Checking the conformance of orchestrations with respect to choreographies in web services: A formal approach *

Gregorio Díaz[1] and Ismael Rodríguez[2]

[1]Universidad de Castilla-La Mancha
Gregorio.Diaz@uclm.es

[2]Universidad Complutense de Madrid
isrodrig@sip.ucm.es

Abstract. In this paper we present a formal model to represent orchestrations and choreographies, and we provide some semantic relations to detect their *conformance*, i.e., whether a set of orchestrations representing some web services leads to the overall communications described in a choreography.

1 Introduction

We present a formal framework to define *models* of asynchronous web services as well as to study them. Our main goal is allowing to define orchestrations and choreographies as well as to *compare* them. That is, given the orchestration of some web services and a choreography defining how these web services should interact, we provide a diagnostic method to decide whether the interaction of these web services necessarily leads to the required observable behavior, i.e. whether the orchestration *conforms to* the choreography. Models of orchestrations and choreographies are constructed by means of two different languages, and some formal semantic relations define how the terms defined in both languages are compared. Our modeling languages focus on accurately defining asynchronous communication aspects. In particular, languages explicitly consider service identifiers, specific senders/addressees, message buffers, etc.

There are few related works that deal with the asynchronous communication in contracts for web service context. In fact, we are only aware of three works from van der Alst et al. [7], Kohei Honda et al. [4] and, Bravetti and Zavattaro [2]. In particular, van der Alst et al. [7] present an approach for formalizing compliance and refinement notions, which are applied to service systems specified using open Workflow Nets (a type of Petri Nets) where the communication is asynchronous. The authors show how the contract refinement can be done independently, and they check whether contracts do not contain cycles. Kohei Honda et al. [4] present a generalization of binary session types to multiparty sessions for π-calculus. They provide a new notion of types which can directly abstract the intended conversation structure among n-parties as *global scenarios*, retaining an intuitive type syntax. They also provide a consistency criteria for a conversation structure

* Research partially supported by projects TIN2006-15578-C02, PEII09-0232-7745, and CCG08-UCM/TIC-4124.

with respect to the protocol specification (contract), and a type discipline for individual processes by using a *projection*. Bravetti and Zavattaro [2] allow to compare systems of orchestrations and choreographies by means of the *testing* relation given by [1, 3]. Systems are represented by using a process algebraic notation, and operational semantics for this language are defined in terms of labeled transitions systems. On the contrary, our framework uses an extension of *finite state machines* to define orchestrations and choreographies, and a semantic relation based on the *conformance* relation [5, 6] is used to compare both models. In addition, let us note that [2] considers the suitability of a service for a given choreography *regardless* of the actual definition of the rest of services it will interact with, i.e. the service must be valid for the considered role *by its own*. This eases the task of finding a suitable service fitting into a choreography role: Since the rest of services do not have to be considered, we can search for suitable services for each role *in parallel*. However, let us note that sometimes this is not realistic. In some situations, the suitability of a service actually depends on the activities provided by the rest of services. For instance, let us consider that a *travel agency* service requires that either the *air company* service or the *hotel* service (or both) provide a transfer to take the client from the airport to the hotel. A hotel providing a transfer is *good* regardless of whether the air company provides a transfer as well or not. However, a hotel not providing a transfer is valid for the travel agency *only if* the air company does provide the transfer. This kind of subtle requirements and conditional dependencies is explicitly considered in our framework. Thus, contrarily to [2], our framework considers that the suitability of a service depends on what the rest of services actually do.

2 Formal model

In this section we present our languages to define models of orchestrations and choreographies. Some preliminary notation is presented next.

Definition 1. Given a type A and $a_1, \ldots, a_n \in A$ with $n \geq 0$, we denote by $[a_1, \ldots, a_n]$ the *list of* elements $a_1, \ldots, a_n$ of A. We denote the empty list by $[\,]$.

Given two lists $\sigma = [a_1, \ldots, a_n]$ and $\sigma' = [b_1, \ldots, b_m]$ of elements of type A and some $a \in A$, we have $\sigma \cdot a = [a_1, \ldots, a_n, a]$ and $\sigma \cdot \sigma' = [a_1, \ldots, a_n, b_1, \ldots, b_m]$.

Given a set of lists L, a *path-closure* of L is any subset $V \subseteq L$ such that for all $\sigma \in V$ we have

- either $\sigma = [\,]$ or $\sigma = \sigma' \cdot a$ for some σ' with $\sigma' \in V$.
- there do not exist $\sigma', \sigma'' \in V$ such that $\sigma \cdot a = \sigma'$ and $\sigma \cdot b = \sigma''$ with $a \neq b$.

We say that a path-closure V of L is *complete* in L if it is *maximal* in L, that is, if there does not exist a path-closure $V' \subseteq L$ such that $V \subset V'$. The set of all complete path-closures of L is denoted by $\texttt{Complete}(L)$. □

We present our model of web service *orchestration*. The internal behavior of a web service in terms of its interaction with other web services is represented by a *finite state machine* where, at each state s, the machine can receive an input i and produce an output o as response before moving to a new state s'. Moreover, each transition explicitly defines which service must send i: A sender identifier snd is attached to the transition denoting that, if i is sent by service snd, then the transition can be triggered. We assume that all web services are identified by a given identifier belonging to a set ID. Moreover, transitions also denote the *addressee* of the output o, which is denoted by an identifier adr. Let us note that web services receive messages asynchronously. This is represented in the model by considering an *input buffer* where all inputs received and not processed yet are cumulated.

Definition 2. Given a set of service identifiers ID, a *service* for ID is a tuple (id, S, I, O, s_{in}, T) where $id \in ID$ is the identifier of the service, S is the set of states, I is the set of inputs, O is the set of outputs, $s_{in} \in S$ is the initial state, and T is the set of transitions. Each transition $t \in T$ is a tuple (s, i, snd, o, adr, s') where $s, s' \in S$ are the initial and final states respectively, $i \in I$ is an input, $snd \in ID$ is the required sender of i, $o \in O$ is an output, and $adr \in ID$ is the addressee of o. A transition (s, i, snd, o, adr, s') is also denoted by $s \xrightarrow{(snd,i)/(adr,o)} s'$.

Given a service $M = (id, S, I, O, s_{in}, T)$, a *configuration* of M is a pair $c = (s, b)$ where $s \in S$ is a state of M and b is an *input buffer* for M. An input buffer for M is a list $[(id_1, i_1), \ldots, (id_k, i_k)]$ where $id_1, \ldots, id_k \in ID$ and $i_1, \ldots, i_k \in I$. The *initial configuration* of M is $(s_{in}, [\,])$. The set of all input buffers is denoted by $\mathcal{B}$.

Let $b = [(id_1, i_1), \ldots, (id_k, i_k)] \in \mathcal{B}$ with $k \geq 0$ be an input buffer. We define the following functions: $\texttt{exists}(b, id, i)$ holds iff $(id, i) \in \{(id_1, i_1), \ldots, (id_k, i_k)\}$; $\texttt{insert}(b, id, i) = b \cdot (id, i)$; $\texttt{remove}(b, id, i) = [(id_1, i_1), \ \ldots, \ (id_{j-1}, i_{j-1}), (id_{j+1}, i_{j+1}), \ldots, (id_k, i_k)]$ provided that $j \in \mathbb{N}$ is the minimum value such that $j \in [1..k]$, $id = id_j$, and $i = i_j$. □

Once we have presented our model of web service orchestration, we provide a way to compose services into systems. In formal terms, a system is a tuple of services. The configuration of a system is given by the tuple of configurations of each service in the system.

Definition 3. Let $ID = \{id_1, \ldots, id_p\}$. In addition, for all $1 \leq j \leq p$, let $M_j = (id_j, S_j, I_j, O_j, s_{j,in}, T_j)$ be a service for ID. We say that $\mathcal{S} = (M_1, \ldots, M_p)$ is a *system of services* for ID.

For all $1 \leq j \leq p$, let c_j be a configuration of M_j. We say that $c = (c_1, \ldots, c_p)$ is a *configuration* of $\mathcal{S}$. Let $c'_1, \ldots, c'_p$ be the initial configurations of $M_1, \ldots, M_p$, respectively. We say that $(c'_1, \ldots, c'_p)$ is the *initial configuration* of $\mathcal{S}$. □

Next we formally define how systems *evolve*, i.e. how a service of the system triggers a transition and how this affects other services in the system. In fact, the next definition presents the *operational semantics* of systems. In general, outputs of services will be considered as inputs of the services these outputs are sent to. Besides, we consider a special case of input/output that will be used to denote a *null* communication. In particular, if the input of a transition is *null* then we are denoting that the service can take this transition without waiting for any previous message from any other service, that is, we denote a *proactive* action of the service. Similarly, a *null* output denotes that no message is sent to other service after taking the corresponding transition. In both cases, the sender and the addressee of the transition are irrelevant, respectively, so in these cases they will also by denoted by a *null* symbol.

Definition 4. Let $ID = \{id_1, \ldots, id_p\}$ be a set of service identifiers and $\mathcal{S} = (M_1, \ldots, M_p)$ be a *system of services* for ID where for all $1 \leq j \leq p$ we have that $M_j = (id_j, S_j, I_j, O_j, s_{j,in}, T_j)$. Let $c = (c_1, \ldots, c_p)$ be a configuration of $\mathcal{S}$ where for all $1 \leq j \leq p$ we have $c_j = (s_j, b_j)$.

An *evolution* of $\mathcal{S}$ from the configuration c is a tuple $(c, snd, i, proc, o, adr, c')$ where $i \in I_1 \cup \ldots \cup I_p$ is the input of the evolution, $o \in O_1 \cup \ldots \cup O_p$ is the output of the evolution, $c' = ((s'_1, b'_1), \ldots, (s'_p, b'_p))$ is the new configuration of $\mathcal{S}$, and $snd, proc, adr \in ID$ are the sender, the processer, and the addressee of the evolution, respectively. All these elements must be defined according to one of the following choices:

(a) *(evolution activated by some service by itself)* For some $1 \leq j \leq p$, let us suppose $s_j \xrightarrow{(null,null)/(adr',o)} s' \in T_j$. Then, $s'_j = s'$ and $b'_j = b_j$. Besides, $snd = null$, $proc = id_j$, $adr = adr'$;

(b) *(evolution activated by processing a message from the input buffer of some service)* For some $1 \leq j \leq p$, let $s_j \xrightarrow{(snd',i)/(adr',o)} s' \in T_j$ and let us suppose $\texttt{exists}(b_j, snd', i)$ holds. Then, $s'_j = s'$ and $b'_j = \texttt{remove}(b_j, snd, i)$. Besides, $snd = snd'$, $proc = id_j$, and $adr = adr'$;

where, both in (a) and (b), the new configurations of the rest of services are defined according to one of the following choices:

(1) *(no message is sent to other service)* If $adr' = null$ or $o = null$ then for all $1 \leq q \leq k$ with $q \neq j$ we have $s'_q = s_q$ and $b'_q = b_q$.

(2) *(a message is sent to other service)* Otherwise, let $id_g = adr'$ for some $1 \leq g \leq k$. Then, we have $s'_g = s_g$ and $b'_g = \texttt{insert}(b_g, id_j, o)$. Besides, for all $1 \leq q \leq k$ with $q \neq j$ and $q \neq g$ we have $s'_q = s_q$ and $b'_q = b_q$. □

We distinguish two kinds of traces. A *sending trace* is a sequence of outputs ordered as they are *sent* by their corresponding senders. A *processing trace* is a sequence of inputs ordered as they are *processed* by the services which receive them, that is, they are ordered as they are taken from the input buffer of each addressee service to trigger some of its transitions. Both traces attach some information to explicitly denote the services involved in each operation.

Definition 5. Let $\mathcal{S}$ be a system and let c_1 be the initial configuration of $\mathcal{S}$. In addition, let $(c_1, snd_1, i_1, proc_1, o_1, adr_1, c_2)$, $(c_2, snd_2, i_2, proc_2, o_2, adr_2, c_3), \ldots,$ $(c_k, snd_k, i_k, proc_k, o_k, adr_k, c_{k+1})$ be k consecutive evolutions of $\mathcal{S}$.

Let $a_1 \leq \ldots \leq a_r$ denote all indexes of non-null outputs in the previous sequence, i.e. we have $j \in \{a_1, \ldots, a_r\}$ iff $o_j \neq null$. Then, $[(proc_{a_1}, o_{a_1}, adr_{a_1}), \ldots,$

$(proc_{a_r}, o_{a_r}, adr_{a_r})]$ is a *sending trace* of $\mathcal{S}$. In addition, if there do not exist $snd', i', proc', o', adr', c'$ such that $(c_{k+1}, snd', i', proc', o', adr', c')$ is an evolution of $\mathcal{S}$ then we also say that $[(proc_{a_1}, o_{a_1}, adr_{a_1}), \ldots, (proc_{a_r}, o_{a_r}, adr_{a_r}), \texttt{stop}]$ is a sending trace of $\mathcal{S}$. The set of sending traces of $\mathcal{S}$ is denoted by $\texttt{sendTraces}(\mathcal{S})$.

Let $a_1 \leq \ldots \leq a_r$ denote all indexes of non-null inputs in the previous sequence, i.e. we have $j \in \{a_1, \ldots, a_r\}$ iff $i_j \neq null$. Then, $[(snd_{a_1}, i_{a_1}, proc_{a_1}), \ldots, (snd_{a_r}, i_{a_r}, proc_{a_r})]$ is a *processing trace* of $\mathcal{S}$. In addition, if there do not exist $snd', i', proc', o', adr', c'$ such that $(c_{k+1}, snd', i', proc', o', adr', c')$ is an evolution of $\mathcal{S}$ then we also say that $[(snd_{a_1}, i_{a_1}, proc_{a_1}), \ldots, (snd_{a_r}, i_{a_r}, proc_{a_r}), \texttt{stop}]$ is a processing trace of $\mathcal{S}$. The set of all processing traces of $\mathcal{S}$ is denoted by $\texttt{processTraces}(\mathcal{S})$. □

Next we introduce our formalism to represent choreographies. Contrarily to systems of orchestrations, this formalism focuses on representing the interaction of services as a whole. Thus a single machine, instead of the composition of several machines, is considered.

Definition 6. A *choreography machine* $\mathcal{C}$ is a tuple $\mathcal{C} = (S, M, ID, s_{in}, T)$ where S denotes the set of states, M is the set of messages, ID is the set of service identifiers, $s_{in} \in S$ is the initial state, and T is the set of transitions. A transition $t \in T$ is a tuple (s, m, snd, adr, s') where $s, s' \in S$ are the initial and final states, respectively, $m \in M$ is the message, and $snd, adr \in ID$ are the sender and the addressee of the message, respectively. A transition (s, m, snd, adr, s') is also denoted by $s \xrightarrow{m/(snd \rightarrow adr)} s'$. A *configuration* of $\mathcal{C}$ is any state $s \in S$. □

The next definition presents the operational semantics of choreography machines. Contrarily to systems of services, *null* inputs/outputs are not available, i.e, all communications are effective. Evolutions are activated simply by taking any transition from the current state.

Definition 7. Let $\mathcal{C} = (S, M, ID, s_{in}, T)$ be a choreography machine and $s \in S$ be a configuration of $\mathcal{C}$.

An *evolution* of $\mathcal{C}$ from s is any transition $(s, m, snd, adr, s') \in T$ from state s. The *initial configuration* of $\mathcal{C}$ is s_{in}. □

As we did before for systems of services, next we identify the sequences of messages that can be produced by a choreography machine.

Definition 8. Let c_1 be the initial configuration of a choreography machine $\mathcal{C}$. Let $(c_1, m_1, snd_1, adr_1, c_2)$, $\ldots$, $(c_k, m_k, snd_k, adr_k, c_{k+1})$ be $k \geq 0$ consecutive evolutions of $\mathcal{C}$. We say that $\sigma = [(snd_1, m_1, adr_1), \ldots, (snd_k, m_k, adr_k)]$ is a *trace* of $\mathcal{C}$. In addition, if there do not exist m', snd', adr', c' such that $(c_{k+1}, m', snd', adr', c')$ is an evolution of $\mathcal{C}$ then we also say that $[(snd_1, m_1, adr_1), \ldots, (snd_k, m_k, adr_k), \texttt{stop}]$ is a trace of $\mathcal{C}$. The set of all traces of $\mathcal{C}$ is denoted by $\texttt{traces}(\mathcal{C})$. □

Now we are provided with all the required formal machinery to define our *conformance relations* between systems of orchestrations and choreographies.

Definition 9. Let $\mathcal{S}$ be a system of services and $\mathcal{C}$ be a chorography machine.

We say that $\mathcal{S}$ *conforms to* $\mathcal{C}$ *with respect to sending actions*, denoted by $\mathcal{S}\ \texttt{conf}_s\ \mathcal{C}$, if either $\emptyset \subset \texttt{Complete}(\texttt{sendTraces}(\mathcal{S})) \subseteq \texttt{Complete}(\texttt{traces}(\mathcal{C}))$ or we have $\emptyset = \texttt{Complete}(\texttt{sendTraces}(\mathcal{S})) = \texttt{Complete}(\texttt{traces}(\mathcal{C}))$.

We say that $\mathcal{S}$ *fully conforms to* $\mathcal{C}$ *with respect to sending actions*, denoted by $\mathcal{S}\ \texttt{conf}_s^f\ \mathcal{C}$, if $\texttt{Complete}(\texttt{sendTraces}(\mathcal{S})) = \texttt{Complete}(\texttt{traces}(\mathcal{C}))$.

We say that $\mathcal{S}$ *conforms to* $\mathcal{C}$ *with respect to processing actions*, denoted by $\mathcal{S}\ \texttt{conf}_p\ \mathcal{C}$, if $\emptyset \subset \texttt{Complete}(\texttt{processTraces}(\mathcal{S})) \subseteq \texttt{Complete}(\texttt{traces}(\mathcal{C}))$ or $\emptyset = \texttt{Complete}(\texttt{processTraces}(\mathcal{S})) = \texttt{Complete}(\texttt{traces}(\mathcal{C}))$.

We say that $\mathcal{S}$ *fully conforms to* $\mathcal{C}$ *with respect to sending actions*, denoted by $\mathcal{S}\ \texttt{conf}_p^f\ \mathcal{C}$, if $\texttt{Complete}(\texttt{processTraces}(\mathcal{S})) = \texttt{Complete}(\texttt{traces}(\mathcal{C}))$.

We say that $\mathcal{S}$ *conforms to* $\mathcal{C}$, denoted by $\mathcal{S}\ \texttt{conf}\ \mathcal{C}$, if $\mathcal{S}\ \texttt{conf}_s\ \mathcal{C}$ and $\mathcal{S}\ \texttt{conf}_p\ \mathcal{C}$.

We say that $\mathcal{S}$ *fully conforms to* $\mathcal{C}$ ($\mathcal{S}\ \texttt{conf}^f\ \mathcal{C}$) if $\mathcal{S}\ \texttt{conf}_s^f\ \mathcal{C}$ and $\mathcal{S}\ \texttt{conf}_p^f\ \mathcal{C}$. □

3 Conclusions and future work

In this paper we have presented a formal framework for defining models of orchestrations and choreographies. We have defined some formal semantic relations allowing to detect whether the behavior described by the orchestration of each involved web service correctly leads to the behavior described by a choreography. The suitability of a service for a given choreography may depend on the activities of the rest of services it will be connected with, which contrasts with previous works [2]. In order to take into account the effect of asynchrony, we have separately considered the moments where messages are sent and the moments where they are actually processed.

References

1. M. Boreale, R. D. Nicola, and R. Pugliese. Trace and testing equivalence on asynchronous processes. *Inf. Comput.*, 172(2):139–164, 2002.
2. M. Bravetti and G. Zavattaro. Contract compliance and choreography conformance in the presence of message queues. In *Proc. of 5th International workshop on Web Services and Formal Methods, WS-FM'08, LNCS (in press)*. Springer, 2008.
3. I. Castellani and M. Hennessy. Testing theories for asynchronous languages. In *FSTTCS*, pages 90–101, 1998.
4. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284, 2008.
5. J. Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.
6. J. Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99, LNCS 1664*, pages 46–65. Springer, 1999.
7. W. M. P. van der Aalst, N. Lohmann, P. Massuthe, C. Stahl, and K. Wolf. From public views to private views - correctness-by-design for services. In *WS-FM*, pages 139–153, 2007.