Journal of Logical and Algebraic Methods in Programming 86 (2017) 30–76

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Orchestrated session compliance

Franco Barbanera [a,*], Steffen van Bakel [b], Ugo de'Liguoro [c]

[a] Dipartimento di Matematica e Informatica, Università degli Studi di Catania, Viale A. Doria 6, 95125 Catania, Italy
[b] Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2BZ, UK
[c] Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy

ARTICLE INFO

Article history:
Received 15 November 2015
Received in revised form 1 August 2016
Accepted 5 August 2016
Available online 2 September 2016

Keywords:
Compliance
Session contracts
Orchestration
Subcontract

ABSTRACT

We investigate the notion of orchestrated compliance for client/server interactions in the context of *session contracts*. The orchestrators we study have unbounded buffering capabilities and, besides never sending messages which have not been received, are such that any message from the client is eventually delivered by the orchestrator to the server. Moreover, no infinite interaction can consist definitely of messages from the server which are kept by the orchestrator. The subcontract relation induced by this new notion of compliance is also investigated.

1. Introduction

Session types and contracts are two formalisms used to study client/server protocols. Session types have been introduced in [1] as a tool for statically checking safe message exchanges through channels. Contracts as proposed in [2–4] are a subset of CCS [5] without the silent action τ 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 [6–9], a formalism interpreting session types into a subset of contracts. Session contracts can be seen as binary session types without value or channel passing (that is, without the so-called "session delegation").

In the theory of contracts, as well as in the formalism of session contracts, the notion of *compliance* plays a central role. A client ρ is called *strongly compliant* with a server σ (written $\rho \dashv \sigma$) whenever *all* of its *requests* are satisfied by the server without ever blocking in any infinite interaction. Now it might be the case that client satisfaction fails just because of a difference in the order in which the partners exchange information, or because one of them provides 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, requests for particular data. For the sake of simplicity, we consider just two particular requests, namely for *temperature* and *humidity*. After sending the requests, the MDPS expects to receive the data in the order they were sent. In the session-contract formalism, the interface for this simplified MDPS can be stated as follows:

$$\mathsf{MDPS} = \mathsf{rec}\, x.\overline{\mathtt{tempReq}}.\overline{\mathtt{humReq}}.\mathtt{temperature}.\mathtt{humidity}.x$$

* Corresponding author.
E-mail addresses: barba@dmi.unict.it (F. Barbanera), svanbakel@imperial.ac.uk (S. van Bakel), ugo.deliguoro@unito.it (U. de'Liguoro).

[1] They were dubbed *session behaviours* in [6,7]. For sake of uniformity and since *session contract* expresses their properties more accurately, we adhere here to this name.

http://dx.doi.org/10.1016/j.jlamp.2016.08.002
2352-2208/

(Here, as in CCS, a symbol like 'a' stands for on input action, whereas '$\overline{a}$' 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*:

$$\mathsf{WeatherStation} = \mathsf{rec}\,x.\,\mathtt{tempReq}.\mathtt{humReq}.(\overline{\mathtt{temperature}}.\overline{\mathtt{humidity}}.\overline{\mathtt{wind}}.x \oplus \overline{\mathtt{humidity}}.\overline{\mathtt{temperature}}.\overline{\mathtt{wind}}.x)$$

With respect to strong compliance, we have that $\mathsf{MDPS} \not\dashv \mathsf{WeatherStation}$, since the client MDPS has no input action for the wind data, and also because it might happen that temperature and humidity data are delivered in a different order than expected by the MDPS. A natural solution is to devise 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. The notion of orchestration has been introduced in the setting of web-service interaction, in particular for business processes:

> "*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." [10]

In the context of the theory of contracts, this solution has been formalised and investigated by Padovani in [11], 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 communicating actions performed by an orchestrator have the following forms:

$\langle a, \overline{a}\rangle$ (resp. $\langle \overline{a}, a\rangle$): the orchestrator gets a from the client (resp. server) and immediately delivers it to the server (resp. client) in a synchronous way.
$\langle a, \varepsilon\rangle$ (resp. $\langle \varepsilon, a\rangle$): the orchestrator gets a from the client (resp. server) and stores it in a buffer.
$\langle \overline{a}, \varepsilon\rangle$ (resp. $\langle \varepsilon, \overline{a}\rangle$): the orchestrator takes a from the buffer and sends it to the client (resp. server).

So an orchestrator enabling compliance for our example is

$$\mathsf{f} = \mathsf{rec}\,x.\langle \mathtt{tR}, \overline{\mathtt{tR}}\rangle.\langle \mathtt{hR}, \overline{\mathtt{hR}}\rangle.(\langle \overline{\mathtt{t}}, \mathtt{t}\rangle.\langle \overline{\mathtt{h}}, \mathtt{h}\rangle.\langle \varepsilon, \mathtt{w}\rangle.x \vee \langle \varepsilon, \mathtt{h}\rangle.\langle \overline{\mathtt{t}}, \mathtt{t}\rangle.\langle \overline{\mathtt{h}}, \varepsilon\rangle.\langle \varepsilon, \mathtt{w}\rangle.x) \tag{1}$$

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

The orchestrator f is not a valid orchestrator according to [11]: indeed the $\mathtt{wind}$ information is discarded by never delivering it to the client, so that the buffer corresponding to f must be unbounded. Unbounded buffers are not allowed in [11], where boundedness is essential to guarantee decidability of the compliance relation and the possibility of synthesising orchestrators.

We investigate the notion of orchestrated compliance in the setting of session contracts, even in presence of unbounded buffering capabilities of orchestrators. In this system it will be possible to prove that

$$\mathsf{MDPS} \dashv\!\vdash_{\mathsf{f}} \mathsf{WeatherStation}$$

i.e. that MDPS and WeatherStation are compliant when their interaction is mediated as f (also denoted by $\mathsf{f} : \mathsf{MDPS} \dashv\!\vdash \mathsf{WeatherStation}$).

In a two-party session-based interaction, the choice among several continuations always depends on exactly one of the two actors. Hence *session orchestrators* are designed so that internal decisions of the parties are unaffected by the orchestration and any non-deterministic choice depends solely on the partners. This is obtained by restricting the syntax for orchestrators such that for instance orchestrators like $\langle \varepsilon, a\rangle.\mathsf{f}_1 \vee \langle b, \varepsilon\rangle.\mathsf{f}_2$ 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 on the partners.

In our system we aim at ruling out certain fake complying interactions, like between the client $\overline{a}.\overline{b}$ and the server a through the orchestrator $\langle a, \overline{a}\rangle.\langle b, \varepsilon\rangle$. In this case the client gets the illusion that all its requests are satisfied, whereas its output $\overline{b}$ never reaches the server, being indefinitely kept inside the orchestrator's buffer. While in the contract setting of [11] such interactions are allowed and these parties are compliant, in our context we forbid an orchestrators to behave like $\langle a, \overline{a}\rangle.\langle b, \varepsilon\rangle$ which never deliver a message from the client to the server.

Another sort of fake compliance we will prevent is that between a client like $\overline{a}.\overline{b}$ and a server like $a.\mathsf{rec}\,x.\overline{c}$ by means of the orchestrator $\langle a, \overline{a}\rangle.\mathsf{rec}\,x.\langle \varepsilon, c\rangle$. This is because this interaction would go on indefinitely even if after the satisfaction of the first client's request, the orchestrator indefinitely stores all the server's messages. Such unwanted behaviour of an orchestrator is automatically ruled out in [11] by the boundedness constraint on buffers, while we admit unbounded buffers. On the other hand, as done also in [11], we rule out orchestrators sending messages that they have never received.