n : S_n\rangle\rangle$, with $n \geq 1$.*
The option represented by the label l_i ($1 \leq i \leq n-1$) is assumed to have higher priority than the one represented by l_{i+1}.

The process calculus remains unchanged, as well as the operational semantics, but for rule [LINK], which we now replace by

[LINKPT]
$\mathtt{request}_S(k)P \mid \mathtt{accept}_{S'}(k)Q \longrightarrow (vk)(\langle k \rangle f \mid P\{k^-/k\} \mid Q\{k^+/k\})$ if $f = \mathbf{Synth}(S, S') \neq \mathbf{fail}$

where **Synth** is the orchestrator synthesis algorithm described in Figure 6 and for which the following holds.

Lemma 6.3. *i)* **Synth**$(S, S') \neq$ **fail** *iff* $S \dashv S'$;

ii) $\mathsf{f} =$ **Synth**$(S, S') \neq$ **fail** *implies* $\mathsf{f} : S \dashv S'$ *and* f *is deterministic.*

The proof of the above lemma is easy by definition of **Synth**, which in turn can be easily obtained from that of orchestrated compliance. Moreover, by construction, it is possible to show that the synthesized orchestrator always chooses the highest-priority option for *speculative types with priorities*.

Process-level-specified priorities As mentioned in the introduction, it is possible to specify the priorities among failure-amenable options at the process level instead of at the type level. In order to do that, there is no need to change the set of types as provided in Definition 2.1. We change instead the speculative selection operator.

Definition 6.4 (Processes with priorities). *We modify the set P of Processes in Def. 3.1 by replacing* $k \triangleleft [l_i{:}P_i]_{i \in I}$ *by* $k \triangleleft \langle\!\langle l_1{:}P_1, \ldots, l_n{:}P_n \rangle\!\rangle$, *with* $n \geq 1$.
The option represented by the label l_i $(1 \leq i \leq n-1)$ *is assumed to have higher priority than the one represented by* l_{i+1}.

The operational semantics remains unchanged, but for the rules [LINK] and [ORCHSSEL] which are now replaced by

> [LINKPP]
> $\texttt{request}_S(k)P \mid \texttt{accept}_{S'}(k)Q \longrightarrow (\nu k)(\langle k \rangle \mathsf{f} \mid P\{k^-/k\} \mid Q\{k^+/k\})$
> $$\text{if } \mathsf{f} = \textbf{Synth}^{\text{UD}}(S, S') \neq \textbf{fail}$$

> [ORCHSSELPP]
> $(\nu k)(\langle k \rangle \oplus_{i \in I} \mathsf{f}_i \mid k \triangleleft \langle\!\langle l_1{:}P_1, \ldots, l_n{:}P_n \rangle\!\rangle \mid k \triangleright \{l_j{:}Q_j\}_{j \in J}) \longrightarrow (\nu k)(\langle k \rangle \mathsf{f}_c \mid P_m \mid Q_c)$
> $$\text{if } c \in I \cap J, \, 1 \leq m \leq n \text{ and } l_c = l_m, \text{ where } m = \min\{h \mid l_m \in \{l_i\}_{i \in I}\}$$

where **Synth**$^{\text{UD}}$ returns the (possibly) non deterministic orchestrator exhibiting all safe options for speculative selection choices. It can be easily obtainined out of **Synth** by replacing the third **else** clause in Figure 6 by the clause

else **if** $(S = \&\{l_i{:}S_i\}_{i \in I}$ and $S' = \boxplus\langle\!\langle l_j{:}S'_j \rangle\!\rangle_{j \in J})$ **or** $(S = \boxplus\langle\!\langle l_j{:}S_j \rangle\!\rangle_{j \in J}$ and $S' = \&\{l_i{:}S'_i\}_{i \in I})$
    then let res $=$ **Synth**$^{\text{all}}$ $([S_h \dashv S'_h]_{h \in I \cap J})$
        in if res $\neq [\ ]$ **then** $l_{c_1}.\mathsf{f}_1 \oplus \ldots \oplus l_{c_n}.\mathsf{f}_n$ **where** res $= [(\mathsf{f}_1, c_1), \ldots (\mathsf{f}_n, c_n)]$
            else fail

where **Synth**$^{\text{all}}$ is defined by

Synth$^{\text{all}}([\,]) = [\ ]$
Synth$^{\text{all}}((S_c \dashv S'_c){:}\mathsf{xs}) =$
    let $\mathsf{f} =$ **Synth**$^{\text{UD}}(S_c \dashv S'_c)$ **in** (**if** $\mathsf{f} =$ **fail then Synth**$^{\text{all}}(\mathsf{xs})$ **else** $(\mathsf{f}, c){:}$**Synth**$^{\text{all}}(\mathsf{xs}))$

Remark 6.5. *The algorithms* **Synth** *and* **Synth**$^{\text{UD}}$ *can be adapted to the case of recursive orchestrators and types following the treatment of recursion in [6].*

7 Conclusions and future work

We have defined a type system with binary session types for a calculus with orchestrated interactions. The condition for session opening is here more permissive than in usual calculi with session types in that the types of the processes participating in a session have just to be *compliant* rather than dual to each other. The relation of compliance stems from the theory of contracts [15, 16, 11, 7]. In particular, our compliance relation has been inspired by the orchestrated compliances proposed in [18] and [6], where

---

$\textbf{Synth}(S \dashv S') \;=\; \textbf{if } S = \textsf{end then } \mathbb{1}$

$\qquad\qquad \textbf{else if } (S =\textbf{?}[G].S_1 \textbf{ and } S' =\textbf{!}[G].S_1') \textbf{ or } (S =\textbf{!}[G].S_1 \textbf{ and } S' =\textbf{?}[G].S_1')$

$\qquad\qquad\qquad \textbf{then let } \mathsf{f} = \textbf{Synth}\,(S_1 \dashv S_1')$

$\qquad\qquad\qquad\qquad \textbf{in if } \mathsf{f} \neq \textbf{fail then } \bullet.\mathsf{f} \textbf{ else fail}$

$\qquad\qquad \textbf{else if } (S =\textbf{?}[S_1^p].S_2 \textbf{ and } S' =\textbf{!}[S_1^p].S_2') \textbf{ or } (S =\textbf{!}[S_1^p].S_2 \textbf{ and } S' =\textbf{?}[S_1^p].S_2')$

$\qquad\qquad\qquad \textbf{then let } \mathsf{f} = \textbf{Synth}(S_2 \dashv S_2')$

$\qquad\qquad\qquad\qquad \textbf{in if } \mathsf{f} \neq \textbf{fail then } \bullet.\mathsf{f} \textbf{ else fail}$

$\qquad\qquad \textbf{else if } (S = \&\{l_i{:}S_i\}_{i\in I} \textbf{ and } S' = \boxplus\langle\!\langle l_j{:}S_j'\rangle\!\rangle_{j\in J}) \textbf{ or } (S = \boxplus\langle\!\langle l_j{:}S_j\rangle\!\rangle_{j\in J} \textbf{ and } S' = \&\{l_i{:}S_i'\}_{i\in I})$

$\qquad\qquad\qquad \textbf{then let } \mathrm{res} = \textbf{Synth}^{\textbf{fst}}\,([S_h \dashv S_h']_{h\in I\cap J})$

$\qquad\qquad\qquad\qquad \textbf{in if } (\mathrm{res} = \textbf{fail}) \textbf{ then fail else } l_c.\mathsf{f} \textbf{ where } (\mathsf{f},c) = \mathrm{res}$

$\qquad\qquad \textbf{else if } (S = \bigoplus\{l_i{:}S_i\}_{i\in I} \textbf{ and } S' = \&\{l_j{:}S_j'\}_{j\in J}) \textbf{ or } (S = \&\{l_j{:}S_j\}_{j\in J} \textbf{ and } S' = \bigoplus\{l_i{:}S_i'\}_{i\in I})$

$\qquad\qquad\qquad \textbf{then let } \forall i \in I.\mathsf{f}_i = \textbf{Synth}(S_i, S_i')$

$\qquad\qquad\qquad\qquad \textbf{in if } \forall i \in I.\mathsf{f}_i \neq \textbf{ fail then } \sum_{i\in I} l_i.\mathsf{f}_i \textbf{ else fail}$

$\qquad\qquad \textbf{else fail}$

$\qquad \textbf{where } \textbf{Synth}^{\textbf{fst}}([\,]) = \textbf{fail}$

$\qquad\qquad\quad \textbf{Synth}^{\textbf{fst}}((S_c \dashv S_c'){:}\mathsf{xs}) = \textbf{let } \mathsf{f} = \textbf{Synth}(S_c \dashv S_c') \textbf{ in } (\textbf{if } \mathsf{f} \neq \textbf{fail then } (\mathsf{f},c) \textbf{ else } \textbf{Synth}^{\textbf{fst}}(\mathsf{xs}))$

Figure 5: The algorithm **Synth**.

possible stuck states can be avoided by means of orchestrating processes. In the present paper the points where an orchestrator can affect a computation are made visible in the calculus by introducing a novel operator that we dub *speculative selection*. This implies extending the syntax of usual session types, so resulting in a session-type counterpart of the retractable session contracts of [4], where backtracking is taken into account instead of orchestration. Typable processes are shown to be free from errors. An interpretation of speculative selection as an actual language construct is then provided in form of priority selection.

Recursion has not been considered in the present paper, in order to focus on the main ideas and differences with other session-types formalisms. Adding recursion should not pose any technical difficulties and can be dealt with, for types and orchestrators, along the lines of [18] and [6]. Recursion for processes can be treated as done, among others, in [13].
Besides adding recursion, the present formalism could be extended with a notion of subtyping. A subtyping relation for our type system can be formalized following ideas present in [2]. It should hence be possible to define our orchestrated compliance by composing its symmetric restriction with the subtype relation. For what concerns the treatment of higher-order in the subtype relation, one could take into account investigations like those carried on in [2, 9] for the session-contracts formalism.

An interesting extension of the present investigation is adapting speculative choices and orchestrated interactions in the setting of multiparty asyncronous sessions. In case one considered the multiparty session types of [14], it would be reasonable to associate an orchestrator to each channel involved in a multiparty session. The orchestrator would hence act as an input filter for the buffer associated to a channel. Of course the introduction of speculative selection in a multiparty scenario would imply to reconsider the relationship between global types and local types. Actually no speculative-selection type can reasonably come out by projecting a global type. So processes (and their related types) using

speculative selections can be looked at as modules one can adapt, by means of orchestrators, in order to comply (precisely or at least[3]) with the global interaction pattern represented by the global type. From a different perspective a global type, if any, could be synthesized out of a number of local types, similarly to what done in [17]. In our "speculative" setting, however, a global type should rather be considered as a global adaptor, a centralized orchestrator, something similar to the *medium process* of [10] whose action would now be restricted to driving the speculative choices.

By taking into account the discussion of Remark 2.6, it would be interesting to define a *complementary relation* as the composition of compliance and subtyping and investigating whether in our setting it can be interpreted as a generalization of duality.

References

[1] F. Barbanera and U. de'Liguoro. Two notions of sub-behaviour for session-based client/server systems. In *PPDP*, pages 155–164. ACM Press, 2010. `doi:10.1145/1836089.1836109`.

[2] F. Barbanera and U. de'Liguoro. Sub-behaviour relations for session-based client/server systems. *MSCS*, 25(6):1339–1381, 2015. `doi:10.1017/S096012951400005X`.

[3] F. Barbanera and U. de'Liguoro. A game interpretation of retractable contracts. In *Proceeding COORDINATION 2016*, volume 9686 of *LNCS*, pages 18–34. Springer, 2016. `doi:10.1007/978-3-319-39519-7_2`.

[4] F. Barbanera, M. Dezani-Ciancaglini, I. Lanese, and U. de'Liguoro. Retractable contracts. In *PLACES 2015*, volume 203 of *EPTCS*, pages 61–72. Open Publishing Association, 2016. `doi:10.4204/EPTCS.203.5`.

[5] F. Barbanera, I. Lanese, and U. de'Liguoro. Retractable and speculative contracts. In *COORDINATION'17*, volume 10319 of *LNCS*. Springer, 2017. `doi:10.1007/978-3-319-59746-1`.

[6] Franco Barbanera, Steffen van Bakel, and Ugo de'Liguoro. Orchestrated session compliance. *J. Log. Algebr. Meth. Program.*, 86(1):30–76, 2017. `doi:10.1016/j.jlamp.2016.08.002`.

[7] G. T. Bernardi and M. Hennessy. Modelling session types using contracts. *Mathematical Structures in Computer Science*, 26(3):510–560, 2016. `doi:10.1017/S0960129514000243`.

[8] Giovanni Bernardi, Ornela Dardha, Simon J. Gay, and Dimitrios Kouzapas. On duality relations for session types. In *TGC 2014*, pages 51–66. Springer, 2014. `doi:10.1007/978-3-662-45917-1_4`.

[9] Giovanni Bernardi and Matthew Hennessy. Using higher-order contracts to model session types. *Logical Methods in Computer Science*, 12(2), 2016. `doi:10.2168/LMCS-12(2:10)2016`.

[10] Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In *FORTE 2016*, pages 74–95. Springer, 2016. `doi:10.1007/978-3-319-39570-8_6`.

[11] G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. *ACM Trans. on Prog. Lang. and Sys.*, 31(5):19:1–19:61, 2009. `doi:10.1145/1538917.1538920`.

[12] Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005. `doi:10.1007/s00236-005-0177-z`.

[13] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based progr. In *ETAPS'98*, volume 1381 of *LNCS*, 1998. `doi:10.1007/BFb0053567`.

[14] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. `doi:10.1145/2827695`.

---

[3]In such a case, clean-up reductions would turn out to be useful also in the multiparty setting, where it is reasonable not to have any particular bias towards one of the participants in a multiparty session.

[15] C. Laneve and L. Padovani.   The must preorder revisited: An algebraic theory for web ser-vices contracts.   In *CONCUR'07*, volume 4703 of *LNCS*, pages 212–225. Springer, 2007. `doi:10.1007/978-3-540-74407-8\_15`.

[16] Cosimo Laneve and Luca Padovani. The Pairing of Contracts and Session Types. In *ICGT'08*, volume 5065 of *LNCS*, pages 681–700. Springer, 2008. `doi:10.1007/978-3-540-68679-8\_42`.

[17] Julien Lange and Alceste Scalas. Choreography synthesis as contract agreement. In *ICE '13*, volume 131 of *EPTCS*, pages 52–67. Open Publishing Association, 2013. `doi:10.4204/EPTCS.131.6`.

[18] L. Padovani. Contract-based discovery of web services modulo simple orchestrators. *Theoretical Computer Science*, 411:3328–3347, 2010. `doi:10.1016/j.tcs.2010.05.002`.

[19] Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007. `doi:10.1016/j.entcs.2007.02.056`.