Orchestrated compliance  
for session-based client/server interactions*

(Proceedings of 8th Interaction and Concurrency Experience, EPTCS 189, pp. 21-36)

Steffen van Bakel  
Imperial College London  
London, UK  
svb@doc.ic.ac.uk

Franco Barbanera  
Università di Catania  
Catania, Italy  
barba@dmi.unict.it

Ugo de’Liguoro  
Università di Torino  
Torino, Italy  
deliguoro@di.unito.it

s.vanbakel@imperial.ac.uk

Abstract
We investigate the notion of orchestrated compliance for client/server interactions in the context of session contracts. Devising the notion of orchestrator in such a context makes it possible to have orchestrators with unbounded buffering capabilities and at the same time to guarantee any message from the client to be eventually delivered by the orchestrator to the server, while preventing the server from sending messages which are kept indefinitely inside the orchestrator. The compliance relation is shown to be decidable by means of 1) a procedure synthesising the orchestrators, if any, making a client compliant with a server, and 2) a procedure for deciding whether an orchestrator behaves in a proper way as mentioned before.

1 Introduction

Session types and contracts are two formalisms used to study client/server protocols. Session types have been introduced in [18] as a tool for statically checking safe message exchanges through channels. Contracts, on the other hand, as proposed in [13, 20, 14, 15], are a subset of CCS without \( \tau \), that address the problem of abstractly describing behavioural properties of systems by means of process algebra. In between these two formalisms lie session contracts\(^1\) as introduced in [2, 4, 9, 10]; this is a formalism interpreting the session types into a subset of contracts.

In the theory of contracts, as well as in the formalism of session contracts, the notion of compliance plays a central role. A client \( \rho \) is defined as being compliant with a server \( \sigma \) (written as \( \rho \vdash \sigma \)) whenever all of its requests are satisfied by the server. Now it might be the case that client satisfaction cannot be achieved just because of a difference in the order in which the partners exchange information, or because one of them provide some extra unneeded information.

Consider the example of a meteorological data processing system (MDPS) that is permanently connected to a weather station to which it sends, according to its processing needs, particular data requests. For the sake of simplicity, we consider just two particular requests, namely for temperature and humidity. After the requests, the MDPS expects to receive the

\* This work was partially supported by COST Action IC1201 BETTY, MIUR PRIN Project CINA Prot. 2010LHT4KM and Torino University/Compagnia San Paolo Project SALT.

\(^1\) They were dubbed session behaviours in [2, 4]. For sake of uniformity and since session contract sounds more appealing, we adhere here to that name.
data in the order they were requested. In the session-contracts formalism the interface for the simplified MDPS can be stated as follows:

$$\text{MDPS} = \text{rec} \times \text{tempReq} \cdot \text{humReq} \cdot \text{temperature} \cdot \text{humidity} \cdot x$$

(Here, as in CCS, a symbol like ‘a’ stands for on input action, whereas ‘!’ denotes the corresponding output). We assume a weather station to be able to send back the asked-for information in the order decided by its sensors, interspersed with information about wind speed:

$$\text{WeatherStation} = \text{rec} \times \text{tempReq} \cdot \text{humReq} \cdot (\text{temperature} \cdot \text{humidity} \cdot \text{wind} \cdot x \oplus \text{humidity} \cdot \text{temperature} \cdot \text{wind} \cdot x)$$

With the standard notion of compliance, it is not difficult to check that $$\text{MDPS} \not\dashv \text{WeatherStation},$$ since the client MDPS has no input action for the wind data, and also since it could occur that the temperature and humidity data are delivered in a different order than expected by the MDPS.

A natural solution to this would consist of devising a process that acts as a mediator (here called orchestrator) between the client and the server, coordinating them in a centralised way in order to make them compliant. This sort of solution is the one adopted in the practice of web-service interaction, in particular for business processes, where the notion of orchestration has been introduced and developed:

“Orchestration: Refers to an executable business process that may interact with both internal and external web services. Orchestration describes how web services can interact at the message level, including the business logic and execution order of the interactions.” [22]

In the context of the theory of contracts, this solution was formalised and investigated by Padovani [21], where orchestrators are processes that cannot affect the internal decisions of the client nor of the server, but can affect the way their synchronisation is carried out.

The orchestrating actions an orchestrator can perform have the following forms:

- $$(\text{a}, \text{t})$$ (resp. $$(\text{t}, \text{a})$$): the orchestrator gets a from the client (resp. server) and immediately delivers it to the server (resp. client) in a synchronous way.
- $$(\text{a}, \epsilon)$$ (resp. $$(\epsilon, \text{a})$$): the orchestrator gets a from the client (resp. server) and stores it in the buffer.
- $$(\epsilon, \text{t})$$ (resp. $$(\text{t}, \epsilon)$$): the orchestrator takes a from the buffer and sends it to the client (resp. server).

So a possible orchestrator enabling compliance for our example would be

$$f = \text{rec} \times (\text{tR}, \epsilon) \cdot (\text{hR}, \epsilon) \cdot (\epsilon, \text{t}) \cdot (\epsilon, \text{h}) \cdot (\epsilon, \text{w}) \cdot x$$

where $\text{tR}$, $\text{hR}$, $\text{t}$, $\text{h}$, and $\text{w}$ stand for tempReq, humReq, temperature, humidity, and wind, respectively. The orchestrator f rearranges the order of messages when necessary, and retains the wind information, not needed by MDPS.

Actually, the orchestrator f is not a valid orchestrator in the sense of [21]: indeed the wind information is never delivered to the client (i.e. it is implicitly discarded), so that the buffer corresponding to f would be unbounded. Unbounded buffers are not allowed in [21], where boundedness of buffers is used to guarantee both decidability and the possibility of synthesising orchestrators. In a session setting instead, as is the present one, decidability and or-
chestrators synthesis can be established even in presence of unbounded buffering capabilities of orchestrators.

In a two-parties session-based interaction, the choice among several continuations always depends on just one of the two actors. To let our formalism fully adhere to such a viewpoint our *session orchestrators*, besides (as argued in [21]) being processes that cannot affect the internal decision of the client or the server, are such that they do not create any non-determinism besides that already present in the partners. This will correspond to restricting the syntax in such a way that orchestrators like, for instance, $(\epsilon,a)\cdot f_1 \lor (b,\epsilon)\cdot f_2$, are not allowed. In fact, in the latter orchestrator, the choice of receiving an input from the client or from the server would not depend solely on the partners. The $f$ described above does respect this syntax restriction.

Moreover, in our system it will be possible to prove that $$f : \text{MDPS} \vdash \text{WeatherStation}$$ i.e.: MDPS and WeatherStation manage to be compliant (represented by $\vdash$ in our context) when their interaction is mediated by $f$.

In our system we will also manage to prevent the presence of *fake orchestrated complying interactions*, like that between the client $\sigma_1$ and the server $\sigma_2$ through the orchestrator $(a,\sigma_1)\cdot (b,\epsilon)$. In this case the client gets the illusion that *all its requests* are satisfied, whereas its output $b$ never reaches the server, but will be indefinitely kept inside the orchestrator’s buffer. While in the contract setting of [21] such compliant interactions are allowed, in our session context we manage to rule out orchestrators behaving like $(a,\sigma_1)\cdot (b,\epsilon)$, which never deliver a message from the client to the server.

We shall prove that properties like the one just mentioned, characterising *well-behaved orchestrators*, are decidable. Given an $f$, decidability of orchestrated compliance through $f$ will be proved. We will also show that, given a client and a server, it is possible to synthesise *all* the orchestrators that make the client and system compliant, if any.

## 2 Session contracts and orchestrated compliance

Session contracts are a restriction of contracts [20, 15]. They are designed to be in one-to-one correspondence to session types [18] without delegation (in [2, 4] a version with delegation was investigated). The restriction consists in constraining internal and external choices in a way that limits the non-determinism to (internal) output selection.

**Definition 2.1 (Session Contracts)**

1) Let $\mathcal{N}$ be a countable set of symbols and $\overline{\mathcal{N}} = \{ \pi | a \in \mathcal{N} \}$. The set $\text{RSC}$ of raw session contracts is defined by the grammar in Figure 1, where:

- for external and internal choices, $n \geq 1$, and $a_i \in \mathcal{N}$ (hence $\overline{\pi_i} \in \overline{\mathcal{N}}$) for all $1 \leq i \leq n$;
- the variable $x$ is a *session-contract variable* out of a denumerable set; we consider occurrences of $x$ in $\sigma$ bound in $\text{rec} \ x \cdot \sigma$. An occurrence of $x$ in $\sigma$ is free if it is not bound,

![Figure 1: The grammar of raw session-contracts](image-url)

---

$\sigma, \rho ::= 1$ success

$| a_1 \cdot \sigma_1 + \cdots + a_n \cdot \sigma_n$ external choice

$| \pi_1 \cdot \sigma_1 \oplus \cdots \oplus \pi_n \cdot \sigma_n$ internal choice

$| x$ variable

$| \text{rec} \ x \cdot \sigma$ recursion

---
and we write $\text{fv}(\sigma)$ for the set of free variables in $\sigma$. $\sigma$ is said to be closed whenever $\text{fv}(\sigma) = \emptyset$.

$\text{Act} = \mathcal{N} \cup \overline{\mathcal{N}}$ is the set of actions.

ii) The set SC of session contracts is the subset of closed raw session contracts such that in $a_1 \cdot \sigma_1 + \cdots + a_n \cdot \sigma_n$ and $\overline{a}_1 \cdot \sigma_1 + \cdots + \overline{a}_n \cdot \sigma_n$, the $a_i$ and the $\overline{a}_i$ are, in both, pairwise distinct; moreover, in $\text{rec} x . \sigma$ the expression $\sigma$ is not a variable.

As usual, we abbreviate $a_1 \cdot \sigma_1 + \cdots + a_n \cdot \sigma_n$ by $\sum_{i=1}^n a_i \cdot \sigma_i$, and $\overline{a}_1 \cdot \sigma_1 + \cdots + \overline{a}_n \cdot \sigma_n$ by $\bigoplus_{i=1}^n \overline{a}_i \cdot \sigma_i$. We also use the notations $\sum_{i \in I} a_i \cdot \sigma_i$ and $\bigoplus_{i \in I} \overline{a}_i \cdot \sigma_i$ for finite and non-empty $I$. We take the equi-recursive view of recursion, by equating $\text{rec} x . \sigma$ with $\sigma\{x/\text{rec} y . \sigma\}$.

The trailing 1 is normally omitted: for example, we will write $a + b$ for $a \cdot 1 + b \cdot 1$. Session contracts will be considered modulo commutativity of internal and external choices.

The operational semantics of session contracts is given in terms of a labeled transition system (LTS) $\sigma \xrightarrow{\alpha} \sigma'$ where $\sigma, \sigma' \in \text{SC}$ and $\alpha$ either belongs to a set of actions $\text{Act}$ or is an internal action $\tau$.

**Definition 2.2 (LTS for Session Contracts)** We define the labelled transition system $(\text{SC}, \text{Act}, \rightarrow)$ by the rules:

$\overline{a}_1 \cdot \sigma_1 + \cdots + \overline{a}_n \cdot \sigma_n \xrightarrow{\tau} \overline{a}_k \cdot \sigma_k$

$\overline{a} \cdot \sigma \xrightarrow{\pi} \sigma$

$a_1 \cdot \sigma_1 + \cdots + a_n \cdot \sigma_n \xrightarrow{\alpha} \sigma_k$

where $1 \leq k \leq n$, and $\sigma \xrightarrow{\alpha} \sigma'$ is short for $(\sigma, \alpha, \sigma') \in \rightarrow$. We shall use $\rightarrow$ as shorthand for $\xrightarrow{\tau}$. As usual, we write $\Rightarrow \rightarrow$ for $\rightarrow^*$ and $\xrightarrow{\alpha} \rightarrow$ for $\rightarrow^* \xrightarrow{\alpha} \rightarrow^*$ with $\alpha \in \text{Act}$.

Notice that reduction is not defined through contextual rules, so reduction only takes place at the ‘top’ level. Thereby, it is impossible for $\text{rec} x . \sigma$ to unfold more than once without consuming a guard (remember that $\sigma$ is not a variable): so recursion is contractive in the usual sense. We will safely assume that no two consecutive rec binders (as in $\text{rec} x . \text{rec} y . \sigma$) are present in a session contract.

We observe that $\xrightarrow{\alpha}$ is well defined, in that if $\sigma \in \text{SC}$ and $\sigma \xrightarrow{\alpha} \sigma'$ (or $\sigma \Rightarrow \rightarrow \sigma'$), then $\sigma' \in \text{SC}$.

**Session orchestrators** As also done in [21] in the context of the theory of contracts, we intend to investigate the notion of compliance when the interaction between a client and a server are mediated by an orchestrator. Different from the contract setting, the session setting we are in induces some natural restrictions to the syntax of orchestrators, making it safe to have orchestrators with unbounded buffers. Moreover, it is possible to check whether any output from the client is eventually delivered by the orchestrator to the server, as well as whether there might be an infinite interaction which falsely progresses because it is made only of outputs from the server to the orchestrator (see Section 4).

The set of actions an orchestrator can perform, that we take from [21], have been informally described in the introduction.\(^2\)

It can be reasonably argued that orchestrators must not show any internal non-determinism. Taking into account now the session-based interactions of our setting, such an assumption should be further extended, keeping in mind that in a session-based client/server interaction

\(^2\) One could wonder whether just asynchronous orchestration actions can be taken into account, since any $\langle a, \overline{a} \rangle$ action can be safely mimicked by two asynchronous ones, namely $\langle a, \epsilon \rangle, \langle \epsilon, \overline{a} \rangle$ (similarly for $\langle \overline{a}, \overline{a} \rangle$). A difference in fact would arise only for what concerns implementation, since the protocol for a synchronous exchange would not involve the use of a buffer, which is instead necessary for asynchronous actions. Such an implementation issue seems unlikely to be related to our theoretical treatment. In contrast, we shall point out in Remark 4.5 how implementation related aspects might affect our formalisation.
any possible non-determinism is due only to the internal non-determinism of the two partners. We therefore define our session-orchestrators so as to enforce this point of view. It follows that the only choice we allow in session-orchestrators (represented by \( \lor \)) in expressions like \( f \lor g \) is an external one, and it is necessarily driven by the internal choice of one of the two partners. This implies that the actions immediately exhibited by \( f \) and \( g \) must have the same direction, i.e. must belong to just one of the two subsets \( \{ \langle a, e \rangle, \langle a, \pi \rangle \mid a \in \mathcal{N} \} \) or \( \{ \langle e, a \rangle \mid \langle \pi, a \rangle \mid a \in \mathcal{N} \} \). Besides, orchestration actions of the form \( \langle \pi, e \rangle \) or \( \langle e, \pi \rangle \) must be used just as prefixes \( \mu \) in orchestrators like \( \mu.f \).

The other ruled-out cases, like \( \langle \tau, e \rangle \cdot f' \lor \langle e, b \rangle \cdot g' \) or \( \langle c, e \rangle \cdot f' \lor \langle e, b \rangle \cdot g' \), would conflict with the session viewpoint or, like \( \langle \tau, e \rangle \cdot f' \lor \langle b, e \rangle \cdot g' \), would be meaningless according to the syntax of session contracts.

We now formally define orchestration actions by partitioning them into different syntactic categories.

**Definition 2.3 (Session-Orchestration Actions)** We define \( \text{OrchAct} \) as the set of session-orchestration actions described by the following grammar (where \( a \in \mathcal{N} \) and \( \pi \in \mathcal{N} \)):

\[
\begin{align*}
\mu & ::= \eta \mid \iota \mid \epsilon \\
\eta & ::= \langle a, e \rangle \mid \langle a, \pi \rangle \\
\iota & ::= \langle e, a \rangle \mid \langle \pi, a \rangle \\
\epsilon & ::= \langle \pi, e \rangle \mid \langle e, \pi \rangle
\end{align*}
\]

We let \( \mu, \mu', \mu_1, \ldots \) range over orchestration actions. By \( \mu \) we shall denote either a finite sequence \( \mu_1 \ldots \mu_n \) in \( \text{OrchAct}^* \) or an infinite sequence \( \mu_1 \ldots \mu_n \ldots \) in \( \text{OrchAct}^\omega \).

Session orchestrators are now defined as follows.

**Definition 2.4 (Session Orchestators)** We define \( \text{Orch} \) as the set of session orchestrators, ranged over by \( f, g, h, \ldots \), described by the closed terms generated by the following grammar:

\[
\begin{align*}
f, g & ::= 1 \\
& \mid \iota \cdot f_1 \lor \cdots \lor \iota \cdot f_n \quad (n \geq 1) \\
& \mid \iota \cdot f_1 \lor \cdots \lor \iota \cdot f_n \quad (n \geq 1) \\
& \mid \epsilon \cdot f \\
& \mid x \\
& \mid \text{rec } x \cdot f
\end{align*}
\]

We impose session orchestrators to be contractive, i.e. the \( f \) in \( \text{rec } x \cdot f \) is assumed to not be a variable.

The expression 1 represents the orchestrator offering no action. \( \epsilon \cdot f \) offers just the orchestration action of the category \( \epsilon \) and continues as \( f \), whereas \( \iota \cdot f_1 \lor \cdots \lor \iota \cdot f_n \) and \( \iota \cdot f_1 \lor \cdots \lor \iota \cdot f_n \) offer \( n \) (uni-directional) actions of the syntactical categories, respectively, \( \iota \) and \( \iota \). Recursive orchestrators can be expressed by means of the \( \text{rec } \) binder and recursion variables, in the usual way. As for session contracts, orchestrators are defined as to have recursion variables guarded by at least one orchestration action. In the following we shall often refer to ‘session orchestrators’ as simply ‘orchestrators.’ As for session contracts, we take an equi-recursive point of view, so identify \( \text{rec } x \cdot f \) and \( f \{ x / \text{rec } x \cdot f \} \).

We now define the operational semantics of orchestrators as an LTS.

**Definition 2.5 (LTS for Orchestrators)** We define the labelled transition system
(Orch, OrchAct, →) by the rules:

\[
\begin{align*}
\mu : f &\xrightarrow{\mu} f \\
f \xrightarrow{\mu} f' &\xrightarrow{f \lor g \xrightarrow{\mu} f'} \\
g \xrightarrow{\mu} g' &\xrightarrow{f \lor g \xrightarrow{\mu} g'}
\end{align*}
\]

Given a sequence \( \mu \), we write \( f \xrightarrow{\mu} \) whenever \( f \xrightarrow{\mu_1} f_1 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_n} f_n \) if \( \mu = \mu_1 \cdots \mu_n \in \text{OrchAct}^* \), or \( f \xrightarrow{\mu_1} \cdots \mu_n \xrightarrow{\mu_{n+1}} \cdots f_n \) if \( \mu = \mu_1 \cdots \mu_n \cdots \in \text{OrchAct}^\infty \). We write \( f \not\xrightarrow{\mu} \) if there is no \( \mu \) such that \( f \xrightarrow{\mu} \).

**Definition 2.6 (Orchestrator Traces)** Let \( f \in \text{Orch} \).

i) The set \( \text{Tr}(f) \subseteq (\text{OrchAct}^* \cup \text{OrchAct}^\infty) \) of traces of \( f \) is defined by: \( \text{Tr}(f) = \{ \mu | f \xrightarrow{\mu} \} \).

ii) The set \( \text{MaxTr}(f) \subseteq (\text{OrchAct}^* \cup \text{OrchAct}^\infty) \) of maximal traces of \( f \) is defined by

\[
\text{MaxTr}(f) = \{ \mu \in \text{Tr}(f) | \exists f' [ f \xrightarrow{\mu} f' \not\xrightarrow{\mu} ] \text{ or } \mu \in \text{OrchAct}^\infty \}
\]

As in [21], we define an *orchestrated system* as a triple \( (\rho, f, \sigma) \) (written \( \rho \parallel_f \sigma \)) of a (client) session-contract \( \rho \) and a (server) session contract \( \sigma \) interacting with each other while being supervised by the orchestrator \( f \).

For the operational semantics of orchestrated systems, we define labelled reductions instead of a reduction relation (as done in [21]). We label orchestrated-systems’ transitions by the orchestration actions which make them possible, since in our setting we need to check for particular conditions of orchestrator buffers after the evolution of an orchestrated system. A buffer can be explicitly coupled with an orchestrator or can be represented implicitly by the actions performed by the orchestrator. The latter is the choice of [21], that we maintain.

**Definition 2.7 (Orchestrated Systems operational semantics)**

i) The operational semantics of orchestrated systems is defined as follows:

\[
\begin{align*}
\rho &\rightarrow \rho' \\
\rho \parallel_f \sigma &\rightarrow \rho' \parallel_f \sigma' \\
\sigma &\rightarrow \sigma'
\end{align*}
\]

\[
\begin{align*}
\rho \xrightarrow{\alpha} \rho' &\xrightarrow{f \langle \alpha, a \rangle} f' \sigma \xrightarrow{\alpha} \sigma' \\
\rho \parallel_f \sigma &\xrightarrow{\alpha} \rho' \parallel_f \sigma' \\
f \xrightarrow{\alpha} f' &\xrightarrow{\langle \alpha, a \rangle} \sigma \xrightarrow{\alpha} \sigma'
\end{align*}
\]

ii) We define \( \rho \parallel_f \sigma \not\rightarrow \) as: \( \rho \parallel_f \sigma \not\rightarrow \) &\( \Leftrightarrow \exists \mu [ \rho \parallel_f \sigma \xrightarrow{\mu} ] \).

We write \( \xrightarrow{\mu} \) for \( \not\rightarrow \circ \rightarrow \circ \rightarrow \), and \( \mu \) for \( \not\rightarrow \mu_1 \circ \rightarrow \circ \rightarrow \mu_n \) (resp. \( \not\rightarrow \circ \rightarrow \circ \rightarrow \circ \rightarrow \)) if \( \mu \) is finite (resp. infinite).

We now define a notion of compliance which is coarser than expected because of possible unfair behaviour of the orchestrators, which will be refined in Section 4.

**Definition 2.8 (Disrespectful and Strict Orchestrated Compliance)** An orchestrator \( f \) is said to be \( \rho, \sigma \) strict whenever, for any finite \( \mu \), \( f \xrightarrow{\mu} \) implies \( \rho \parallel_f \sigma \xrightarrow{\mu} \). We define:

i) \( f : \rho \xrightarrow{\mu} \sigma \) if \( \rho \rho' \) is \( \rho \sigma \) strict, and for any \( \mu, \rho' \) and \( \sigma' \), the following holds:

\[
\rho \parallel_f \sigma \xrightarrow{\mu} \rho' \parallel_f \sigma' \not\rightarrow \implies \mu' = 1.
\]

ii) \( f : \rho \xrightarrow{\mu} \sigma \) if \( \exists f [ f : \rho \xrightarrow{\mu} \sigma ] \).
3 Orchestrators Synthesis

In this section we define an inference system $\triangleright^{\triangleright}$ for (possibly open) orchestrators to deduce that $f : \rho \rightarrow^{\triangleright} \sigma$, under finitely many assumptions of a certain shape. We first establish that the system is sound with respect to the $\rightarrow^{\triangleright}$ relation. Then, on the basis of that system, we provide an algorithm Synth for orchestrator synthesis which, given $\rho$ and $\sigma$, returns the set of all orchestrators $f$ such that $f : \rho \rightarrow^{\triangleright} \sigma$ (namely with $\Gamma = \emptyset$) and hence that $f : \rho \rightarrow^{\triangleright} \sigma$. The algorithm is essentially an exhaustive proof search for $\triangleright$ that can be shown to be always terminating.

**Definition 3.1 (The Orchestrators Inference System $\triangleright^{\triangleright}$)** The judgements of the system are expressions of the form $\Gamma \triangleright^{\triangleright} f : \rho \rightarrow^{\triangleright} \sigma$, where $\rho, \sigma \in SC$, $f$ is a (possibly open) orchestrator and $\Gamma$ is a set of assumptions of the form $x : \rho_1 \rightarrow^{\triangleright} \sigma_1$ such that: $x : \rho_1 \rightarrow^{\triangleright} \sigma_1 \in \Gamma \& \ y : \rho_1 \rightarrow^{\triangleright} \sigma_1 \in \Gamma \Longrightarrow x = y$ (so $\Gamma$ represents an injective mapping from variables to expressions of the form $\rho_1 \rightarrow^{\triangleright} \sigma_1$). The axioms and rules of the system are described in Figure 2.

In the inference system of Figure 2 the symbol $\rightarrow^{\triangleright}$ is a relation symbol representing the relation $\rightarrow^{\triangleright}$ as defined in Definition 2.8. In order to give the intuition behind the inference system, let us briefly comment on one of the rules, say (Cn+L). In case it is possible to show that $f'$ is an orchestrator for $f_p : \rho \rightarrow^{\triangleright} \sigma$, orchestrated compliance can be obtained for $\sum_{i \in I} a_i : \rho_i \rightarrow^{\triangleright} \sigma_i$ by means of $\langle \rho_p, \sigma \rangle : f'$, since the $\langle \rho_p, \sigma \rangle$ action satisfies one of the input requests $a_i$. In case $x \notin fn(f')$, we get that $\text{rec} x \langle \rho_p, \sigma \rangle : f' = \langle \rho_p, \sigma \rangle : f'$. This means that axiom (Ax) has been used in the derivation of $f'$ and the interaction between $\sum_{i \in I} a_i : \rho_i$ and $\sigma$ finitely succeeds if the actions described in the branch from (Cn+L) to (Ax) are performed. In case $x \in fn(f')$, rule (Hvr) has been used for $f'$, and a successful infinite interaction is possible between $\sum_{i \in I} a_i : \rho_i$ and $\sigma$ when the orchestrator repeatedly performs the actions in the branch from (Cn+L) to (Hvr), as described by the recursive orchestrator $\text{rec} x \langle \rho_p, \sigma \rangle : f'$.

**Definition 3.2 (Judgment Semantics)** Let $\Gamma = \{ x_1 : \rho_1 \rightarrow^{\triangleright} \sigma_1, \ldots, x_k : \rho_k \rightarrow^{\triangleright} \sigma_k \}$, and $\theta$ be a map
\[ \text{Synth}(\Gamma, \rho, \sigma) = \]

\[
\text{if } x : \rho \dashv \vdash \sigma \in \Gamma \text{ then } \{ x \} \\
\text{else if } \rho = 1 \text{ then } \{ 1 \} \\
\text{else if } \rho = \sum_{i \in I} a_i \cdot \rho_i \text{ and } \sigma = \sum_{i \in I} a_i \cdot \sigma_i \text{ then} \\
\quad \text{let } \Gamma' = \Gamma, \chi \vdash \sigma \text{ in} \\
\quad \{ \text{rec.} x.\langle \tau, \epsilon \rangle \cdot f \mid f \in \text{Synth}(\Gamma', \rho, \sigma) \} \\
\quad \text{else if } \rho = \bigoplus_{i \in I} a_i \cdot \rho_i \text{ and } \sigma = \bigoplus_{i \in I} a_i \cdot \sigma_i \text{ then} \\
\quad \text{let } \Gamma' = \Gamma, \chi \vdash \sigma \text{ in} \\
\quad \{ \text{rec.} x.\bigcup_{i \in I} \{ a_i \cdot f_i \mid f_i \in \text{Synth}(\Gamma', \rho, \sigma) \} \} \\
\quad \text{else if } f = \bigcup_{i \in I} a_i \cdot \rho_i \text{ and } \sigma = \bigcup_{i \in I} a_i \cdot \sigma_i \text{ and } K \subseteq I \text{ then} \\
\quad \text{let } \Gamma' = \Gamma, \chi \vdash \sigma \text{ in} \\
\quad \{ \text{rec.} x.\bigcup_{i \in I} \{ a_i \cdot f_i \mid f_i \in \text{Synth}(\Gamma', \rho, \sigma) \} \} \\
\quad \text{else } \emptyset \]

Figure 3: The algorithm Synth.

such that \( \theta(x_i) = f_i \), where the \( f_i \)s are proper (i.e. closed) orchestrators. Then we define:

\[
\theta \vdash \Gamma \quad \triangleq \quad \forall (x_i : \rho_i \dashv \vdash \sigma_i) \in \Gamma \; [\theta(x_i) : \rho_i \dashv \vdash \sigma_i] \\
\Gamma \vdash f : \rho \dashv \vdash \sigma \quad \triangleq \quad \forall \theta \; [\theta \vdash \Gamma \implies \theta(f) : \rho \dashv \vdash \sigma]
\]

where \( \theta(f) \) is the result of substituting, for all variables \( x \in f \), all free occurrences of \( x \) by \( \theta(x) \).

**Theorem 3.3 (Soundness)** If \( \Gamma \vdash^a f : \rho \dashv \vdash \sigma \) then \( \Gamma \vdash f : \rho \dashv \vdash \sigma \).

*Proof:* (Sketch\(^3\)) It is possible to devise a sound and complete system \( \triangleright \) for judgments of the shape \( \Gamma \triangleright f : \rho \dashv \vdash \sigma \), where \( f \) is a closed orchestrator and where \( \Gamma \) is a set of assumptions on closed orchestrators (not on variables as in \( \vdash^a \)). Now it can be proved that if \( f \) is closed and \( \Gamma \vdash^a f : \rho \dashv \vdash \sigma \) is derivable, then for any \( \theta \) such that \( \theta \vdash \Gamma \) we have \( \theta(\Gamma) \triangleright f : \rho \dashv \vdash \sigma \), where \( \theta(\Gamma) \) is the result of substituting all orchestrator variables \( x \) by \( \theta(x) \). Then the thesis follows from the soundness of \( \triangleright \).

The synthesis algorithm Synth is defined in Figure 3. Given a set of assumptions \( \Gamma \), a client \( \rho \) and a server \( \sigma \), the algorithm computes a set of orchestrators \( \mathcal{F} \) such that for all \( f \in \mathcal{F} \) a derivation of \( \Gamma \vdash^a f : \rho \dashv \vdash \sigma \) exists. The algorithm essentially mimics the rules of the inference system of Figure 2. Intuitively, in case we are looking for orchestrators for \( \rho = \bigoplus_{i \in I} a_i \cdot \rho_i \) and \( \sigma = \bigoplus_{i \in I} a_i \cdot \sigma_i \) under the assumptions \( \Gamma \), we notice that they can be inferred for such \( \rho \) and \( \sigma \) in system \( \vdash^a \) only by means of rules (Ctrl+R) or (Ctrl+L) and that their form is, respectively, \( \text{rec.} x.\bigcup_{i \in I} \{ a_i \cdot f_i \mid f_i \in \mathcal{F} \} \) or \( \text{rec.} x.\bigcup_{i \in I} \{ a_i \cdot f_i \mid f_i \in \mathcal{F} \} \), where the \( f_i \)s and the \( f_i \)s are the orchestrators for the pairs \( \rho_i, \sigma_i \) and \( \rho_i, \sigma_i \), respectively. This accounts for the fourth clause in the synthesis algorithm. We can prove the algorithm to be sound.

\(^3\) See [1] for details.
Lemma 3.4 If $\text{Synth}(\Gamma, \rho, \sigma) = \mathcal{F} \neq \emptyset$ then, for all $f \in \mathcal{F}$, $\Gamma \vdash^g f : \rho \vdash^g \sigma$ is derivable.

On the other hand, the algorithm is complete in the following sense:

Lemma 3.5 If $f : \rho \vdash^g \sigma$ and $\text{Synth}(\emptyset, \rho, \sigma)$ terminates then $f \in \text{Synth}(\emptyset, \rho, \sigma)$.

It remains to show that $\text{Synth}$ is terminating:

Lemma 3.6 For all $\Gamma, \rho$ and $\sigma$, the execution of $\text{Synth}(\Gamma, \rho, \sigma)$ terminates.

Proof: (Sketch) The proof is based on the fact that all session contracts in the recursive calls of $\text{Synth}$ are either a sub-expression of $\rho$ or $\sigma$, or a sub-expression of the session contracts occurring in the judgments in $\Gamma$ (which is finite). Since session contracts are regular trees, their sub-expressions are a finite set, so that the test $x : \rho \vdash^g \sigma \in \Gamma$ (where $x$ is any variable) at the beginning of $\text{Synth}$ cannot fail infinitely many times.

We draw the consequences of the above lemmas and of the soundness theorem in the next corollary:

Corollary 3.7 The relation $\vdash^g$ is decidable; moreover if $\rho \vdash^g \sigma$ then there is a finite and computable set $\mathcal{F}$ containing all the orchestrators $f$ such that $f : \rho \vdash^g \sigma$.

4 Respectfulness

The definition of orchestrators implies they have buffering capabilities. The sort of buffer taken into account in [21], as well as by us, is made of a finite number of bi-directional buffers, one for each possible name handled by the orchestrator. A bi-directional buffer is actually made of two distinct buffers, one containing the messages received from the client that have to be delivered to the server, and the other one containing the messages received from the server that should be delivered to the client.

In [21] orchestrators are restricted to have bounded buffering capabilities and such a restriction is used in the proofs of several properties concerning contract orchestrators. In our setting we can eliminate that restriction, so allowing more client/server pairs to be compliant, like for instance $\text{rec}_x.a$ and $\text{rec}_x.\overline{a}.a$, and the example in the introduction.

We will now formalise the notion of buffer.

Definition 4.1 (Buffers) i) A bi-directional buffer $B$ is a set of the form $\{c_a a^s | a \in \mathcal{N}\}$ where, for any $a \in \mathcal{N}$, $c_a, s_a \in \mathbb{Z}$. The $c_a$ in $c_a a^s$ represents the number of $a$'s in the part of the buffer containing messages sent by the client to the server. The $s_a$ in $c_a a^s$ represents the number of $a$'s in the part of the buffer containing messages sent by the server to the client.

ii) We define: $\tilde{\emptyset} = \{0.a^0 | a \in \mathcal{N}\}$ and

\[
|a|B = (B \setminus \{c_a a^s\}) \cup \{c_a a^{s+1}\} \quad \quad |a|B^+ = (B \setminus \{c_a a^s\}) \cup \{c_a a^{s+1}\}
\]

\[
\overline{a}B = (B \setminus \{c_a a^s\}) \cup \{c_a a^{s-1}\} \quad \quad \overline{a}B^+ = (B \setminus \{c_a a^s\}) \cup \{c_a a^{s+1}\}
\]

iii) We denote by $|B|_a$ the number of $a$'s in the server-to-client part of the buffer, i.e. $|B|_a = s_a$ and similarly for the client-to-server part, i.e. $a|B| = c_a$.

--

4 See [1] for details.
Definition 4.2 Given \( \mu \in \text{OrchAct}^* \cup \text{OrchAct}^\omega \), we define \( _a\mu \), its left-restriction to a name \( a \),
as follows (λ is the empty sequence):

\[
\begin{align*}
  a|λ &= λ, \\
  a|(μμ') &= μ_a|μ', & \text{if } μ ∈ \{ ⟨ε,π⟩, ⟨a,ε⟩ \}, \\
  a|(μμ') &= a|μ' & \text{otherwise}.
\end{align*}
\]

**Definition 4.3 (Respectful sequences and orchestrators)** Let \( μ ∈ \OrchAct^* \cup \OrchAct^∞ \) and \( μ ∈ \OrchAct \).

i) Given \( S ⊆ N \), we say \( μ \) to be definitely-\( S \) whenever:

\[ \exists k \ ∀ m ≥ k \ [ \text{the } m\text{-th element of } μ \text{ belongs to } S] \]

For sets that are singletons we write ‘definitely-\( μ \)’ instead of ‘definitely-\{ \mu \}’.

ii) We say \( μ \) to be a sound sequence whenever:

\[ ∀ a ∈ N \ ∀ n ≤ |μ| \ [ a|\oi(μ_1 · · · μ_n) ≥ 0 \text{ and } |\oi(μ_1 · · · μ_n)|_a ≥ 0] \]

iii) We say \( μ \) to be client-respectful sequence whenever, for any \( a ∈ N \):

\[ a|μ \text{ is finite and } a|\oi μ = 0 \text{ or } a|μ \text{ is infinite and non-definitely-}⟨a,ε⟩ \]

iv) We say \( μ \) to be non-definitely server-inputted whenever:

\[ μ \text{ is infinite } ⇒ μ \text{ is non-definitely-}⟨ε,a⟩ \text{ } | \text{ } a ∈ N \]

v) We say \( μ \) to be respectful whenever \( μ \) is sound, client-respectful and non-definitely server-inputted.

vi) We say that an orchestrator \( f \) is respectful whenever every \( μ ∈ \MaxTr(f) \) is so.

We will look now at a few examples in order to get a better intuition about the above definition.

**Example 4.4**

- The finite sequence \( ⟨a,ε⟩·⟨ε,\overline{b}⟩·⟨ε,π⟩ \) is not respectful since it is not sound. In fact, for the name \( b \), we have that \( |\oi(⟨a,ε⟩·⟨ε,\overline{b}⟩·⟨ε,π⟩)|_b = −1 < 0 \).
- The sequence \( ⟨a,ε⟩·⟨b,ε⟩·⟨ε,π⟩ \) instead, is sound, but nonetheless it is not client-respectful, since it is not infinite and for the name \( b \) we have \( b|\oi(⟨a,ε⟩·⟨b,ε⟩·⟨ε,π⟩) = 1 ≠ 0 \).
- The orchestrator \( f = ⟨c,\overline{c}⟩·\rec x.(⟨π,a⟩ ∨ ⟨ε,b⟩·⟨c,\overline{c}⟩·x) \) is not respectful since it is not client-respectful. In fact, for the sequence \( μ = ⟨c,\overline{c}⟩·c·⟨b,\overline{b}⟩·⟨c,ε⟩·⟨b,\overline{b}⟩·c·⟨c,\overline{c}⟩·x \), we have that \( c|μ \) is infinite and \( c|μ = ⟨c,ε⟩·⟨c,\overline{c}⟩·c·⟨c,\overline{c}⟩·x \) is definitely-\( ⟨c,ε⟩ \). In fact, from the second element on it is made of \( ⟨ε,c⟩ \) actions.
- The orchestrator \( f = ⟨c,\overline{c}⟩·\rec x.(⟨π,a⟩ ∨ ⟨ε,b⟩·⟨c,\overline{c}⟩·x) ≠ −1 < 0 \).
- The orchestrator \( f = ⟨c,\overline{c}⟩·\rec x.(⟨π,a⟩ ∨ ⟨ε,b⟩·⟨c,\overline{c}⟩·x) \) is not respectful since it is not definitely server-inputted. In fact, the infinite sequence \( μ = ⟨c,\overline{c}⟩·⟨c,ε⟩·⟨ε,b⟩·⟨c,\overline{c}⟩·x \) is definitely-\( ⟨ε,a⟩ \) \text{ } | \text{ } a ∈ N \). The orchestrator \( ε \) in the introduction, instead, is non-definitely server-inputted, and also respectful, as a matter of fact.

**Remark 4.5** By Definition 4.2, the sequence \( a|μ \) in Definition 4.3((iii)) cannot contain synchronous orchestration actions like \( ⟨a,π⟩ \). Hence, for example, the orchestrator

\[ g = \rec x.(⟨a,ε⟩·⟨a,π⟩·x) \]

is not client-respectful, and so it is not respectful at all. This is because one \( a \) coming from the client will never be delivered to the server since any subsequent output \( π \) will be paired with a further input of \( a \). This might be irrelevant when distinct occurrences of the same message are indistinguishable, but in general the number of input-output actions matters.

On the other hand forcing the orchestrator to immediately forward a message is a desirable capability, which would be definitely lost by equating \( ⟨a,ε⟩·⟨ε,π⟩ \) to \( ⟨a,π⟩ \), and by ruling out
We can now properly define the full notion of compliance and characterise it.

**Definition 4.6 (Orchestrated Session Compliance)**  

i) We say that a client $\rho$ is compliant with a server $\sigma$ through the orchestration of $f$, and denote this by $f : \rho \dashv \sigma$, whenever  

\[
\rho \parallel f \sigma \mathrel{\overset{\mu}{\Longrightarrow}} \rho' \parallel f' \sigma' \not\dashv \quad \text{implies} \quad \rho' = 1 \quad \text{and} \quad \mu \text{ is respectful, and}
\]

ii) We write $\rho \dashv \sigma$ whenever there exists an orchestrator $f$ such that $f : \rho \dashv \sigma$.

Notice that we cannot define orchestrated compliance by simply imposing

\[
\langle \text{mediation of the orchestrator} \rangle \not\dashv
\]

Proposition 4.7  

i) $f : \rho \dashv \sigma \implies \exists f' [f' : \rho \dashv \sigma \text{ such that } f' \text{ is } \rho-\sigma \text{ strict}].$

ii) $f : \rho \dashv \sigma \text{ and } f \text{ is } \rho-\sigma \text{ strict} \iff f : \rho \dashv^\infty \sigma \text{ and } f \text{ is respectful.}$

iii) $\rho \dashv \sigma \iff \exists f [f : \rho \dashv^\infty \sigma \text{ where } f \text{ is respectful}].$

In order to show decidability, we provide a characterisation of respectfulness based on the notion of buffer-aware trees and its related labellings below.

**Definition 4.8 (Buffer-aware trees of $f$)**  

i) Let $a \in N$. We define the buffer-aware $a$-tree of an orchestrator $f$, denoted by $\mathcal{A}a(f)$, as the tree defined by induction in Figure 4. The edges of the tree have a left- and a right-weight denoting, respectively, the increment of the client-to-server and of the server-to-client buffer caused by the orchestration actions performed by $f$.

Given an edge $e$ of a buffer-aware $a$-tree $t$, we denote is left (resp. right) weight by $lw(e)$ (resp. $rw(e)$).

ii) We define the buffer-aware $+\text{-tree of an orchestrator } f$, denoted by $\mathcal{A}+(f)$, as the tree with the same nodes and edges as any $\mathcal{A}a(f)$, but such that the left (resp. right) weight of an edge $e$ is $\sum_{a \in N} lw(\mathcal{A}a(f))(e)$ (resp. $\sum_{a \in N} rw(\mathcal{A}a(f))(e)$).

Note that the left and right weights of the edges of a buffer-aware $+\text{-tree of an orchestrator } f$ are either 0, $-1$, or $+1$.

**Definition 4.9 (Buffer-labelling of $\mathcal{A}a(f)$)** We define the buffer-labelling of $\mathcal{A}a(f)$ by labelling its nodes with left and right labels as follows: given a node $N$ and the path $P$ in $\mathcal{A}a(f)$ from the root to $N$, we left-label $N$ with the sum of all the left-weights of the edges in $P$, whereas we right-label $N$ with the sum of all the right-weights of the edges in $P$.

We now provide characterisations for the properties defining respectfulness.

**Definition 4.10 (Sound buffer-labelling)** The buffer-labelling of $\mathcal{A}a(f)$ is sound whenever

i) there is no negative left-label and no negative right-label and

ii) for any leaf $x$ and corresponding $\text{rec } x$ node, if $k$ is the left (resp. right) label of $x$ and $h$ is the left (resp. right) label of $\text{rec } x$, then: $k - h \geq 0$.

Proposition 4.11 The following conditions are equivalent:
\[ \overline{\text{cTs}}^a(1) = 1 \quad \overline{\text{cTs}}^a(x) = x \]

\[
\begin{align*}
\overline{\text{cTs}}^a(\langle \varepsilon, \pi \rangle \cdot f') &= 0 \mid -1 \\
\overline{\text{cTs}}^a(\langle a, \varepsilon \rangle \cdot f') &= +1 \mid 0 \\
\overline{\text{cTs}}^a(\langle \pi, \varepsilon \rangle \cdot f') &= +1 \mid 0 \\
\overline{\text{cTs}}^a(\langle \varepsilon, a \rangle \cdot f') &= 0 \mid +1 \\
\overline{\text{cTs}}^a(\langle a, \varepsilon \rangle \cdot f') &= 0 \mid +1 \\
\overline{\text{cTs}}^a(\langle \varepsilon, a \rangle \cdot f') &= 0 \mid +1 \\
\overline{\text{cTs}}^a(f_1 \lor \ldots \lor f_n) &= \overline{\text{cTs}}^a(f_1) \ldots \overline{\text{cTs}}^a(f_n) \\
\overline{\text{cTs}}^a(\text{rec } x \cdot f') &= \text{rec } x \cdot \overline{\text{cTs}}^a(f') \\
\end{align*}
\]

\[ \text{Figure 4: Buffer-aware } a \text{-tree} \]

\begin{itemize}
  \item[i)] \( f \) is sound.
  \item[ii)] For any \( a \in \mathcal{N} \), the buffer-labelling of \( \overline{\text{cTs}}^a(f) \) is sound.
\end{itemize}

\textbf{Proof:} ((ii) \( \Rightarrow \) (i)) By the labelling, it is impossible to get a non-client-respectful sequence out of \( f \).

((i) \( \Rightarrow \) (ii)) By contraposition; assume that for a name \( b \in \mathcal{N} \), the buffer-labelling of \( \overline{\text{cTs}}^b(f) \) be unsound. Then we have two cases to consider:

i) There is a negative label. We then get immediately an unsound sequence.

ii) There exists a leaf \( x \) and its corresponding \( \text{rec } x \cdot \) node, if \( k \) is the left(or right-)label of \( x \) and \( h \) is the left-(or right-)label of \( \text{rec } x \cdot \), we have \( k - h < 0 \). It is immediate to get an unsound sequence.

We say that a node gets to \( 1 \) whenever its subtree contains a \( 1 \) node.

\textbf{Definition 4.12 (Client-Respectful Buffer-Labelling)} The buffer-labelling of \( \overline{\text{cTs}}^a(f) \) is client-respectful whenever

i) any \( 1 \) node is left-labelled with 0;

ii) for any leaf \( x \) and corresponding node of its binder \( \text{rec } x \cdot \), if \( k \) is the left-label of \( x \) and \( h \) is the left-label of \( \text{rec } x \cdot \), then

a) if the \( \text{rec } x \cdot \) node gets to \( 1 \), then \( h = k \);

b) otherwise, if all the left-labels of the edges from \( x \) to \( \text{rec } x \cdot \) are 0 then \( h = 0 \);

iii) for any path from a leaf \( x \) to its corresponding \( \text{rec } x \cdot \) node, either no edge is right-weighted with +1 or there is at least an edge with right-weight \(-1\).

\textbf{Proposition 4.13} The following conditions are equivalent:

i) \( f \) is client-respectful.

ii) For any \( a \in \mathcal{N} \), the buffer-labelling of \( \overline{\text{cTs}}^a(f) \) is client-respectful.

\textbf{Proof:} ((ii) \( \Rightarrow \) (i)) By the labeling rule it is impossible to get a non client-respectful sequence out of \( f \). For finite sequences this impossibility is guaranteed by clauses ((ii)) and ((iii)) of Definition 4.12, for infinite ones by clause ((iii)).

((i) \( \Rightarrow \) (ii)) By contraposition; assume that for a name \( b \in \mathcal{N} \), the buffer-labelling of \( \overline{\text{cTs}}^b(f) \) be non-client-respectful. We consider the four possible cases:
i) A label of a 1 leaf is not 0. In that case we immediately get a finite sequence out of $f$ which is non-client-respectful.

ii) There is a node $x$ labelled with $k$ and its corresponding node $\text{rec} \cdot x$. gets to 1 and it is labelled with $h$, with $k \neq h$. Then the sequence out of $f$ corresponding to going to $\text{rec} \cdot x$, then from node $\text{rec} \cdot x$ to $x$ a non negative number of times $n$ and finally to the 1 node cannot be client-respectful, since at the end the client-to-server buffer for $b$ would have $n \times (h - k)$ elements in it.

iii) There is a node $x$ labelled with $k$, its corresponding node $\text{rec} \cdot x$. does not get to 1, all the left-labels of the edges from $x$ to $\text{rec} \cdot x$. are 0 and $h = 0$. In that case the trace $\mu$ corresponding to the infinite path starting from the root and then keeping indefinitely on passing through $\text{rec} \cdot x$ and $x$ is such that $b \parallel \mu$ is finite and $|\mathcal{C}(b \parallel \mu)| \neq 0$.

iv) there exists a path from a leaf $x$ to its corresponding $\text{rec} \cdot x$. such that there are some right-weighted edges right-weighted with $+1$ and no edge with right-weight $−1$. Then it is immediate to get an infinite definitely server-inputted sequence out of $f$ which is definitely-$⟨b, ε⟩$ and hence not client-respectful.

Definition 4.14 (Non definitely server-inputted $*$-tree) Given an orchestrator $f$, its $*$-tree $\mathcal{C}f_*(f)$ is non-definitely server-inputted whenever, for any path from a leaf $x$ to its corresponding $\text{rec} \cdot x$. node, either no edge is right-weighted with $+1$ or there is at least an edge with right-weight $−1$.

Proposition 4.15 The following conditions are equivalent:

i) $f$ is not definitely server-inputted.

ii) $\mathcal{C}f_*(f)$ is non-definitely server-inputted.

Proof: ((ii) ⇒ (i)) By the labelling it is impossible to get a definitely server-inputted sequence out of $f$.

((i) ⇒ (ii)) By contraposition; assume that $\mathcal{C}f_*(f)$ be definitely server-inputted. So there exists a path from a leaf $x$ to its corresponding $\text{rec} \cdot x$. such that there are some right-weighted edges right-weighted with $+1$ and no edge with right-weight $−1$. Then it is immediate to get a definitely server-inputted sequence out of $f$.

In summary we have established:

Theorem 4.16 Orchestrator respectfulness is decidable.

From the above result and from decidability of $\vdash^*$ (Corollary 3.7) we can get decidability of $\vdash$. The algorithm to decide whether $\rho \vdash \sigma$ will first compute $\mathcal{F} = \text{Synth}(\emptyset, \rho, \sigma)$; then if $\mathcal{F} \neq \emptyset$ it suffices to check whether there is a strict and respectful $f \in \mathcal{F}$, which is a decidable problem by the above.

Theorem 4.17 Given $\rho$ and $\sigma$, it is decidable whether $\rho \vdash \sigma$.

We conclude by observing that in [21], the lack of unbounded buffering capabilities prevents orchestrators to be used to ensure client compliance with a server that might send an unbounded number of unnecessary outputs. To let such sort of interaction possible, in [3] the notion of skp-compliance (dubbed $\vdash^{skp}$) was investigated for session contracts, where a client is compliant with a server whenever all its requests can be satisfied thanks to the possibility of discarding a (possibly unbounded) number of unnecessary server outputs. Interactions of this sort can actually be carried out by means of our session orchestrators, since it is possible to prove that $\rho \vdash^{skp} \sigma$ implies $\rho \vdash^* \sigma$. In the example in the introduction, in fact, the wind
information is unbounded and ‘discarded’ by the orchestrator.

5 Related and future work

The notion of compliance naturally induces a substitutability relation on servers that may be used for implementing contract-based query engines (see [21] for a detailed discussion). Hence it seems worthwhile, as a development of the results of the present paper, to investigate the session sub-contract relation induced by our orchestrated compliance on session contracts. Whereas this sort of server substitutability relation is at the core of the results in [21], we deem it relevant to investigate also client substitutability, in the style of what was done in [2, 4] for session contract and in [11] for the more general notion of contract.

An approach to the formal description of service contracts in terms of automata has been recently developed in [7]. The notion of contract automaton is related to that of contract as well as of session contract. Besides, the notion of contract agreement in [7] somewhat resembles that of compliance. In the framework of that paper, orchestrators are synthesised to enforce contract composition to adhere to the requirements for contract agreement. Even if the authors of [7] work on the overall satisfaction in a multiparty composition of principals, it is definitely worthwhile, as a future investigation, to study the relation between the notion of orchestration, as developed in [21] and in the present paper, and the approach of [7], which in turn has been related in [8] to the model of choreography of communicating finite state machines (CFMS) [12]. For what concerns session contracts in particular, the investigation of the correspondence with the above mentioned formalisms could start from the result concerning the correspondence of binary session types with a particular two-communicating-machines subclass (see [17] for references). Such a correspondence between session types and communicating machines has been pushed further to the multiparty setting in [17].

Many properties of the model of CFMS which are untractable ceases to be so when Bags, instead of - or together with - FIFO queues are taken into account [16]. The similarity of contracts and session contracts with the CFMS model suggests to investigate the use of bags for session-contract interactions to reduce decidability problems in our context to problems in the CFMS model with bags. What does a bag correspond to in our context is however not immediate to device. In fact, by putting a bag in between $a.F$ and $a + b$ would result in a number of possible non-deterministic evolutions of the system: as soon as $a$ is in the bag, it could be used as input for the server; or, in case both $a$ and $b$ get into the bag, the server could non-deterministically choose amongst them; etc. Such a behaviour of the system, however, goes far beyond the session setting we are in, where non-determinism is restricted to occur only inside the client and server.

Session contracts have been also investigated in papers like [5, 6] where, overloading the name, they also have been dubbed session types. In [5] the authors establish a relation between session contracts and a model based on game-theoretic notions, showing that compliance corresponds to the existence of particular winning strategies. It should be interesting to investigate the meaning and role of the notion of orchestration in such a game-theoretical setting.

Acknowledgments. We are grateful to the referees for their helpful and meaningful advices. The interaction with them has been pleasing and fruitful thanks to the forum tool provided by the workshop organisation. We also wish to thank Mariangiola Dezani for her everlasting support.
References


