Interface-based service composition with aggregation

Mila Dalla Preda[1], Maurizio Gabbrielli[1], Claudio Guidi[2], Jacopo Mauro[1], and Fabrizio Montesi[3]

[1] Lab. Focus, Department of Computer Science/INRIA, University of Bologna, Italy.
`dallapre | gabbri | jmauro @unibo.it`
[2] italianaSoftware srl, Imola, Italy
`cguidi@italianasoftware.com`
[3] IT University of Copenhagen, Denmark
`fabr@itu.dk`

Abstract. Service-oriented architectures (SOAs) usually comprehend in-the-middle entities such as proxies or service mediators that compose services abstracting from the order in which they exchange messages. Although widely used, these entities are usually implemented by means of ad-hoc solutions.
In this paper we generalise this composition mechanism by identifying the primitive notion of *aggregation*. We formally define the semantics of aggregation in terms of a process calculus. We also provide a reference implementation for this primitive by extending the Jolie language, thus allowing for the experimentation with real SOA scenarios.

1 Introduction

Service-Oriented Computing (SOC) is a programming paradigm for distributed systems based upon the composition of services, autonomous computational entities which can be dynamically discovered and invoked in order to form complex and loosely coupled systems. Service-oriented systems are called Service-Oriented Architectures (SOAs).

Composition is the key aspect of SOC, and it is usually obtained through programming methodologies that impose specific orders of interactions between services. Examples are orchestration and choreography, where the order of interactions is respectively specified from the point of view of a single service or from that of the whole network. We call this kind of composition *flow-based*, referring to its explicit programming of the interaction flows. However, mechanisms based on constraining a specific order of interactions are not the only possible approaches to composition [12]. In practice, it is often the case that distributed networks are supported by entities such as proxies and service buses, which can act as transparent intermediaries between services. These entities are especially useful for handling the topology of an SOA, linking different networks together, or for enacting some functionality that does not depend on the order of interactions between the bridged services (e.g., logging). We call this kind of composition *flow-transparent*.

Flow-based and flow-transparent compositions are represented by a multitude of tools and specifications. For example, in Web Services, orchestration is usually achieved by using WS-BPEL; choreography is addressed in terms of WS-CDL, YAWL, or BPMN.

On the other hand, many commercial platforms for SOC implement flow-transparent composition through an Enterprise Service Bus (ESB) [6], a middleware that provides an abstraction layer to integrate different services in a single SOA. The consumer services communicate with the ESB which translates incoming messages by using a suitable protocol (e.g. REST , JNI, SOAP, etc.) and then routes their translated version to the correct service. Flow-transparent composition comprehends also all the proxy services used for specific tasks in network architectures, such as caching proxies, reverse proxies, and load balancers. Even though flow-transparent composition is widely used, there is no work, to the best of our knowledge, that studies its basic characteristics at the foundational level of a programming model. In this paper we provide such a study, presenting an interpretation in terms of a process calculus. We identify a basic mechanism called *aggregation* for programming flow-transparent composition. Aggregation defines a proxy entity, called *aggregator*, which composes *aggregated* services in a flow-transparent way. Aggregators can change the topology of an SOA by exposing the *interfaces* (collections of operations and their types) of some aggregated services. They can also implement custom functionalities through the specification of code that, by construction, abstracts from the order in which communications are performed. These enhanced aggregators, called *smart aggregators*, can for instance check the content of a message for authorization credentials and then decide whether it must be forwarded or rejected, or it can store some logging information.

We use our model to formalise some properties that we expect in flow-transparent composition. For example, we show that for some aggregators flow-transparent composition does not interfere with the behaviour of flow-based composition, i.e. the order of communications is always preserved. Moreover, we show that aggregators are transparent to operations and interfaces allowing the design of a system that could be easily maintained and adapted to small but also even structural changes.

We show how our study can be used in practice by presenting a reference implementation that extends Jolie [17], a full-fledged service-oriented programming language for building SOAs which is based on the formal process calculus SOCK [8]. We introduce smart aggregation in Jolie building on its support to interface-based composition and structured data types [9, 15]. Our formal model is based on SOCK, ensuring that the properties that we present are preserved in the implementation. It is worth noting that even though Jolie was originally conceived for orchestrating services, its extension to include flow-transparent composition is rather smooth since it exploits primitive Jolie notions such as sessions and input/output operations. The rest of this article is structured as follows: in Section 2 we present some basic notions. In Section 3 we describe the primitive for aggregation in terms of some simple examples while in section 4 we provide its formalization in SOCK. Section 5 presents the implementation in the Jolie language while Section 6 concludes, discussing some related work and indicating directions for future research.

2 Network model

In this section we describe the basic notions that we need to define the deployment of a network of services and, therefore, to define aggregation. A network consists of some service definitions deployed at some locations and the structure of the connections between them. Our notion of *connection* depends on those of *interfaces* and *communication points*, which we define in the following.

We consider the following disjoint sets: the set Var of variables ranged over by x, y; the set Val of values ranged over by v; the set Loc of locations ranged over by l; the set $\mathcal{O}$ of operation names ranged over by o. Finally, we use the bold notation $\mathbf{k}$ to denote a vector $\langle k_0, k_1, \ldots, k_n \rangle$.

In SOC, an *interface* describes the operations exposed by a service. Here we use a simple definition inspired by the WSDL standard [3].

Definition 1 (Interface). *An interface I is a set of* one-way *(OW) and* request-response *(RR) operations with different names.*

An OW operation describes an invocation that does not wait for a response; it is denoted by $o(\mathbf{x})$, where o is the name of the operation and $\mathbf{x}$ are its arguments. An RR operation, denoted by $o(\mathbf{x})(\mathbf{y})$, describes an invocation that waits for a response, so together with the name o of the operation and its arguments $\mathbf{x}$ here we have also the arguments $\mathbf{y}$ that are received back by the invoker. We assume that an interface I cannot contain two operations with the same name. We write $o \in I$ to indicate that an interface I contains an operation whose name is o, omitting the arguments.

Service aggregation is based on the creation of a service (the *aggregator*) with an interface that incorporates other interfaces of existing services. Therefore we introduce a specific operation for manipulating interfaces. In particular, we introduce *argument extension*, which is captured by the (overloaded) function extend that takes an (OW or RR) operation, a list of arguments names and returns a new operation:

$$\mathit{extend}(o(\mathbf{x}), \mathbf{x}') = o(\mathbf{x}\mathbf{x}') \qquad \mathit{extend}(o(\mathbf{x})(\mathbf{y}), \mathbf{x}') = o(\mathbf{x}\mathbf{x}')(\mathbf{y})$$

The extend function can be defined over interfaces in the natural way:

$$\mathit{extend}(I, \mathbf{x}') = \{\mathit{extend}(o(\mathbf{x}), \mathbf{x}') \mid o(\mathbf{x}) \in I\} \cup \{\mathit{extend}(o(\mathbf{x})(\mathbf{y}), \mathbf{x}') \mid o(\mathbf{x})(\mathbf{y}) \in I\}$$

The deployment of a service $\mathcal{S}$ is defined in terms of its *communication points*.

Definition 2 (Communication point). *A communication point is a pair (I, l), where I is an interface and l is a location.*

We distinguish between *input* and *output communication points*. An input communication point (I, l) defines the operations (those contained in the interface I) that a service exposes at the location l. These are the functionalities that other services can invoke. An output communication point (I, l), on the other hand, specifies the operations (those in I) that a service will invoke on location l. These are the functionalities that the service requires from a given location. Given a service $\mathcal{S}$ we denote with $\mathit{In}(\mathcal{S})$ its input communication points and with $\mathit{Out}(\mathcal{S})$ its output communication points.

In order to define the deployment of a network we need to define how its services are connected. Intuitively a connection between a service $\mathcal{S}$ and a service $\mathcal{S}'$ allows the first to invoke the operations of the second: connections are directed. We call a connection between different services *external connection*.

Definition 3 (External connection). *Given services $\mathcal{S}$ and $\mathcal{S}'$, an external connection is a pair of communication points (out, in) such that $in = (I, l) \in In(\mathcal{S})$, $out = (I', l) \in Out(\mathcal{S}')$ and $I \subseteq I'$.*

Next we enrich the communication capability of a service by introducing the notion of *internal connection*, which consists of a link between an input and an output communication point in the same service. This notion allows the programming of bridge services that can forward messages received on an input communication point to an output communication point (thus another service).

Definition 4 (Internal connection). *Given a service $\mathcal{S}$ an internal connection is a pair (in, out) where $in = (I, l) \in In(\mathcal{S})$, $out = (I', l') \in Out(\mathcal{S})$ and there exists a list $\mathbf{k}$ of names of arguments such that $I = extend(I', \mathbf{k})$.*

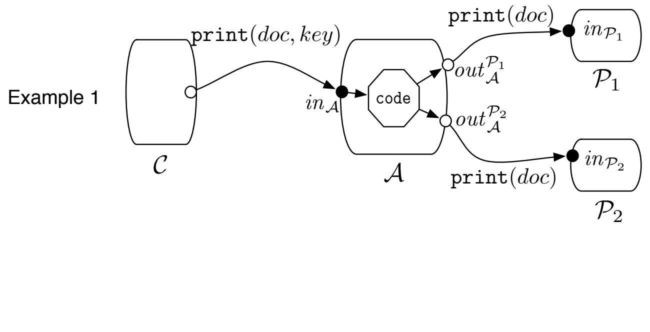
Observe that the interface of $in \in In(\mathcal{S})$ can be an extension of the interface of $out \in Out(\mathcal{S})$ because we want to be able to modify the interfaces of aggregated services.

We say that a service is *directly linked* to another when there exists an external connection from the first to the second. More loosely, a service is *linked* to another, and can therefore invoke it, if there exists a (directed) path consisting of external and internal connections from the first to the second.

3 Some motivating examples

In order to highlight the key concepts and advantages of our primitives for service aggregation we consider, as an example, the case of a printer service exposing its functionalities to an intranet. The intranet is trusted, so no authentication is required for the invokers that want to use the printer. When we extend the use of the printer service functionalities to an untrusted network, say the Internet, we require that the invokers send an authentication token together with the other data required for using the printer. We can easily model this scenario by using a *smart aggregator* service that forwards calls from the Internet to the printer service, which acts as an aggregated service. This aggregator, for each message received from the Internet, checks the authentication token and, if it is correct, it forwards the rest of the message to the printer service. Conversely, the messages coming from the intranet do not need any authentication, hence they are directly sent to the printer. Note that we do not modify the printer service: the aggregator is an external service, and the printer service is not aware of its existence.

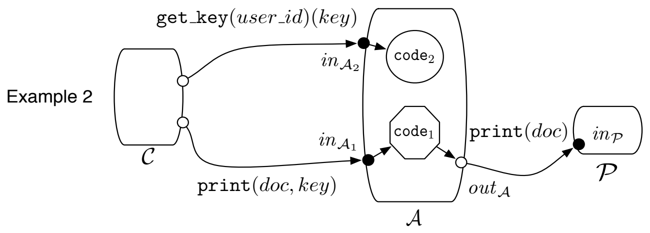
Graphically a scenario where two printers exposing the same interface are aggregated is depicted in the following way:



Now a scenario that constraints the printer at location `loc1` to accept requests only from internet users knowing the key "0000" while the printer at location `loc2` can be used only by users providing the key "1111" can be implemented allowing the aggregator service to execute the following code at every operation invocation.

```
if key == "0000" then
  forward loc1
else if key == "1111" then
  forward loc2
```

It is worth noting that aggregating services could also enhance the behavior of the services since the aggregator could also provide new functionalities on its own. For example, the aggregator service $\mathcal{A}$ in the printing setting could provide a new operation $\texttt{get\_key}(user\_id)(key)$ that, given an identifier of the client, returns the key that could be used to exploit the printing facilities. In this scenario the client should first try to get the key from the service $\mathcal{A}$ through the invocation of the operation `get_key` and then, by using the obtained key, it could proceed by invoking the `print` operation.



The new functionalities added by the aggregating services can be extremely useful in practice. In the previous case, for instance, the operation `get_key` could be exploited for dynamically balancing the workload of the two printers.

4 The formal model

SOCK [8] is a process calculus for Service-Oriented Computing, featuring request-response invocations as a native primitive. It provides the theoretical basis for the implementation of the Jolie language [17]. In this section we extend SOCK with aggregation. We will omit some details that do not influence our presentation. Full definitions can be found in [7].

First we introduce the notion of courier session, which specifies the code that has to be executed by the aggregator before forwarding the message to the final recipient. Next we introduce in the calculus the notion of communication points, which provide an explicit specification of the deployment of services. This allows us to model internal and external connections, and therefore communication among services which are not directly linked (see the terminology introduced at the end of Section 2).

$P ::= \mathbf{0}$	null process	$P;P$	sequence
$\bar{\epsilon}$	output	$P \vert P$	parallel
$\Sigma_i \epsilon_i; P_i$	external choice	$Wait(\mathtt{c}, \mathbf{y})$	wait after solicit
$x := e$	assignment	$Exec(\mathtt{c}, o, \mathbf{y}, P)$	exec after request
$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$	if then else		

output $\bar{\epsilon} ::= o(\mathbf{x})@out$	notification	input $\epsilon ::= o(\mathbf{x})$	reception
$o(\mathbf{x})(\mathbf{y})@out$	solicit	$o(\mathbf{x})(\mathbf{y})\{P\}$	request

Table 1: Process Syntax

4.1 Session

A service in SOCK is a process that can instantiate multiple inner processes equipped with a local state, called sessions. Sessions can send/receive messages and perform computations. Session *behaviours* $P, Q, \ldots$ define the actions to be performed by sessions. A selection of their syntax is reported in Table 1. We denote with $\mathbb{P}$ the set of possible session behaviours. $\mathbf{0}$ is the null process; $\bar{\epsilon}$ is an output, while ϵ is an input; $\Sigma_i \epsilon_i; P_i$ is a standard input-guarded choice; $x := e$ assigns the evaluation of expression e to variable x. We leave the syntax for expressions undefined, assuming that they are first-order expressions including variables and values in Val. $\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$ is an if-then-else choice; $P;P$ and $P|P$ represent, respectively, sequential and parallel composition. $Exec(\mathtt{c}, o, \mathbf{y}, P)$ and $Wait(\mathtt{c}, \mathbf{y})$ are runtime terms that are only used in the semantics. $Exec(\mathtt{c}, o, \mathbf{y}, P)$ represents a server-side running request-response: P is the process computing the answer, o the name of the operation, $\mathbf{y}$ the vector of variables to be used for the answer, and $\mathtt{c}$ the private channel to use to send back the answer. Symmetrically, $Wait(\mathtt{c}, \mathbf{y})$ is the process waiting for the response on client-side: $\mathtt{c}$ is the channel used for receiving the answer and $\mathbf{y}$ the vector of variables to be used for storing the answer. An input ϵ can either be a one-way (OW) $o(\mathbf{x})$ or a request-response (RR) $o(\mathbf{x})(\mathbf{y})\{P\}$, where o is the name of the operation, $\mathbf{x}$ is the vector of variables where to store the received information, and P is the process that has to be executed before sending the information contained in $\mathbf{y}$. An output $\bar{\epsilon}$ can either be the invocation of an OW operation $o(\mathbf{x})@out$ (called notification) or of an RR operation $o(\mathbf{x})(\mathbf{y})@out$ (called solicit-response), where o is the operation name, $\mathbf{x}$ is the vector of variables containing the information to send, $\mathbf{y}$ the vector of variables to store the response, and out specifies the output communication to invoke. An output $o(\mathbf{x})@out$ (or $o(\mathbf{x})(\mathbf{y})@out$) is well formed if o is contained in the interface of the used output communication point, namely $out = (I, l)$ implies $o \in I$.

Let $\sigma : Var \to Val$ be a memory map that associates values to variables and let $\mathfrak{M}$ denote the set of possible memory maps.

Definition 5 (Service session). *A service session T is a pair (P, σ). We denote with $\mathbf{P} = \mathbb{P} \times \mathfrak{M}$ the set of possible service sessions.*

The semantics of a (service) session is specified by a labelled transition system (lts): $(\mathbf{P}, Labels_{\mathbf{P}}, \to_{\mathbf{P}})$. $Labels_{\mathbf{P}}$ is ranged over by α which is defined as follows:

$$\textsc{Reception:}\quad (o(\mathbf{x}),\sigma) \xrightarrow{o(\mathbf{v})}_{\mathbf{P}} (\mathbf{0},\sigma[\mathbf{v}/\mathbf{x}])$$

$$\textsc{Notification:}\quad (o(\mathbf{x})@out,\sigma) \xrightarrow{o(\sigma(\mathbf{x}))@out}_{\mathbf{P}} (\mathbf{0},\sigma)$$

$$\textsc{Solicit:}\quad (o(\mathbf{x})(\mathbf{y})@out,\sigma) \xrightarrow{\uparrow o(\sigma(\mathbf{x}))\mapsto\nu\mathsf{c}@out}_{\mathbf{P}} (\mathit{Wait}(\mathsf{c},\mathbf{y}),\sigma)$$

$$\textsc{SResp:}\quad (\mathit{Wait}(\mathsf{c},\mathbf{y}),\sigma) \xrightarrow{\downarrow\mathsf{c}\mapsto o(\mathbf{v})}_{\mathbf{P}} (\mathbf{0},\sigma[\mathbf{v}/\mathbf{y}])$$

$$\textsc{Request:}\quad (o(\mathbf{x})(\mathbf{y})\{P\},\sigma) \xrightarrow{\uparrow\mathsf{c}\mapsto o(\mathbf{v})}_{\mathbf{P}} (\mathit{Exec}(\mathsf{c},o,\mathbf{y},P),\sigma[\mathbf{v}/\mathbf{x}])$$

$$\textsc{RExe:}\quad \frac{(P,\sigma)\xrightarrow{\alpha}_{\mathbf{P}}(P',\sigma')}{(\mathit{Exec}(\mathsf{c},o,\mathbf{y},P),\sigma)\xrightarrow{\alpha}_{\mathbf{P}}(\mathit{Exec}(\mathsf{c},o,\mathbf{y},P'),\sigma')}$$

$$\textsc{RResp:}\quad (\mathit{Exec}(\mathsf{c},o,\mathbf{y},\mathbf{0}),\sigma) \xrightarrow{\downarrow o(\sigma(\mathbf{y}))@\mathsf{c}}_{\mathbf{P}} (\mathbf{0},\sigma)$$

$$\textsc{Choice:}\ \frac{(\epsilon_i,\sigma)\xrightarrow{\alpha}_{\mathbf{P}}(Q_i,\sigma')}{(\Sigma_i\epsilon_i;P_i,\sigma)\xrightarrow{\alpha}_{\mathbf{P}}(Q_i;P_i,\sigma')}\quad \textsc{Assignment:}\ \frac{[\![e]\!]\sigma=v}{(x:=e,\sigma)\xrightarrow{\tau}_{\mathbf{P}}(\mathbf{0},\sigma[v/x])}$$

Table 2: Session semantics

$$\begin{array}{llll}
\alpha ::= \tau & \text{Silent Action} & \uparrow o(\mathbf{v})\mapsto\nu\mathsf{c}@(I,l) & \text{Solicit}\\
\quad\ o(\mathbf{v})@(I,l) & \text{Notification} & \downarrow\mathsf{c}\mapsto o(\mathbf{v}) & \text{SResponse}\\
\quad\ o(\mathbf{v}) & \text{Reception} & \uparrow\mathsf{c}\mapsto o(\mathbf{v}) & \text{Request}\\
 & & \downarrow o(\mathbf{v})@\mathsf{c} & \text{RResponse}
\end{array}$$

τ is an internal action. $o(\mathbf{v})@(I,l)$ and $o(\mathbf{v})$ model respectively the delivery and the reception of an OW operation. Label $\uparrow o(\mathbf{v})\mapsto\nu\mathsf{c}@(I,l)$ models the invocation of an RR operation to the output communication point (I,l), where $\nu\mathsf{c}$ denotes the new private channel c created for receiving the response later, while label $\uparrow\mathsf{c}\mapsto o(\mathbf{v})$ models the reception of an RR operation on the private channel c. RR invocations are closed by labels $\downarrow\mathsf{c}\mapsto o(\mathbf{v})$ and $\downarrow o(\mathbf{v})@\mathsf{c}$, which denote respectively the reception and the delivery of the response. The transition relation $\rightarrow_{\mathbf{P}}$ is the least relation that satisfies the rules in Table 2 (we report only a selection) and that is closed up to structural equivalence $\equiv$ (namely the least congruence relation satisfying the axioms $P|Q \equiv Q|P$; $P|(Q|R) \equiv (P|Q)|R$; $P|\mathbf{0} \equiv P$; $\mathbf{0};P \equiv P$). We briefly describe the rules in Table 2. Rules Reception and Notification model the reception and deliver of the one-way operation $o(\mathbf{x})$. Rule Solicit says that when a service sends a RR operation $o(\mathbf{x})(\mathbf{y})$ it establishes a fresh channel c on which it then waits for the answer. Once the answer is received the results are stored in variables $\mathbf{y}$ as described by rule SResp. Rule Request models the reception of a request for the RR operation $o(\mathbf{x})(\mathbf{y})\{P\}$ on channel c: the received values are stored in variables $\mathbf{x}$ and then process P is executed. The execution of process P is modeled by rule RExe. Once process P terminates the values contained in variables $\mathbf{y}$ are sent back to the invoking service on channel c, as modeled by rule RResp. In Assignment, $[\![e]\!]\sigma$ denotes the evaluation of expression e on σ. The rule Choice is standard.

4.2 Services

We define now the semantics of a service, building on that of a service session. A service is responsible for the creation and management of its sessions that, like threads in processes, are the entities actually implementing the functionalities required by the invokers. We introduce aggregation at the service level with the primitive $\texttt{agg}(\mathit{List})$. This primitive specifies the internal connections of the aggregator service and the courier process. A courier $\mathfrak{C}$ has the syntax:

$$\mathfrak{C} ::= o(\mathbf{xz}) \rightsquigarrow \hat{P} \;\mid\; o(\mathbf{xz})(\mathbf{y}) \rightsquigarrow \hat{P}$$

where $o(\mathbf{xz})$ and $o(\mathbf{xz})(\mathbf{y})$ are the input operations that should be forwarded and $\hat{P}$ is the process to be executed. The courier process $\hat{P}$ differs from a standard session process P in the fact that it cannot receive inputs, meaning that the term $\Sigma_i \epsilon_i; P_i$ can not appear in $\hat{P}$, and in the fact that it can contain the new term $\texttt{forward(}\mathit{out}\texttt{)}$ that forwards the message that has activated the courier session to the output communication point out. We denote by $\mathbb{C}$ the set of possible couriers ranged over by $\mathfrak{C}$, and by Δ the parallel composition of couriers, that is: $\Delta = \mathfrak{C} \mid \mathfrak{C}|\Delta$. We assume that two couriers in a Δ never start with a same operation o. We write $\mathfrak{C} \in \Delta$ for saying that $\mathfrak{C}$ is in Δ. The idea is that when an input $o(\mathbf{xz})$ (resp. $o(\mathbf{xz})(\mathbf{y})$) arrives to the service the corresponding courier $o(\mathbf{xz}) \rightsquigarrow \hat{P}$ (resp. $o(\mathbf{xz})(\mathbf{y}) \rightsquigarrow \hat{P}$) is considered and a new session that we call *courier session* is created. The process of this courier session is obtained by replacing every occurrence of the term $\texttt{forward(}\mathit{out}\texttt{)}$ in $\hat{P}$ by the term $o(\mathbf{x})@\mathit{out}$ (resp. $o(\mathbf{x})(\mathbf{y})@\mathit{out}$). We denote this substitution by $\hat{P}[o(\mathbf{x})]$ when the input message is an OW operation, and by $\hat{P}[o(\mathbf{x})(\mathbf{y})]$ when the input is an RR operation. Thus, when a service receives an input $o(\mathbf{v})$ that matches the courier $o(\mathbf{xz}) \rightsquigarrow \hat{P}$ the service creates a courier session $(\hat{P}[o(\mathbf{x})], \sigma_\perp[\mathbf{v}/\mathbf{xz}])$ where $\sigma_\perp$ denotes a fresh memory map. If instead the input $o(\mathbf{v})(\mathbf{y})$ that matches the courier $o(\mathbf{xz})(\mathbf{y}) \rightsquigarrow \hat{P}$ is received, the service creates a courier session $(\mathit{Exec}(\mathrm{c}, o, \mathbf{y}, \hat{P}[o(\mathbf{x})(\mathbf{y})]), \sigma_\perp[\mathbf{v}/\mathbf{xz}])$ where c is the channel to be used to send the reply.

Observe that the input operation that activates a courier session and the output operation performed by the forward term are related by the extension function, indeed $o(\mathbf{xz}) = \mathit{extend}(o(\mathbf{x}), \mathbf{z})$ and $o(\mathbf{xz})(\mathbf{y}) = \mathit{extend}(o(\mathbf{x})(\mathbf{y}), \mathbf{z})$. This models the fact that the newly created session executes the process $\hat{P}$ that consumes part of the input (namely $\mathbf{z}$) and then forwards the remaining information [4].

Note also that the term courier session just indicates a session that is created from a courier process $\hat{P}$, once such a session has been created there is no difference with a standard session.

In this paper we abstract from how a service can route an incoming message to the right internal running session, since it is an orthogonal aspect to our presentation. The interested reader may consult [16]. Here we simply assume that messages are delivered to the right session. The sessions in execution at a given instant of time are specified by the execution environment $\mathcal{E}$. We will denote by ε the empty execution environment and by $T_1| \ldots |T_n$ the environment having $T_1, \ldots, T_n$ ($n \geq 1$) as session. Operator $|$ is commutative. We can now define the primitive for aggregation presented in Section 3. The syntax is $\texttt{agg}(\mathbb{I})$ where $\mathbb{I}$ is a list of tuples of the form $\langle \mathit{in}, \{\mathit{out}_1, \ldots, \mathit{out}_n\}, \Delta\rangle$ and:

[4] In the actual implementation the output communication point in a forward primitive can be omitted if it can be unambiguosly determined by looking at the deployment information of the service.

$$\textsc{Start}: \quad \frac{(P, \sigma_\perp) \xrightarrow{\alpha}_{\mathbf{P}} (P', \sigma)}{\langle \mathit{In}, \mathit{Out} \rangle\, P \oplus \mathsf{agg}(\mathbb{I})[\![\mathcal{E}]\!] \xrightarrow{\alpha}_{\mathbf{S}} \langle \mathit{In}, \mathit{Out} \rangle\, P \oplus \mathsf{agg}(\mathbb{I})[\![\mathcal{E} | (P', \sigma')]\!]}$$

$$\textsc{Cour}_1: \quad \frac{\alpha = o(\mathbf{v}) \quad |\mathbf{v}| = |\mathbf{xz}| \quad \langle (\mathit{extend}(I, \mathbf{z}), l), \{\mathit{out}_1, \ldots, \mathit{out}_n\}, \Delta \rangle \in \mathbb{I} \quad o \in I \quad o(\mathbf{xz}) \rightsquigarrow \hat{P} \in \Delta}{\langle \mathit{In}, \mathit{Out} \rangle P \oplus \mathsf{agg}(\mathbb{I})[\![\mathcal{E}]\!] \xrightarrow{\alpha}_{\mathbf{S}} \langle \mathit{In}, \mathit{Out} \rangle P \oplus \mathsf{agg}(\mathbb{I})[\![\mathcal{E} | (\hat{P}[o(\mathbf{x})], \sigma_\perp[\mathbf{v}/\mathbf{xz}])]\!]}$$

$$\textsc{Cour}_2: \quad \frac{\alpha = \uparrow \mathtt{c} \mapsto o(\mathbf{v}) \quad |\mathbf{v}| = |\mathbf{xz}| \quad \langle (\mathit{extend}(I, \mathbf{z}), l), \{\mathit{out}_1, \ldots, \mathit{out}_n\}, \Delta \rangle \in \mathbb{I} \quad o \in I \quad o(\mathbf{xz})(\mathbf{y}) \rightsquigarrow \hat{P} \in \Delta}{\langle \mathit{In}, \mathit{Out} \rangle P \oplus \mathsf{agg}(\mathbb{I})[\![\mathcal{E}]\!] \xrightarrow{\alpha}_{\mathbf{S}} \langle \mathit{In}, \mathit{Out} \rangle P \oplus \mathsf{agg}(\mathbb{I})[\![\mathcal{E} | (\mathit{Exec}(\mathtt{c}, o, \mathbf{y}, \hat{P}[o(\mathbf{x})(\mathbf{y})]), \sigma_\perp[\mathbf{v}/\mathbf{xz}])]\!]}$$

$$\textsc{Exe}: \quad \frac{(Q, \sigma) \xrightarrow{\alpha}_{\mathbf{P}} (Q', \sigma')}{\langle \mathit{In}, \mathit{Out} \rangle P \oplus \mathsf{agg}(\mathbb{I})[\![\mathcal{E} | (Q, \sigma)]\!] \xrightarrow{\alpha}_{\mathbf{S}} \langle \mathit{In}, \mathit{Out} \rangle P \oplus \mathsf{agg}(\mathbb{I})[\![\mathcal{E} | (Q', \sigma')]\!]}$$

Table 3: Service semantics

- $\mathit{in} = (\mathit{extend}(I, \mathbf{x}), l)$ is an input communication point, where $\mathbf{x}$ denotes the array of additional arguments that the incoming messages of interface I should provide;
- $\{\mathit{out}_1, \ldots, \mathit{out}_n\} = \{(I, l_1), \ldots, (I, l_n)\}$ is a nonempty set of output communication points, sharing the same interface;
- Δ denotes the courier behaviour related to the operations of I, such that for every `forward(`out`)` contained in the courier processes in Δ we have that $\mathit{out} \in \{\mathit{out}_1, \ldots, \mathit{out}_n\}$.

We can finally formally define a service. We use $\mathcal{S}$ to denote a service and $\mathbf{S}$ to denote the set of all possible services.

Definition 6 (Service). *A service $\mathcal{S}$ is defined as:*

$$\mathcal{S} ::= \langle \mathit{In}, \mathit{Out} \rangle\, P \oplus \mathit{agg}(\mathbb{I})[\![\mathcal{E}]\!]$$

where In and Out are the set of input and output communication points of the service; P specifies the behaviour of the service sessions; $\mathsf{agg}(\mathbb{I})$ specifies the aggregating behaviour of the service; $\mathcal{E}$ is the environment of the executing sessions.

Observe that the internal connections of a service engine is specified by $\mathbb{I}$. For every element $\langle \mathit{in}, \{\mathit{out}_1, \ldots, \mathit{out}_n\}, \Delta \rangle$ in the list $\mathbb{I}$ we have a corresponding set of internal connections $\{(\mathit{in}, \mathit{out}_i) \,|\, 1 \leq i \leq n\}$. Thus, as expected, we have an internal connection every time that we perform aggregation in a service.

A service is *well-formed* if for every input communication point $\mathit{in} = (I, l)$ in In we have that for every operation $o \in I$ exactly one of the following holds:

- P can receive in input the operation o;
- o is aggregated, namely there exists at least one tuple $\langle \mathit{in}, \{\mathit{out}_1, \ldots, \mathit{out}_n\}, \Delta \rangle \in \mathbb{I}$ such that $\mathit{in} = (I, l) \in \mathit{In}$, $\{\mathit{out}_1, \ldots, \mathit{out}_n\} \subseteq \mathit{Out}$, and $o \in I$.

This means that every operation declared by an input communication point of a service engine is either implemented by the service itself or aggregated.

From the above definition we can observe that a service consists of two main components: P and $\mathsf{agg}(\mathbb{I})$. The first one specifies the behaviour of its internally implemented sessions, while the second one specifies which interfaces the service aggregates

$$\textsc{Not/Rec}: \frac{\mathcal{S}_1 \xrightarrow{o(\mathbf{v})@(I,l)}_{\mathbf{S}} \mathcal{S}_1' \quad \mathcal{S}_2 \xrightarrow{o(\mathbf{v})}_{\mathbf{S}} \mathcal{S}_2 \quad (I,l) \in Out(\mathcal{S}_1) \quad (I,l) \in In(\mathcal{S}_2) \quad o \in I}{\mathcal{S}_1 | \mathcal{S}_2 \xrightarrow{\mu(o(\mathbf{v}))}_{\mathbf{N}} \mathcal{S}_1' | \mathcal{S}_2'}$$

$$\textsc{Sol/Req}: \frac{\mathcal{S}_1 \xrightarrow{\uparrow o(\mathbf{v}) \mapsto \nu \mathtt{c} @(I,l)}_{\mathbf{S}} \mathcal{S}_1' \quad \mathcal{S}_2 \xrightarrow{\uparrow \mathtt{c} \mapsto o(\mathbf{v})}_{\mathbf{S}} \mathcal{S}_2' \quad (I,l) \in Out(\mathcal{S}_1) \quad (I,l) \in In(\mathcal{S}_2) \quad o \in I}{\mathcal{S}_1 | \mathcal{S}_2 \xrightarrow{\mu(\uparrow o(\mathbf{v}))}_{\mathbf{N}} \mathcal{S}_1' | \mathcal{S}_2'}$$

$$\textsc{Res}: \frac{\mathcal{S}_1 \xrightarrow{\downarrow \mathtt{c} \mapsto o(\mathbf{v})}_{\mathbf{S}} \mathcal{S}_1' \quad \mathcal{S}_2 \xrightarrow{\downarrow o(\mathbf{v}) @ \mathtt{c}}_{\mathbf{S}} \mathcal{S}_2'}{\mathcal{S}_1 | \mathcal{S}_2 \xrightarrow{\mu(\downarrow o(\mathbf{v}))}_{\mathbf{N}} \mathcal{S}_1' | \mathcal{S}_2'} \qquad \textsc{S-Exe}: \frac{\mathcal{S} \xrightarrow{\tau} \mathcal{S}'}{\mathcal{S} | \mathsf{N} \xrightarrow{\tau}_{\mathbf{N}} \mathcal{S}' | \mathsf{N}}$$

Table 4: Network semantics

and how the service manipulates the incoming messages before forwarding them to the aggregated communication points. Observe that a simple form of aggregation where messages are only forwarded, as in Example 2 of Section 3, can be seen as a special case of the more general notion of smart aggregation where messages are elaborated by the courier process. In fact, in the first case the courier is only composed by the forward primitive ($\hat{P} = \texttt{forward}(out)$). Hence, without loss of generality, we can assume that a courier is defined for every aggregated operation.

The semantics of the service engine is specified by an LTS $(\mathbf{S}, \mathit{Labels}_{\mathbf{P}}, \rightarrow_{\mathbf{S}})$ where $\rightarrow_{\mathbf{S}} \subseteq \mathbf{S} \times \mathit{Labels}_{\mathbf{P}} \times \mathbf{S}$ is the least relation that satisfies the rules in Table 3, where $\sigma_{\perp}$ denotes a fresh memory map, and its main features are the following. Rules Start, Cour$_1$, and Cour$_2$ model the creation of sessions and courier sessions. Rule Start is standard: when an operation implemented by the service is invoked a new session is created that will handle the request. Rules Cour$_1$ and Cour$_2$ are structurally similar: they create a session to handle the arrival of an aggregated operation. These sessions run a courier code where all the forward primitives are replaced by an output primitive. The last rule, Exe, models the execution of an existing session.

4.3 Network

Definition 7 (Network). *A network* N *is a parallel composition of service engines:* $\mathsf{N} ::= \mathcal{S} \mid \mathsf{N}|\mathcal{S}$.

As argued above, the different services in a network can communicate when they are connected through their input and output communication points. Table. 4 reports the transition rules for a network of services. The first three rules model the communication between services in a network, while the rule S-Exe models the internal evolution of a service in a network, namely the execution of service actions that do not involve input/output operations. Rule Not/Rec models the one-way communication between two services, while request-response communications are modeled through the two rules Sol/Req and Res which represents, respectively, the delivery and reception of an RR operation and the delivery and reception of the answer to an RR operation. The communication rules describe both direct communication between directly connected services, and aggregated communication between services connected by a sequence of external and internal connections.

4.4 Properties

We are now going to show the adequacy of our model by formalizing some proprieties that the flow-transparent mechanism of aggregation preserves, namely flow-based neutrality, operation transparency, and interface transparency. In the following $\langle \beta_1, \ldots, \beta_n \rangle$ ($\langle \beta_1, \ldots, \rangle$) denotes a finite (infinite) trace. The set $[\![\mathsf{N}]\!]$ of maximal finite and infinite traces of a network N are defined as follows:

$$[\![\mathsf{N}_1]\!] = \{\langle \beta_1, \ldots, \beta_n \rangle | \exists \mathsf{N}_2, \ldots, \mathsf{N}_{n+1}\ \forall i\ \mathsf{N}_i \xrightarrow{\beta_i}_{\mathbf{N}} \mathsf{N}_{i+1}\ \wedge \text{ for any } \beta\ \mathsf{N}_{n+1} \not\xrightarrow{\beta}_{\mathbf{N}}\} \cup$$
$$\{\langle \beta_1, \beta_2, \ldots \rangle | \forall i > 1\ \exists \mathsf{N}_i\ \mathsf{N}_{i-1} \xrightarrow{\beta_{i-1}}_{\mathbf{N}} \mathsf{N}_i\}$$

The flow-based neutrality propriety states that the behaviour of a system of services does not change when the messages are rerouted through an aggregator. This is guaranteed by the fact that a simple aggregator, i.e. an aggregator that only forwards the messages, does not alter the flow between the invoker and the callee as stated by the following.

Proposition 1 (Flow-based neutrality). *If $\mathcal{S}_1$ is a service having output communication point (I, l), $\mathcal{S}_2$ is a service having input communication point (I, l), $\mathcal{A}$ aggregates the interface I of $\mathcal{S}_2$, and $\mathcal{S}_1'$ is the service obtained from $\mathcal{S}_1$ by replacing all the locations l with the locations of the $\mathcal{A}$ aggregator, then $\langle \beta_1, \beta_2, \ldots \rangle \in [\![\mathcal{S}_1 | \mathcal{S}_2]\!]$ iff $\langle \beta_1', \beta_2', \ldots \rangle \in [\![\mathcal{S}_1' | \mathcal{A} | \mathcal{S}_2]\!]$ where $\beta_i' = \beta_i, \beta_i$ if β_i is a label involving an operation in I, $\beta_i' = \beta_i$ otherwise. Analogously for the finite traces.*

The property of operation transparency states that the forwarding of the messages to aggregated services does not depend on the names of the aggregated functionalities. This property is guaranteed by the `forward(`*out*`)` construct. In fact, by definition, this construct does not depend on the name of the single operations in the aggregated interface, but it redirects all the operations in the interface to the corresponding output communication point.

Interface transparency means that it is possible to reuse aggregators definitions whenever the interfaces of the aggregated services are modified. Since addition or deletion of operations to an interface can be seen as merge or partition of interfaces, the interface transparency property is guaranteed by the fact that it is possible to merge two or more aggregators providing different functionalities into one aggregator without modifying the courier code. In the following, we denote with $A_{I,\mathcal{S}}$ the aggregator that aggregates the interface I of the service $\mathcal{S}$.

Proposition 2 (Interface transparency). *Assume that I_1, I_2 are interfaces, $\mathcal{C}$, $\mathcal{S}$ are services and $\mathcal{S}$ has input communication point $(I_1 \cup I_2, l)$. Then $[\![\mathcal{S} | \mathcal{C} | \mathcal{A}_{I_1,\mathcal{S}} | \mathcal{A}_{I_2,\mathcal{S}}]\!] = [\![\mathcal{S} | \mathcal{C} | \mathcal{A}_{I_1 \cup I_2,\mathcal{S}}]\!]$.*

We argue that these three properties are the basic ones that a language allowing flow-transparent composition should observe. Indeed the programmer thanks to the flow-based neutrality could reuse existing orchestrators, and thanks to operation and interface transparency can forget about low level, repetitive and usually error prone details. The combination of these three proprieties allows the development of a modular system that can be easily and quickly modified to accommodate the need of a fast changing environment.

5 Implementation in Jolie

In this section we substantiate our approach by showing the use of the new aggregation primitives that have been included in the Jolie language. Here we will just outline the use of this new primitives for the implementation of the examples introduced in Section 3, for a detail description please see [7] instead.

In order to define the aggregator of Example 1 we need to extend the interface of the printer by adding a new argument. The operation that allows us to extend every OW operation of an interface with an additional argument of type *KeyType* can be defined by using the (new) keyword `interface extender` as follows:

```
interface extender AuthIntExtender { OneWay: *(KeyType) }
```

Exploiting this new construct we can now define the new input communication point of the aggregator in the following way

```
inputPort AggregatorPort {
  Location: "socket://localhost:8000"
  Protocol: soap
  Aggregates: { Printer1Port, Printer2Port } with AuthIntExtender
}
```

Here the keyword `Aggregates` that we introduce allows us to specify which services are aggregated. This is obtained by declaring that the output ports *Printer1Port* and *Printer2Port* (defining the output communication points to invoke the printer services) are aggregated in the input port *AggregatorPort* that defines the input communication point of the aggregator service. The input port specifies that messages are accepted on the port 8000 using the socket mechanism and the SOAP protocol.

The primitive `Aggregates` is used here to aggregate the two printers by using the extended interface obtained through the *AuthIntExtender* operator.

To complete the definition of the aggregator we just need to specify the courier session code. If $I_{\mathcal{P}}$ is the interface of the printers this can be done using the new keyword `courier` as follows:

```
courier AggregatorPort {
  [ interface  I_P( request ) ] {
    if ( request.key == "0000" ) {
      forward Printer1Port( request )
    } else if ( request.key == "1111" ) {
      forward Printer2Port( request ) } } }
```

The reading of the previous code should be immediate.

Finally, we use Example 2 to show how functionalities can be added to an aggregator service. Here the goal is to have an aggregator that, besides forwarding the messages to the printer, is also able to provide the user with the key needed for accessing the printer service. Such a key can be obtained by invoking the `get_key`$(user\_id)(key)$ operation, which is included in the new interface defined as follows

```
interface AggregatorInterface
  { RequestResponse: get_key(string)(string) }
```

Suppose that the behaviour of the aggregator service that we want to model is the following: whenever it receives the $\texttt{get\_key}(user\_id)(key)$ operation it returns the key "0000" unless the name of the user is $John$, in which case it returns the key "1010". This can be implemented as follows:

```
main {
  get_key( username )( key ) {
    if ( username == "John" ) { key = "1010" }
    else { key = "0000" } } }
```

Now, in order to completely define the aggregator of Example 2, we just need to modify the input port of the aggregator in the following way.

```
inputPort AggregatorPort {
  Location: "socket://localhost:8000"
  Protocol: sodep
  Interfaces: AggregatorInterface
  Aggregates: Printer1Port with AuthIntExtender
}
```

Notice that now the input port has a new field *Interfaces* that specifies the additional operation that the aggregator provides on its own, in addition to those that are aggregated by using the construct *Aggregates*.
The Jolie code encoding the examples of Section 3 can be retrieved at [1].

6 Related work and conclusions

In this work we studied the foundational aspects of flow-transparent composition of services in the context of SOA. We identified a basic mechanism, called aggregation, that allows programmers to join service functionalities in a loosely coupled, interface based, and behavioural transparent way. We formally defined aggregation in terms of a process calculus and we provided a reference implementation in terms of an extension of the Jolie language. Despite the simplicity of the examples that we provided, it should be clear that aggregation can be used to build, in a rather easy way, large applications along the Enterprise Integration Patterns guidelines.

To the best of our knowledge, this work is the first attempt to bring primitives for the aggregation of services at the same level of the language that is used to define the behavior of a service. Indeed, existing approaches provide tools that define aggregation by mixing different existing solutions. Our work goes in the opposite direction: we englobe into a unique language, with a precise semantics, all the features needed to define services and their aggregation. This facilitates the development of correct software, since the aggregating primitive enforces syntactic and semantic checks which allows to easily prove relevant properties, as previously shown. This advantage is particularly relevant when considering practical, commercial tools for services integration and composition: in this context often the Enterprise Application Integration (EAI) framework [20] is used, which is usually composed by a collection of technologies and services. On the market one can find several EAI mature technologies developed by leading IT companies such as IBM, Oracle and Microsoft. Usually these are implemented by enhancing standard middleware products, often using an Enterprise Service Bus (ESB) [6]. Differently from our approach, all these tools need to operate on top of several existing languages and primitives: for instance, it is not usually possible [24] to implement Enterprise Integration Patterns (EIPs) [10] relying solely on BPEL constructs. This complicates the life of programmers and facilitates the introduction of errors.

Our aggregation mechanism could resemble inheritance in object-oriented languages. However, while inheritance allows the reuse of the *code* of methods, in aggregation what is reused is the executing service itself, since the computation for an aggregated operation invocation is delegated to the aggregated service. We see this as a natural difference, given the fact that aggregation operates in a distributed setting and, as such, locality plays an important role. WSDL [3] is a description language for Web Services that features communication ports. WSDL 2.0 features interface inheritance, allowing an interface to be extended with other operations. This recalls our mechanism of extending an interface using the aggregation primitive. However, in WSDL one can not extend the data type of an operation when using interface inheritance, but only add new operations. The literature reports several attempts of using work-flow techniques [5, 19], AIP planning [23, 4], theorem provers [18, 22] to compose service in an automatic way. Usually these approaches are computationally difficult (often these problems are NP-hard), they make a lot of strong assumptions (like the presence of a common ontology to describe the service functionalities) and they do not scale up to larger systems. In our work we focus on a simpler form of composition with less ambitious goals. Our aggregation mechanism is strongly based on interfaces. There exist other models that exploit types for describing service composition, such as those based on session types [11]. These models, however, are mainly *behavioural* since they focus on aspects such as the order of message exchanges used in the composed services. The aggregation mechanism, instead, focuses on the structure of a service-oriented network and set of operations offered by the composed services. Aggregation and typed behavioral composition play two different, complementary, roles and as future work we plan to add behavioral types to our framework. We also plan to introduce a type system for communication points and connections in order to check the absence of "dangling" output communication points. Moreover we believe that flow-transparent composition facilitates the design of a SOA, since some architectural design decisions may be taken rather early, demanding to the implementation of the (code in the) courier sessions some details.

As another line of future work we are investigating the introduction of dynamic aggregation of services. We would like to extend the current form of static aggregation in order to support dynamic changes of the network topology, thus allowing dynamic creation and deletion of communication points and connections. This could be important for the development of adaptable systems. In this context Jolie could represent an advantage for supporting session stickyness, i.e. the support to track session references in aggregators such as load balancers, since Jolie statically defines the structure of session references (correlation sets) along with service interfaces [16]. Finally our aggregate primitive could be included also in other service-oriented languages based on Web Services, such as WS-BPEL [2], or in other models that are used to formalise service-oriented programming such as those in [14, 13, 21].

References

1. JOLIE Examples Files: `http://www.jolie-lang.org/files/esop2012_aggregation/example.zip`.
2. Web Services Business Process Execution Language Version 2.0: `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`.
3. Web Services Description Language (WSDL) Version 2.0: `http://www.w3.org/TR/wsdl20/`.
4. Piergiorgio Bertoli, Marco Pistore, and Paolo Traverso. Automated composition of Web services via planning in asynchronous domains. *Artif. Intell.*, 174(3-4):316–361, 2010.
5. Fabio Casati, Ski Ilnicki, Li jie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and Dynamic Service Composition in *Flow*. In *CAiSE*, pages 13–31, 2000.
6. David A. Chappell. *Enterprise Service Bus - Theory in practice*. O'Reilly, 2004.
7. M Dalla Preda, M Gabbrielli, C Guidi, J Mauro, and F Montesi. A language for (smart) service aggregation: Theory and practice of interface-based service composition. Technical Report UBLCS-2011-11, University of Bologna, 2011. Available at `http://www.informatica.unibo.it/ricerca/ublcs/2011/UBLCS-2011-11`.
8. Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. SOCK: A Calculus for Service Oriented Computing. In *ICSOC*, pages 327–338, 2006.
9. Claudio Guidi and Fabrizio Montesi. Reasoning about a service-oriented programming paradigm. In *YR-SOC*, pages 67–81, 2009.
10. Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
11. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284, 2008.
12. Rania Khalaf and Frank Leymann. On Web Services Aggregation. In *TES*, pages 1–13, 2003.
13. Cosimo Laneve and Gianluigi Zavattaro. Web-pi at work. In *TGC*, pages 182–194, 2005.
14. Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A formal account of WS-BPEL. In *COORDINATION*, pages 199–215, 2008.
15. Fabrizio Montesi. Jolie: a service-oriented programming language. Master's thesis, University of Bologna, Department of Computer Science, 2010.
16. Fabrizio Montesi and Marco Carbone. Programming services with correlation sets. In *ICSOC*, pages 125–141, 2011.
17. Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Composing Services with JOLIE. In *ECOWS*, pages 13–22, 2007.
18. Jinghai Rao, Peep Küngas, and Mihhail Matskin. Logic-based Web Services Composition: From Service Description to Process Model. In *ICWS*, pages 446–453, 2004.
19. Hans Schuster, Dimitrios Georgakopoulos, Andrzej Cichocki, and Donald Baker. Modeling and Composing Service-Based and Reference Process-Based Multi-enterprise Processes. In *CAiSE*, pages 247–263, 2000.
20. Mostafa Hashem Sherif. *Handbook of Enterprise Integration*. Auerbach Publishers, Incorporated, 2009.
21. Hugo Torres Vieira, Luís Caires, and João Costa Seco. The Conversation Calculus: A Model of Service-Oriented Computation. In *ESOP*, pages 269–283, 2008.
22. Richard J. Waldinger. Web Agents Cooperating Deductively. In *FAABS*, pages 250–262, 2000.
23. Dan Wu, Evren Sirin, James A. Hendler, Dana S. Nau, and Bijan Parsia. Automatic Web Services Composition Using SHOP2. In *WWW (Posters)*, 2003.
24. Xin Yuan. Prototype for executable EAI patterns. Master's thesis, University of Stuttgart, 2008.