Formalizing mobility in Service Oriented Computing

Claudio Guidi
Department of Computer Science, Università degli Studi di Bologna,
Mura Anteo Zamboni 7, I-40127 Bologna, Italy.
Email: cguidi@cs.unibo.it

Roberto Lucchi
European Commission, DG Joint Research Centre, Institute for Environment and Sustainability,
Spatial Data Infrastructures Unit,
T.P. 262, I-21020 Ispra (VA), Italy.
Email: roberto.lucchi@jrc.it

***Abstract*— The usual scenario of service oriented systems is characterized by several services offering the same functionalities, by new services that are continuosly deployed and by other ones that are removed. In this context it can be useful to dynamically discover and compose services at run-time. Orchestration languages provide a mean to deal with service composition, while the problem of fulfilling at run-time the information about the involved services is usually referred to as open-endedness. When designing service-based applications both composition and open endedness play a central role. Such issues are strongly related to mobility mechanisms which make it possible to design applications that acquire, during the execution, the information which are necessary to invoke services. In this paper we discuss the forms of mobility for the service oriented computing paradigm. To this end we model a service by means of the notions of interface, location, internal process and internal state, then we formalize a calculus supporting the mobility of each element listed above. We conclude by tracing a comparison between the proposed calculus and the mobility mechanisms supported by the Web Services technology.**

***Index Terms*— Service oriented computing, mobility mechanisms, formal methods, Web services.**

I. INTRODUCTION

Service Oriented Computing is an emerging paradigm where services are platform independent autonomous computational entities that, by means of standard protocols, support interoperability thus allowing to design new and more complex services out of simpler ones. Orchestration languages [1]–[3] provide a mean to program new services whose functionalities are implemented by exploiting existing services. In particular, the workflow is programmed from the perspective of a single endpoint which orchestrates the invocations of all the involved services and collects/elaborates all the corresponding results. Althought the activity is distributed over the system the orchestrating process holds the execution state in a centralized way.

The usual context for service oriented computing is characterized by the fact that new services can appear as well as other ones can disappear during the evolution of the system, and by the fact that a number of services offer the same functionalities. In this scenario it can be useful to select at run-time the specific service to be invoked among the available ones. Moreover, there are other cases where it is not possible to statically know the exact location of a service which is to be invoked. For instance, consider the case of a system where an administrative application updates the software product versions of clients; it could be organized as it follows. Each client is equipped of a *client* service which provides the software update functionality, the administrative application is composed by a *software manager* service and an *update* service. The *software manager* service invokes the *update* one by passing the list of clients which have to be updated, then the *update* service invokes the software update functionality of all the listed *client* services. Since it is realistic to suppose that the set of all clients changes during the evolution of the whole system, the *update* service does not know at design time the locations of the clients, thus it needs to acquire them at run-time and in particular when it is invoked by the *software manager* service. The problem of composing services that are not completely known at design time is usually referred to as *open endedness*.

In order to deal with open endedness the paper lists and discusses the several forms of mobility for the service oriented computing paradigm. In particular we proceed as it follows: i) we define the service by logically classifying the aspects that compose it, ii) we reason on the meaning of supporting the mobility of such aspects, and iii) we present a service-based calculus equipped with mobility mechanisms. In particular, we characterize a service by means of four components: the *location*, the *internal process*, the *interface* and the *internal state*. The location expresses where the service is deployed and then available, the internal process represents the program which implements and permits to supply the service functionalities, the interface represents the access points the service

Research partially funded by EU Integrated Project Sensoria, contract n. 016004.

© 2007 ACADEMY PUBLISHER

can use to interact with other ones and, finally, the internal state represents the information the service internally holds. The definition we propose is not pertaining to a particular technology thus it permits to reason about mobility without referring to a specific technology. We discuss four kinds of mobility: the location mobility, the internal process mobility, the interface mobility and the internal state mobility. Once having discussed each of them we proceed by presenting a service-based calculus equipped with mobility mechanisms supporting all the forms of mobility listed above.

Such a calculus, equipped of an operational semantics, is an extension of a previous work [4], [5] obtained by introducing the notions of service location and of template. Templates define typed message structures which are used to define the expected message types of each access point of the service interface. Finally, we trace a comparison between the mobility forms we propose and the ones supported by the Web service technology which is the most credited proposal for service oriented computing. It emerges that the technology supports only internal state mobility and location mobility. Moreover, a section is dedicated to investigate the request-response interaction pattern mechanism supported by the Web service technology which seems to be weaker than the common interpretation of the request-response interaction pattern behavior.

The paper is structured as it follows. Section II defines a service and reasons about the meaning of the various forms of mobility that could be supported between services. Section III presents the service-based calculus supporting mobility mechanisms and its operational semantics. Section IV compares the mobility mechanisms we propose with the Web services technology. Section V concludes the paper with some final remarks.

II. SERVICES FORMALIZATION AND MOBILITY MECHANISMS

This section is devoted for deducing the basic concepts of services and introducing the mobility mechanisms they deal with.

A. Communication mechanisms

Service oriented computing is a message based paradigm where messages must be seen as structured containers of typed data. Here we start by considering a single data type we name *information* which represents a general information exploited by a SOC application. In the following we will introduce additional data types which will be exploited to support mobility; in particular the *location*, *operation*, *template* and *process* data types. Informally, locations univocally identify the services in the system, operations and templates define the interface of services and, finally, processes represent the internal behavior of services. For the sake of this paper we abstract away from a detailed classification of types even if it is possible to refine types classification by considering other data types (e.g. integer, float, string). As far as the message structure is concerned, for the sake of generality, here we consider a flat structure where messages are seen as arrays of typed data. In the following message structures will be described by introducing the notion of message *template*. A template describes the expected sequence of data types contained within a message.

Let inf be the type denoting the generic information, $\mathcal{T}$, ranged over by $\vec{t}$, be the set of templates defined as arrays of type elements. For example $\vec{t'} = \langle inf, inf, inf \rangle$ represents the structure of a message with three elements whose type is inf. Let Val, ranged over by v, be the set of values on which is defined a total order relation, $Inf \subseteq Val$ be the set of generic information and T be the function that, given $v \in Val$, returns the type of v. Since currently we are considering only the generic information type we define $T(v) = inf$ if $v \in Inf$; the remaining cases where $v \notin Inf$ will be defined in the following where additional types are introduced. We denote with $\widetilde{v} = \langle v_0, v_1, ..., v_n \rangle$ a tuple of values.

Let $\vec{t} = \langle t_1, \ldots, t_n \rangle$ be a template and $\widetilde{v} = \langle v_1, \ldots, v_s \rangle$ be a tuple, we say that $\widetilde{v}$ satisfies $\vec{t}$, denoted as $\vec{t} \vdash \widetilde{v}$, if the following conditions hold:

1) $n = s$,
2) $\forall v_i, T(v_i) = t_i$.

Every message that needs to be communicated between two services has to be exchanged by means of interaction points. Each service indeed exhibits a set of interaction points, called *operations*, that are exploited for sending and receiving requests to or from other services. Each operation is described by a name and an *interaction modality*. According to [6], [7], there are four kinds of peer-to-peer interaction modality divided into two groups:

- Operations which supply a service functionality, *Input operations*:
  - *One-Way*: it is devoted to receive a request message.
  - *Request-Response*: it is devoted to receive a request message which implies a response message to the invoker.
- Operations which request a service functionality, *Output operations*:
  - *Notification*: it is devoted to send a request message.
  - *Solicit-Response*: it is devoted to send a request message which requires a response message.

We call *single message operations* the One-Way and the Notification operations and we call *double message operations* the Request-Response and the Solicit-Response ones. Formally, let $\mathcal{O} \subseteq Val$ be a set of operation names and let Op be the set of operations defined as it follows:

$$\begin{aligned} Op = & \left\{(o, ow, \vec{t}) \mid o \in \mathcal{O}, \vec{t} \in \mathcal{T}\right\} \\ & \cup \left\{(o, n, \vec{t}) \mid o \in \mathcal{O},\ \vec{t} \in \mathcal{T}\right\} \\ & \cup \left\{(o, rr, \vec{t}, \vec{t'}) \mid o \in \mathcal{O},\ \vec{t}, \vec{t'} \in \mathcal{T}\right\} \end{aligned}$$

© 2007 ACADEMY PUBLISHER

$$\cup \left\{ (o, sr, \vec{t}, \vec{t'}) \mid o \in \mathcal{O},\ \vec{t}, \vec{t'} \in \mathcal{T} \right\}$$

an operation is identified by a name (o), an interaction modality (ow, n, rr and sr represent One-Way, Notification, Request-Response and Solicit-Response interaction modalities respectively) and one or two templates ($\vec{t}, \vec{t'}$) depending on the fact that the operation deals with a single message (One-Way and Notification operations) or two messages (Request-Response or Solicit-Response operations). In the former case, $\vec{t}$ represents the template of the exchanged message whereas in the latter one $\vec{t}$ represents the template of the request message and $\vec{t'}$ represents the template of the reply one. In the following we use $o_{\vec{t}}$, $\overline{o}_{\vec{t}}$, $o_{\vec{t},\vec{t'}}$ and $\overline{o}_{\vec{t},\vec{t'}}$ to range over Op where $o_{\vec{t}}$ represents a One-Way operation whose name is o and the joint template is $\vec{t}$, $\overline{o}_{\vec{t}}$ represents a Notification operation whose name is o and the joint template is $\vec{t}$, $o_{\vec{t},\vec{t'}}$ represents a Request-Response operation whose name is o and the joint templates are $\vec{t}$ for the receiving message and $\vec{t'}$ for the sending one and, finally, $\overline{o}_{\vec{t},\vec{t'}}$ represents a Solicit-Response operation whose name is o and the joint templates are $\vec{t}$ for the sending message and $\vec{t'}$ for the receiving one. We say that two operations $o_{\vec{t}}$ and $\overline{o'}_{\vec{t'}}$ are *dual* if $o = o'$ and $\vec{t} = \vec{t'}$. Analogously, we say that two operations $o_{\vec{t},\vec{t'}}$ and $\overline{o'}_{\vec{t''},\vec{t'''}}$ are *dual* if $o = o'$, $\vec{t} = \vec{t''}$ and $\vec{t'} = \vec{t'''}$. Formally we define duality in the following way:

$$o_{\vec{t}} \bowtie \overline{o'}_{\vec{t'}} \Leftrightarrow o = o' \wedge \vec{t} = \vec{t'}$$
$$o_{\vec{t},\vec{t'}} \bowtie \overline{o'}_{\vec{t''},\vec{t'''}} \Leftrightarrow o = o' \wedge \vec{t} = \vec{t''} \wedge \vec{t'} = \vec{t'''}.$$

B. A model for representing services

A service is a computational entity located at a specific unique *location* (e.g. a URI) which has an *internal state* and is able to perform one or more *functionalities*. A functionality can be a computational process which executes an algorithm, a coordinating process which needs to interact with other services or both. A service can receive a message by means of an input operation and it can send a message by means of an output one and expliciting the location of the receiver. In other words, the operation expresses *how* to interact with a service whereas the location specifies *where* the service can be accessed. The set of all the operations exhibited by a service represents the *interface* of the service. Let Loc, ranged over by l, be the set of locations where $Loc \subseteq Val$. Formally a service is defined by the following tuple:

$$Service := (I, \mathcal{M}, P_f, l)$$

where $I \subseteq Op$ is the interface containing all the operations it can use, $\mathcal{M}$ is the internal state of the service we use to represent all the information it manages (e.g. variables, databases), P_f is the internal process which expresses the service functionality encoded by exploiting the formalism f and $l \in Loc$ is the location where the service is deployed. We remark that, in order to be as general as possible, in this section we abstract away from the specific formalism f and the representation of the internal state; in the next section such notions will be represented by a specific model.

C. Mobility mechanisms

In this section we describe the mobility mechanisms. To this end we exploit the service notion of Section 2.1 and we reason about the meaning of supporting the mobility of each element of the service tuple, that is: internal state mobility, location mobility, interface mobility and internal process mobility. To the best of our knowledge, Service Oriented Computing paradigm does not support an implicit form of mobility[1] but, since the interaction mechanism is based on message passing, mobility can be achieved by exchanging service elements by means of service interfaces. This fact significantly affects the designing issues because mobility must be explicitly programmed by system designers.

- ***Internal state mobility:*** The mobility of the internal state is strongly related to the message passing communication mechanism. Indeed, the content of a sent message is part of the information contained in the internal state of the sender that the receiver acquires and stores in its internal state. In other words, a message exchange between two services can be seen as an information mobility from the sender internal state to the receiver one.
- ***Location mobility:*** Location mobility deals with the possibility to receive a location by means of a message exchange and to exploit it to access the service deployed at that location. In this way for instance a service can acquire, at run-time, the exact location of a service whose functionalities are known. This is the case of the *update* service discussed in the Introduction section which knows the functionality of the *clients* but not their locations.
- ***Interface mobility:*** Interface mobility means that a service can acquire at run-time all the infomation about an operation and then to exhibit it in its interface. Namely, from a designing poit of view, interface mobility allows for the separation of the operation programming from the information necessary to perform it (i.e. message templates and operation names). Interface mobility, indeed, allows for the communication of the templates and the operation names as usual information which are exploited for characterizing an input or an output primitive within a service at run-time. Thus, a human designer can program an input or an output operation without specifying its name and/or its templates by considering the fact that they can be acquired dynamically during the execution of the service. On the contrary, if the interface mobility is not supported the input/output operation templates and operation names must be known at the design time.

[1]The Web Services Request-Response pattern raises an interesting issue about an hidden location mobility which will be discussed in section IV-C

© 2007 ACADEMY PUBLISHER

- ***Internal process mobility:*** Service functionalities are expressed by the internal processes of a service. The mobility of this component implies that a process can be communicated within a message exchange and executed by the service which receives it. In this case the receiver can enrich its internal functionalities by executing the received process. It is important to highlight the fact that the receiver must be able to execute the received process by exploiting the specific formalism used for encoding it (the issues related to this aspect are out of the scope of this paper).

III. A SERVICE-BASED LANGUAGE WITH MOBILITY MECHANISMS

This section is devoted to model the mobility mechanisms discussed above. In particular, we proceed as it follows: i) we introduce a calculus for representing services accordingly with the model discussed in the previous section, ii) we formalize all the mobility mechanisms by extending step by step the service-based calculus and we describe how services are affected by them.

A. The service-based language

Here, we present a service-based calculus which extends OL, defined in our previous works, by means of locations and operation templates. Such a language allows us to describe systems where each participant is a service[2] and supplies a means for describing service functionalities. For the sake of clarity, we do not take into account asynchronous communication which has been modeled in our previous work. On the other hand, this is an orthogonal aspect which can be separately analyzed w.r.t. mobility mechanisms. Formally, let $Signal$ be a set of signal names ranged over by s, let Var be the set of variables ranged over by x, y, z, u, k, j, we denote a tuple of variables by means of the symbol $\widetilde{x} = \langle x_0, x_1, ..., x_n \rangle$. Let W be a finite ordered non-empty set of indexes, OL is defined by the following grammar:

$$
\begin{aligned}
P, Q ::= & \ \mathbf{0} \ \mid \ x := e \ \mid \epsilon \mid \overline{\epsilon} \mid \chi ? P : Q \\
& \mid \ P ; P \ \mid \ P \mid P \ \mid \ \textstyle\sum_{i \in W}^{+} \epsilon_i ; P_i \ \mid \ \chi \rightleftharpoons P
\end{aligned}
$$

$$
\begin{aligned}
\epsilon & ::= s \ \mid \ o_{\vec{t}}(\tilde{x}) \ \mid \ o_{\vec{t},\vec{t'}}(\tilde{x}, \tilde{y}, P) \\
\overline{\epsilon} & ::= \bar{s} \mid \bar{o}_{\vec{t}} @ l(\tilde{x}) \mid \bar{o}_{\vec{t},\vec{t'}} @ l(\tilde{x}, \tilde{y})
\end{aligned}
$$

$$
\begin{aligned}
P_S & := (P, \mathcal{S}) \\
E & ::= [P_S]_l \ \mid \ E \parallel E
\end{aligned}
$$

where a service-based system E consists of the parallel composition of services. A service $[P_S]_l$ is a couple of a process P and a state $\mathcal{S}$ identified by a location $l \in Loc$. The variables state of a service is described by a function $\mathcal{S} : Var \rightarrow Val \cup \{\bot\}$ from variables to the set $Val \cup \{\bot\}$ ranged over by w[3]. $\mathcal{S}(x)$ represents the value of variable x in the state $\mathcal{S}$ ($\mathcal{S}(x) = \bot$ means that x is not yet initialized), while $\mathcal{S}[w/x]$ denotes the state $\mathcal{S}$ where x holds value w (we use $\mathcal{S}[\tilde{w}/\tilde{x}]$ when dealing with tuples of variables), formally:

$$
\mathcal{S}[w/x] = \mathcal{S}' \qquad \mathcal{S}'(x') = \begin{cases} w & if \ x' = x \\ \mathcal{S}(x') & otherwise \end{cases}
$$

All the services are executed at different locations, thus they can be composed by using only the parallel operator ($\|$). Within a location, processes can be composed in parallel ($|$), sequence (;) and with two different alternative composition operators. The operator $\sum_{i \in W}^{+} \epsilon_i ; P_i$ expresses a non-deterministic choice restricted to be guarded on inputs. Such a restriction is due to the fact that we are not interested to model internal non-determinism in service processes. The operator $\chi ? P : Q$ is the *if-then-else* process where χ expresses a logic condition on the variables (the syntax and the the satisfaction relation for χ is reported in the Appendix). $\chi \rightleftharpoons P$ is the construct for modelling guarded iterations. $\mathbf{0}$ represents the null process whereas the processes $x := e$ deals with variable assignment. Processes s and $\bar{s}$ represent processes synchronizations on signals which are exploited to coordinate the activities of processes running in parallel. As far as the operations are concerned, the process $o_{\vec{t}}(\tilde{x})$ represents a One-Way operation where o is the name of the operation, $\vec{t}$ is the template of the received message and $\tilde{x}$ is the tuple of variables where the received information will be stored. $o_{\vec{t},\vec{t'}}(\tilde{x}, \tilde{y}, P)$ represents the Request-Response operation where o is the name of the operation, $\vec{t}$ is the template of the received message and $\vec{t'}$ is the template of the sent message. The Request-Response receives a message, stores the received information in $\tilde{x}$, executes the process P and, at the end, sends the information contained in $\tilde{y}$ as a response message to the invoker. $\bar{o}_{\vec{t}} @ l(\tilde{x})$ represents the Notification operation where o is the name of the operation, $\vec{t}$ is the template of the sent message, l is the location of the invoked service and $\tilde{x}$ is the tuple of variables which contain the sent message. Finally, $\bar{o}_{\vec{t},\vec{t'}} @ l(\tilde{x}, \tilde{y})$ represents the Solicit-Response operation, where o is the name of the operation, $\vec{t}$ is the template of the sent message, $\vec{t'}$ is the template of the received message, l is the location of the invoked service. The Solicit-Response sends the message contained within the $\tilde{x}$ tuple, waits for the response and stores the received information in the tuple $\tilde{y}$.

The semantics of OL is defined in terms of a labelled transition system which describes the evolution of a service-based system. We define $\rightarrow \subseteq OL \times Act \times OL$ as the least relation which satisfies the axioms and rules of Tables I, II and III where $Act = \{\bar{s}, s, \bar{o}, o, \bar{o}_{\vec{t}} @ l(\tilde{v}), o_{\vec{t}}(\tilde{v}), \bar{o}^{n}_{\vec{t},\vec{t'}}(\tilde{v}), o^{n}_{\vec{t}}(\tilde{v}), \bar{o}_{\vec{t},\vec{t'}} @ l(\tilde{v}, \tilde{y})(n), o_{\vec{t},\vec{t'}} @ l(\tilde{v}, \tilde{y})(n), \tau\}$ is the set of actions ranged over by γ. Table I deals with the rules which models communication and synchronization mechanisms where

[2]In our previous work we referred to this language as an orchestration language. Usually the term orchestrator means a special service which, in order to supply its functionalities, coordinates other services. Here, we use the term service for denoting both orchestrators and simple services.

[3]We extend the order relation on the set Val to the set $Val \cup \{\bot\}$ by considering $\bot < v, \ \forall v \in Val$

© 2007 ACADEMY PUBLISHER

(IN)
$$(s,\mathcal{S}) \stackrel{s}{\rightarrow} (\mathbf{0},\mathcal{S})$$

(OUT)
$$(\bar{s},\mathcal{S}) \stackrel{\bar{s}}{\rightarrow} (\mathbf{0},\mathcal{S})$$

(NOTIFICATION)
$$\frac{\vec{t} \vdash \mathcal{S}(\tilde{x})}{(\bar{o}_{\vec{t}}@l(\tilde{x}),\mathcal{S}) \stackrel{\bar{o}_{\vec{t}}@l(\mathcal{S}(\tilde{x}))}{\longrightarrow} (\mathbf{0},\mathcal{S})}$$

(ONE-WAY)
$$\frac{\vec{t} \vdash \tilde{v}}{(o_{\vec{t}}(\tilde{x}),\mathcal{S}) \stackrel{o_{\vec{t}}(\tilde{v})}{\rightarrow} (\mathbf{0},\mathcal{S}[\tilde{v}/\tilde{x}])}$$

(SOLICIT)
$$\frac{\vec{t} \vdash \mathcal{S}(\tilde{x})}{(\bar{o}_{\vec{t},\vec{t}'}@l(\tilde{x},\tilde{y}),\mathcal{S}) \stackrel{\bar{o}_{\vec{t},\vec{t}'}@l(\mathcal{S}(\tilde{x}),\tilde{y})(n)}{\longrightarrow} (o^n_{\vec{t},\vec{t}'}(\tilde{y}),\mathcal{S})}$$

(REQUEST)
$$\frac{\vec{t} \vdash \tilde{v}}{(o_{\vec{t},\vec{t}'}(\tilde{x},\tilde{y},P),\mathcal{S}) \stackrel{o_{\vec{t},\vec{t}'}(\tilde{v},\tilde{y})(n)}{\rightarrow} (P;\bar{o}^n_{\vec{t},\vec{t}'}(\tilde{y}),\mathcal{S}[\tilde{v}/\tilde{x}])}$$

(RESPONSE-OUT)
$$\frac{\vec{t}' \vdash \mathcal{S}(\tilde{x})}{(\bar{o}^n_{\vec{t},\vec{t}'}(\tilde{x}),\mathcal{S}) \stackrel{\bar{o}^n_{\vec{t},\vec{t}'}(\mathcal{S}(\tilde{x}))}{\longrightarrow} (\mathbf{0},\mathcal{S})}$$

(RESPONSE-IN)
$$\frac{\vec{t}' \vdash \tilde{v}}{(o^n_{\vec{t},\vec{t}'}(\tilde{x}),\mathcal{S}) \stackrel{o^n_{\vec{t},\vec{t}'}(\tilde{v})}{\rightarrow} (\mathbf{0},\mathcal{S}[\tilde{v}/\tilde{x}])}$$

TABLE I.
COMMUNICATION RULES

we have introduced the processes $o^n_{\vec{t},\vec{t}'}(\tilde{x})$ and $\bar{o}^n_{\vec{t},\vec{t}'}(\tilde{x})$ in order to deal with Request-Response and Solicit-Response operations. Each rule requires that a received or a sent message must satisfy the current operation template in order to be performed by means of the satisfaction relation described in Section II-A. The most interesting axiom is the REQUEST one which describes that a Request-Response operation, when invoked, behaves as the specified process P and, once having completed such a process, performs an output that is consumed by the invoking service. It is worth noting that a fresh label n allows us to couple the sender process with the receiver one which are explicitly joint within rules of Table III. Rules SOLICIT and RESPONSE-IN deal with Solicit-Response behaviour where, initially, a message is sent and then the service, by means of the process $o^n_{\vec{t},\vec{t}'}(\tilde{x})$, waits for the response. Table II deals with the rules over P_S where the behaviour of a process coupled with a state is expressed. Rule ASSIGN deals with variable assignment within the services; $e \hookrightarrow_{\mathcal{S}} w$ means that the evaluation process of the expression e within state $\mathcal{S}$ reduces to w. Rule INT-SYNC deals with internal synchronization over signals and CONGRP with internal structural congruence denoted by $\equiv_P$. PAR-INT and SEQ describe the behaviour of processes composed in parallel and sequentially respectively, whereas CHOICE and ITERATION 1/2 describe the behavior of the non-deterministic choice and the guarded iteration respectively. The former one non-deterministically selects an input guarded process among the ones listed in the choice operator, while the latter ones model iteration behaviour. Finally, IF THEN and ELSE rules express the if-the-else semantics. In Table III the rules at the level of service-based systems are considered. Rule ONE-WAYSYNC deals with the synchronization on a One-Way operation between two services whereas rules REQ-SYNC and RESP-SYNC deal with the request and the response message exchanges between a Solicit-Response operation and a Request-Response one. Rule REQ-SYNC exploits a fresh label n which is generated in order to univocally link the response synchronization defined in rule RESP-SYNC. PAR-EXT deals with external parallel composition and CONGRE is for external structural congruence denoted by $\equiv$. INT-EXT expresses the fact that a service behaves in accordance with its internal processes.

(ASSIGN)
$$\frac{e \hookrightarrow_{\mathcal{S}} v}{(x := e,\mathcal{S}) \stackrel{\tau}{\rightarrow} (\mathbf{0},\mathcal{S}[v/x])}$$

(INT-SYNC)
$$\frac{(P,\mathcal{S}) \stackrel{s}{\rightarrow} (P',\mathcal{S})\ ,\ (Q,\mathcal{S}) \stackrel{\bar{s}}{\rightarrow} (Q',\mathcal{S})}{(P \mid Q,\mathcal{S}) \stackrel{\tau}{\rightarrow} (P' \mid Q',\mathcal{S})}$$

(CONGRP)
$$\frac{P \equiv_P P'\ ,\ (P',\mathcal{S}) \stackrel{\gamma}{\rightarrow} (Q',\mathcal{S}'),\ Q' \equiv_P Q}{(P,\mathcal{S}) \stackrel{\gamma}{\rightarrow} (Q,\mathcal{S}')}$$

(PAR-INT)
$$\frac{(P,\mathcal{S}) \stackrel{\gamma}{\rightarrow} (P',\mathcal{S}')}{(P \mid Q,\mathcal{S}) \stackrel{\gamma}{\rightarrow} (P' \mid Q,\mathcal{S}')}$$

(SEQ)
$$\frac{(P,\mathcal{S}) \stackrel{\gamma}{\rightarrow} (P',\mathcal{S}')}{(P;Q,\mathcal{S}) \stackrel{\gamma}{\rightarrow} (P';Q,\mathcal{S}')}$$

(CHOICE)
$$\frac{(\epsilon_i;P_i,\mathcal{S}) \stackrel{\gamma}{\rightarrow} (P',\mathcal{S}') \quad i \in W}{(\sum^+_{i\in W} \epsilon_i;P_i,\mathcal{S}) \stackrel{\gamma}{\rightarrow} (P',\mathcal{S}')}$$

(ITERATION 1)
$$\frac{\mathcal{S} \vdash \chi}{(\chi \rightleftharpoons P,\mathcal{S}) \stackrel{\tau}{\rightarrow} (P;\chi \rightleftharpoons P,\mathcal{S})}$$

(ITERATION 2)
$$\frac{\mathcal{S} \nvdash \chi}{(\chi \rightleftharpoons P,\mathcal{S}) \stackrel{\tau}{\rightarrow} (\mathbf{0},\mathcal{S})}$$

(IF THEN)
$$\frac{\mathcal{S} \vdash \chi}{(\chi?P:Q,\mathcal{S}) \stackrel{\tau}{\rightarrow} (P,\mathcal{S})}$$

(ELSE)
$$\frac{\mathcal{S} \nvdash \chi}{(\chi?P:Q,\mathcal{S}) \stackrel{\tau}{\rightarrow} (Q,\mathcal{S})}$$

(STRUCTURAL CONGRUENGE OVER P)
$$P \mid \mathbf{0} \equiv_P P \quad \mathbf{0};P \equiv_P P$$
$$(P \mid Q) \equiv_P (Q \mid P) \quad (P \mid Q) \mid R \equiv_P P \mid (Q \mid R)$$

TABLE II.
RULES OVER P_S

Now, we remind the service formalization presented in Section II where a service is represented by the tuple $(I,\mathcal{M},P_f,l)$ and we show how an OL service $[P,\mathcal{S}]_l$ is

© 2007 ACADEMY PUBLISHER

(ONE-WAYSYNC)

$$\frac{[P_S]_l \overset{\bar{o}_{\vec{t}}@l'(\tilde{v})}{\rightarrow} [P'_S]_l \ , \ [Q_S]_{l'} \overset{o'_{\vec{t}'}(\tilde{v})}{\rightarrow} [Q'_S]_{l'}, \bar{o}_{\vec{t}} \bowtie o'_{\vec{t}'}}{[P_S]_l \parallel [Q_S]_{l'} \xrightarrow{\tau} [P'_S]_l \parallel [Q'_S]_{l'}}$$

(REQ-SYNC)

$$\frac{[P_S]_l \xrightarrow{\sigma} [P'_S]_l \ , \ [Q_S]_{l'} \xrightarrow{\sigma'} [Q'_S]_{l'}}{[P_S]_l \parallel [Q_S]_{l'} \xrightarrow{\tau} [P'_S]_l \parallel [Q'_S]_{l'}} \begin{cases} n \ fresh \\ \sigma = \bar{o}_{\vec{t},\vec{t}'}@l'(\tilde{v},\tilde{y})(n) \\ \sigma' = o'_{\vec{t}'',\vec{t}'''}(\tilde{v},\tilde{y})(n) \\ \bar{o}_{\vec{t},\vec{t}'} \bowtie o'_{\vec{t}'',\vec{t}'''} \end{cases}$$

(RESP-SYNC)

$$\frac{[P_S]_l \overset{\bar{o}^n_{\vec{t},\vec{t}'}(\tilde{v})}{\rightarrow} [P'_S]_l \ , \ [Q_S]_{l'} \overset{o^n_{\vec{t},\vec{t}'}(\tilde{v})}{\rightarrow} [Q'_S]_{l'}}{[P_S]_l \parallel [Q_S]_{l'} \xrightarrow{\tau} [P'_S]_l \parallel [Q'_S]_{l'}}$$

(CONGRE)

$$\frac{E_1 \equiv E'_1 \ , \ E'_1 \xrightarrow{\gamma} E'_2, \ E'_2 \equiv E_2}{E_1 \xrightarrow{\gamma} E_2}$$

(PAR-EXT)

$$\frac{E_1 \xrightarrow{\gamma} E'_1}{E_1 \parallel E_2 \xrightarrow{\gamma} E'_1 \parallel E_2}$$

(INT-EXT)

$$\frac{P_S \xrightarrow{\gamma} P'_S}{[P_S]_l \xrightarrow{\gamma} \left[P'_S\right]_l}$$

(STRUCTURAL CONGRUENCE OVER E)

$$\frac{P \equiv_P Q}{[P,\mathcal{S}]_l \equiv [Q,\mathcal{S}]_l}$$

$$E_1 \parallel E_2 \equiv E_2 \parallel E_1 \qquad E_1 \parallel (E_2 \parallel E_3) \equiv (E_1 \parallel E_2) \parallel E_3$$

TABLE III.
RULES OVER E

related to it:

- $\mathcal{M}$ is modeled by $\mathcal{S}$.
- l represents the location within both the service model and the OL language.
- P_f is represented by a process P in OL where the formalism f corresponds to OL.
- I represents the interface of a service and it is not explicitly modeled in OL but it can be extracted from the process P. Indeed, by considering a service $[P,\mathcal{S}]_l$, its interface I is defined by the function $\Theta(P)$ where Θ is inductively defined by the following rules:

1. $\Theta(\mathbf{0}) = \phi$
2. $\Theta(x := e) = \phi$
3. $\Theta(s) = \phi$
4. $\Theta(\bar{s}) = \phi$
5. $\Theta(\bar{o}_{\vec{t}}@l(\tilde{x})) = \{(o, n, \vec{t})\}$
6. $\Theta(\bar{o}_{\vec{t},\vec{t}'}@l(\tilde{x},\tilde{y})) = \{(o, sr, \vec{t}, \vec{t}')\}$
7. $\Theta(o_{\vec{t}}(\tilde{x})) = \{(o, ow, \vec{t})\}$
8. $\Theta(o_{\vec{t},\vec{t}'}(\tilde{x},\tilde{y},P)) = \{(o, rr, \vec{t}, \vec{t}')\} \cup \Theta(P)$
9. $\Theta(o^n_{\vec{t},\vec{t}'}(\tilde{x})) = \phi$
10. $\Theta(\bar{o}^n_{\vec{t},\vec{t}'}(\tilde{x})) = \phi$
11. $\Theta(P;P') = \Theta(P) \cup \Theta(P')$
12. $\Theta(P \mid P') = \Theta(P) \cup \Theta(P')$
13. $\Theta(\sum^{+}_{i\in W} \epsilon_i; P_i) = \bigcup_{i\in W} \Theta(\epsilon_i; P_i)$
14. $\Theta(\chi?P : Q) = \Theta(P) \cup \Theta(Q)$
15. $\Theta(\chi \rightleftharpoons P) = \Theta(P)$

It is worth noting that the interface $\Theta(P)$, during the evolution of a service $[P,\mathcal{S}]_l$, is monotonically reduced dependently on the consumption of P. Indeed, let us consider the following example:

$$[\bar{a}_{\vec{t}}(x), \mathcal{S}[4/x]]_l \parallel [a_{\vec{t}}(y), \mathcal{S}']_{l'} \xrightarrow{\tau} [\mathbf{0}, \mathcal{S}[4/x]]_l \parallel [\mathbf{0}, \mathcal{S}'[4/y]]_{l'}$$

Before the synchronization the interfaces of the two services are $I_l = \{(a, n, \vec{t})\}$ and $I_{l'} = \{(a, ow, \vec{t})\}$ respectively, whereas after the synchronization they are $I_l = \phi$ and $I_{l'=}\phi$.

B. Internal state mobility

As we have noticed in Section II the internal state mobility is strongly related to the message passing communication mechanism. Considering Table I and Table III, such a kind of mobility is expressed by the rules which deal with operation processes. In particular, let us consider rules NOTIFICATION and ONE-WAY of Table I in order to clarify how it works. In the former the internal state information $\tilde{v}$ contained within the variables $\tilde{x}$ are sent by exploiting a message whereas in the latter the received information $\tilde{v}$ are stored into the variables $\tilde{x}$ contained within the internal state of the receiver. Rule ONE-WAYSYNC of Table III couples the two rules by correlating the receiver location to that explicited within the notification process. In this case the message content is represented by the tuple of values $\tilde{v}$. Summarizing, internal state mobility is modeled as an information exchange between the internal state of the sender and the internal state of the receiver. Such a mobility mechanism is the cornerstone of service-based systems and supplies the basic layer on which the other mobility mechanisms can be implemented.

C. Location mobility

So far, OL does not deal with location mobility. Locations, indeed, are statically explicited within the Notification and the Solicit-Response primitives. In order to deal with location mobility we modify the syntax of OL by introducing the possibility to express the location

© 2007 ACADEMY PUBLISHER

as the content of a variable. To this end we add two new primitives for the Notification and the Solicit-Response where z is a variable:

$$P ::= \ldots \mid \bar{o}_{\vec{t}}@z(\tilde{x}) \mid \bar{o}_{\vec{t},\vec{t'}}@z(\tilde{x},\tilde{y}) \mid \ldots$$

These new primitives allow us to dynamically bind the receiver location when performing the Notification and Solicit-Response operations by evaluating the content of variable z. Since locations will be acquired by means of an input operation we introduce a new data type, we call loc, representing the type used for locations. The function used to test the conformance between tuples of values and templates will be enriched by considering that, given $v \in Val$, $T(v) = loc$ if $v \in Loc$ (we assume that the set of values of each type is disjunct with each other).

The semantics follows:

(NOTIFICATION WITH LOCATION MOBILITY)

$$\frac{\vec{t} \vdash \mathcal{S}(\tilde{x}),\ T(\mathcal{S}(z)) = loc}{(\bar{o}_{\vec{t}}@z(\tilde{x}), \mathcal{S}) \stackrel{\bar{o}_{\vec{t}}@\mathcal{S}(z)(\mathcal{S}(\tilde{x}))}{\longrightarrow} (\mathbf{0}, \mathcal{S})}$$

(SOLICIT WITH LOCATION MOBILITY)

$$\frac{\vec{t} \vdash \mathcal{S}(\tilde{x}),\ T(\mathcal{S}(z)) = loc}{(\bar{o}_{\vec{t},\vec{t'}}@z(\tilde{x},\tilde{y}), \mathcal{S}) \stackrel{\bar{o}_{\vec{t},\vec{t'}}@\mathcal{S}(z)(\mathcal{S}(\tilde{x}),\tilde{y})(n)}{\longrightarrow} (o^{n}_{\vec{t},\vec{t'}}(\tilde{y}), \mathcal{S})}$$

Variable z is evaluated when the processes are executed. In that phase we exploit types in order to prevent the execution of bad processes: in the case z does not hold a location value, the primitive is not performed. This mechanism allows us to design a service which does not know *a priori* the locations of the services to be invoked that can be acquired during the execution. In order to clarify such a behaviour let us consider the business scenario example depicted in Fig. 1 where a customer purchases a good invoking a shopping service, the shopping service invokes a bank service for performing the payment and the bank service invokes the customer for sending the invoice. In Fig. 1 we have exploited an informal graphical representation where services are represented by circles, the symbol $@uri$ expresses the fact that the service is available at the location uri, the input operations exhibited by a service are represented by a black line whose name is shown within a rectangle and the arrows represent a message exchange. The shopping service exhibits the One-Way BUY, the Bank service exhibits the One-Way PAY and the Customer service exhibits the One-Way REC.

In the following we formalize such a scenario by supposing that the bank service does not know the location of the customer:

$$\vec{t} = \langle loc \rangle \quad \vec{t'} = \langle inf \rangle$$

$$C ::= [add := uri1; inv := \bot$$
$$;\overline{\mathrm{BUY}}_{\vec{t}}@uri2(add); \mathrm{REC}_{\vec{t'}}(inv), \mathcal{S}_c]_{uri1}$$
$$SH ::= [fwadd := \bot$$
$$;\mathrm{BUY}_{\vec{t}}(fwadd); \overline{\mathrm{PAY}}_{\vec{t}}@uri3(fwadd), \mathcal{S}_s]_{uri2}$$
$$B ::= [z_3 := \bot; invoice := msg$$
$$;\mathrm{PAY}_{\vec{t}}(z_3); \overline{\mathrm{REC}}_{\vec{t'}}@z_3(invoice), \mathcal{S}_b]_{uri3}$$

$$System ::= C \parallel SH \parallel B$$

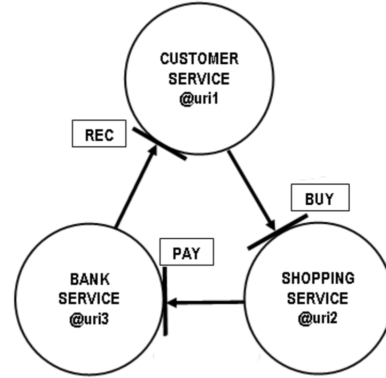

Figure 1. Business scenario example

The shopping service SH located at $uri2$ receives on the One-Way operation BUY the location of the customer C ($uri1$) and stores it within the variable $fwadd$. Moreover, it forwards it to the bank service B (at $uri3$) by exploiting the Notification operation $\overline{\mathrm{PAY}}$. The bank service receives on PAY the customer location and then exploits it for invoking the REC operation of the customer sending the invoice here represented by the value msg. Finally, the customer receives the invoice on REC and stores the message content within the variable inv.

Location mobility introduces a powerful mechanism for designing services in a flexible way. If we consider the Bank service of the example indeed, it exploits the operation $\overline{\mathrm{REC}}_{\vec{t'}}@z_3(invoice)$ in order to be independent from the customer address. The Bank service can send invoices to all the customers which exhibit a One-Way whose name is REC and has a template $\vec{t'}$. On the contrary, if we do not exploit location mobility the Bank service should know the customer address before its execution binding the service to interact to a specific customer. This is the case of the shopping service that, by exploiting the operation $\overline{\mathrm{PAY}}_{\vec{t}}@uri3(fwadd)$, is designed for sending the payment request always to the same Bank service. Furthermore, the example shows that location mobility is built on top of the internal state mobility because the acquired locations are stored within the internal state.

D. Interface mobility

Interface mobility deals with the mobility of all the information related to an operation that is the name of the operation and the templates joint to it. In general, all these information can be acquired dynamically. We model interface mobility in OL by introducing the following primitives where z, u, k and j are variables:

© 2007 ACADEMY PUBLISHER

$$P ::= \ldots \mid \bar{u}_{\vec{k}}@z(\tilde{x}) \mid u_{\vec{k}}(\tilde{x}) \mid \bar{u}_{\vec{k},\vec{j}}@z(\tilde{x},\tilde{y}) \mid u_{\vec{k},\vec{j}}(\tilde{x},\tilde{y},P) \mid \ldots$$

The name of the operations and the templates are evaluated at run-time by reading them from the state. To this end we introduce two new data types op and t which are used to represent the type of operation names and of templates, respectively. Let $v \in Value$, the function T is extended by defining the following cases: i) $T(v) = op$ if $v \in Op$, ii) $T(v) = t$ if $v \in \mathcal{T}$. We will exploit this data types to test that the values stored in the variables are in accordance with the expected data types. The semantics follows:

(NOTIFICATION WITH INTERFACE MOBILITY)

$$\frac{T(\mathcal{S}(k)) = t, T(\mathcal{S}(u)) = op, \overrightarrow{\mathcal{S}(k)} \vdash \mathcal{S}(\tilde{x}), T(\mathcal{S}(z)) = loc}{(\bar{u}_{\vec{k}}@z(\tilde{x}), \mathcal{S}) \xrightarrow{\overline{\mathcal{S}(u)}_{\overrightarrow{\mathcal{S}(k)}}@\mathcal{S}(z)(\mathcal{S}(\tilde{x}))} (\mathbf{0}, \mathcal{S})}$$

(ONE-WAY WITH INTERFACE MOBILITY)

$$\frac{T(\mathcal{S}(k)) = t, T(\mathcal{S}(u)) = op, \overrightarrow{\mathcal{S}(k)} \vdash \mathcal{S}(\tilde{x})}{(u_{\vec{k}}(\tilde{x}), \mathcal{S}) \xrightarrow{\overline{\mathcal{S}(u)}_{\overrightarrow{\mathcal{S}(k)}}(\mathcal{S}(\tilde{x}))} (\mathbf{0}, \mathcal{S})}$$

(SOLICIT WITH INTERFACE MOBILITY)

$$\frac{\begin{array}{c} T(\mathcal{S}(k)) = T(\mathcal{S}(j)) = t, T(\mathcal{S}(u)) = op, \\ \overrightarrow{\mathcal{S}(k)} \vdash \mathcal{S}(\tilde{x}), T(\mathcal{S}(z)) = loc \end{array}}{(\bar{u}_{\vec{k},\vec{j}}@z(\tilde{x},\tilde{y}), \mathcal{S}) \xrightarrow{\alpha} (\mathcal{S}(u)^{n}_{\overrightarrow{\mathcal{S}(k)},\overrightarrow{\mathcal{S}(j)}}(\tilde{y}), \mathcal{S})}$$

$$\alpha = \overline{\mathcal{S}(u)}_{\overrightarrow{\mathcal{S}(k)},\overrightarrow{\mathcal{S}(j)}}@\mathcal{S}(z)(\mathcal{S}(\tilde{x}),\tilde{y})(n)$$

(REQUEST WITH INTERFACE MOBILITY)

$$\frac{T(\mathcal{S}(k)) = T(\mathcal{S}(j)) = t, T(\mathcal{S}(u)) = op, \overrightarrow{\mathcal{S}(k)} \vdash \mathcal{S}(\tilde{x})}{(u_{\vec{k},\vec{j}}(\tilde{x},\tilde{y},P), \mathcal{S}) \xrightarrow{\alpha} (P; \overline{\mathcal{S}(u)}^{n}_{\overrightarrow{\mathcal{S}(k)},\overrightarrow{\mathcal{S}(j)}}(\tilde{y}), \mathcal{S})}$$

$$\alpha = \mathcal{S}(u)_{\overrightarrow{\mathcal{S}(k)},\overrightarrow{\mathcal{S}(j)}}(\mathcal{S}(\tilde{x}),\tilde{y})(n)$$

It is worth noting that the introduction of the interface mobility allows us to distinguish the concept of operation programming from that of the information which characterize it. The former expresses the service capability to perform a One-Way, a Notification, a Request-Response or a Solicit-Response operations represented by the processes $u_{\vec{k}}(\tilde{x})$, $\bar{u}_{\vec{k}}@z(\tilde{x})$ $u_{\vec{k},\vec{j}}(\tilde{x},\tilde{y},P)$ and $\bar{u}_{\vec{k},\vec{j}}@z(\tilde{x},\tilde{y})$ respectively, whereas the latter deals only with the information that are necessary for performing an operation represented by the content of the variables u, k and j.

In this case the interface can change during the evolution of the service, thus we need to modify some rules of the inductive definition of Θ. To this end we first introduce the functions $tN : Val \to Val \cup \{?\}$ and $tT : Val \to Val \cup \{?\}$ for testing if the content of a variable is an operation name or a template respectively. We exploit the symbol ? for expressing the fact that an information related to an operation is unknown. The definition of the functions follows:

$$tN(v) = \begin{cases} v \ if \ v \in \mathcal{O} \\ ? \quad otherwise \end{cases}$$

$$tT(v) = \begin{cases} v \ if \ v \in \mathcal{T} \\ ? \quad otherwise \end{cases}$$

For the sake of brevity we report below only the rules that change w.r.t. the original definition of Θ that are the 5, 6, 7 and 8 ones. It is worth noting that here we extend the domain of Θ by considering also the internal state. This is due to the fact that now the interface depends on the contents of the variables.

5. $\Theta(\bar{u}_{\vec{k}}@z(\tilde{x}), \mathcal{S}) = \{tN(\mathcal{S}(u)), n, tT(\mathcal{S}(k))\}$

6. $\Theta(\bar{u}_{\vec{k},\vec{j}}@z(\tilde{x},\tilde{y}), \mathcal{S}) = \{tN(\mathcal{S}(u)), sr, tT(\mathcal{S}(k), tT(\mathcal{S}(j))\}$
7. $\Theta(u_{\vec{k}}(\tilde{x}), \mathcal{S}) = \{tN(\mathcal{S}(u)), ow, tT(\mathcal{S}(k))\}$
8. $\Theta(u_{\vec{k},\vec{j}}(\tilde{x},\tilde{y},P), \mathcal{S}) = \{tN(\mathcal{S}(u)), rr, tT(\mathcal{S}(k), tT(\mathcal{S}(j))\} \cup \Theta(P)$

Example 3.1: Let us consider the example of Fig. 1 where we suppose that the Bank service does not know *a priori* both the location and the One-Way operation of the customer:

$$\vec{t} = \langle loc, op, t\rangle \quad \vec{t'} = \langle inf\rangle$$

$$\begin{aligned} C ::= [&add := uri1; opN := \text{REC}; opT := \vec{t'}; inv := \bot \\ &; \overline{\text{BUY}}_{\vec{t}}@uri2(add, opN, opT); \text{REC}_{\vec{t'}}(inv), \mathcal{S}_c]_{uri1} \\ SH ::= [&fwadd := \bot; fwopN := \bot; fwopT := \bot; \\ &; \text{BUY}_{\vec{t}}(fwadd, fwopN, fwopT) \\ &; \overline{\text{PAY}}_{\vec{t}}@uri3(fwadd, fwopN, fwopT), \mathcal{S}_s]_{uri2} \\ B ::= [&z_3 := \bot; op := \bot; tp := \bot; invoice = msg \\ &; \text{PAY}_{\vec{t}}(z_3, op, tp); \overline{op}_{\vec{tp}}@z_3(invoice), \mathcal{S}_b]_{uri3} \end{aligned}$$

$$System ::= C \parallel SH \parallel B$$

The customer sends, by means of the variable opN and opT, the operation name (REC) and the operation template ($\vec{t}$) on which it will wait for receiving the invoice. The bank service receives from the shopping service the location, the name of the operation and the template of the operation of the customer and stores them in z_3, op and tp respectively.

The example shows how is possible to design a service (in this case the bank one) with a functionality which deals with an output operation without statically knowing its interface. In general, it is possible to have scenarios where a service partially knows the interface information that is, for example, it knows the name of the operation but it does not know the template or, viceversa, it knows the template but it does not know the name of the operation. In particular, the mobility of the information related

© 2007 ACADEMY PUBLISHER

only to the templates raise some interesting designing issues. A designer that does not know the template of an operation is able to program an input or an output operation but he is not able to predict the structure of the received (or sent) data and, as a consequence, he cannot exactly specify the variables related to the received (sent) data. Let us consider, for example, the output operation of the bank service:

$$\overline{op}_{\vec{tp}}@z_3(invoice)$$

in this case, even if the content of the variable tp is unkwon, there is an implicit knowledge of the template because it can be indirectly extracted by considering the variable $invoice$ which is programmed as the variable that contain the data to send. In general, a full interface mobility cannot be supported without considering a mechanism which allows a designer to formulate some kinds of predictions about the received (sent) data. We can imagine indeed, that the designer could be able to program some kinds of specifications about the variables from which it should be possible to build a sort of dynamic *adaptor* for binding the variables with the received template. The discussion and the formalization of such a kind of machinery is out of the scope of this paper and, at the best of our knowledge, it is an open issue. As a first attempt towards this direction, works on component adaption can be taken into consideration. For example, in [8] Brogi et al. discuss the problem of the adaption between Web Services interfaces where adator specifications are discussed for composing different services with different interfaces.

E. Internal process mobility

In order to deal with internal process mobility we extend the OL language by introducing the following process:

$$P ::= \ldots \mid \texttt{run}(x)$$

$\texttt{run}(x)$ allows us to execute the code contained within the variable x. As previously done for the other kinds of mobility we introduce a new data type representing processes. Let $proc$ be the data type denoting processes, $v \in Val$: $T(v) = proc$ if $proc$ is defined by the term P presented in Section III-A. The semantics of such a primitive is expressed by a new rule that must be added to those presented in Table II:

$$\text{(RUN)}\ \frac{T(\mathcal{S}(x)) = proc}{(\texttt{run}(x), \mathcal{S}) \xrightarrow{\tau} (\mathcal{S}(x), \mathcal{S})}$$

Since the received code can be formed by operation processes, we add a new rule for inductively defining the function Θ which allows us to extract the interface of the service:

$$13.\ \Theta(\texttt{run}(x), \mathcal{S}) = \begin{cases} \Theta(\mathcal{S}(x)) & if\ \mathcal{S}(x) \neq \bot \\ \phi & otherwise \end{cases}$$

Service functionality mobility directly deals with code mobility. In particular it allows us to design services where a specific part of its functionalities are unknown at design time and they are acquired during the execution of the service. In order to clarify this aspect let us consider the example of the shopping service again where we suppose that the customer, that wants to interact with the shopping service, does not know *a priori* the conversation rules to follow. In other words, the customer does not know that it has to exhibit the REC operation in order to receive the invoice from the bank service.

$$\vec{t} = \langle loc, proc\rangle \quad \vec{t'} = \langle inf\rangle \quad \vec{t''} = \langle loc\rangle$$

$$\begin{aligned}
&C ::= [add := uri1; code_1 := \bot \\
&\quad ; \overline{\text{BUY}}_{\vec{t}}@uri2(add, code_1); \texttt{run}(code_1), \mathcal{S}_c]_{uri1} \\
&SH ::= [fwadd := \bot; code_2 :=\text{“}inv := \bot; \text{REC}_{\vec{t'}}(inv)\text{”} \\
&\quad ; \text{BUY}_{\vec{t}}(fwadd, code_2) \\
&\quad ; \overline{\text{PAY}}_{\vec{t''}}@uri3(fwadd), \mathcal{S}_s]_{uri2} \\
&B ::= [z_3 := \bot; invoice = msg \\
&\quad ; \text{PAY}_{\vec{t''}}(z_3); \overline{\text{REC}}_{\vec{t'}}@z_3(invoice), \mathcal{S}_b]_{uri3}
\end{aligned}$$

$$System ::= C \parallel SH \parallel B$$

Here, the customer invokes the operation BUY of the shopping service which is modeled as a Request-Response operation. The customer receives as a response a piece of code and stores it within the variable $code_1$, then it executes it by exploiting the primitive $\texttt{run}(code_1)$. After the execution of the code stored within $code_1$ the system behaves as the example presented in the location mobility section. It is worth noting that the customer receives the input operation REC which enriches at run-time its interface similarly to the case of the interface mobility. Even if the two kind of mobility could appear similar w.r.t. the effects on the interface, they are different from a system design point of view. In the case of interface mobility the designer must specify that an input or an output operation has to be performed without knowing its name and its templates on the contrary, in the case of internal process mobility, the designer does not know the process which will be executed at all.

Some considerations about code mobility issues are necessary. On the one hand when a service executes a process which has been acquired at run-time, it does not know how it behaves. On the other hand, when programming a process which will be executed by another service the internal behavior of such a service is not known. This fact implies a number of issues. First of all, internal processes share the variables state thus the acquired process could interfere with the behavior of the other ones. Moreover, an acquired process could exploit a certain name s to perform internal synchronizations but the same name could be already used by other internal processes, thus alterating also in this case the behavior of the other processes. A formal analysis of these issues

© 2007 ACADEMY PUBLISHER

is out of the scope of this paper but we consider that, to avoid at least the issues listed above, a mechanisms which syntactically renames all the variables and names of the acquired process which interferes with the ones of the internal processes is necessary before executing it.

IV. Web service technology

In this section we briefly present the Web Service technology and we discuss the mobility mechanisms presented in the previous sections w.r.t. it. Furthermore, we discuss a an hidden form of mobility related to the Request-Response pattern.

A. Web Service technology

Web Service technology is a service oriented architecture which achieves interoperability by exploiting XML dialects. It is born from the simple concept of the remote procedure call wrapped by a standardized interface and it is defined up to three basic specifications: WSDL [9], SOAP [10] and UDDI [11]. WSDL is an XML-based language which allows for the specification of the operations (One-Way, Request-Response, Notification and Solicit-Response) exhibited by a service, SOAP defines the message exchange protocol between two services and UDDI is a specification that deals with discovery Web Service registers. Stalelessness, loosely coupling, open endedness and compositionality are the most important features Web Service technology is characterized by. Statelessness deals with the fact that Web services do not mantain the state of a conversation. Each message exchange is a new message exchange completely separated from the previous one. Loosely coupling is intrinsically linked to the concept of business activity which can be intended as a distribuited software application where multiple entities interact each other in order to achieve a specific goal. In this context, since applications can run for a long period of time and resources cannot be blocked because of the risk of deadlock, long running transactions and compensation mechanisms play a fundamental role. In Web Service technology open endedness is achieved by exploiting particular web services which work as service registers mantaining service descriptions and locations. On the contrary, compositionality is addressed by different kind of languages: choreography languages and orchestration languages. The former supply a local description of the web services and deal with the design of the so called orchestrator engines which are web services with the peculiarity to be able to invoke other ones. On the other hand, the latter aims at describing a web service system by supplying a global system view where it is possible to design the interaction among the involved participants. The most credited orchestration language is WS-BPEL [1] which supports compositional operators as parallel, sequence and choice and it has specific primitives to interact with other web services which resemble the input and output operation processes of the OL calculus. As far as choreography languages are concerned the most credited proposal in Web Service technology is WS-CDL [12]. Such a kind of language allows for the designing of the interactions between system participants, the so called roles, by means of compositional operators as sequence, parallel and choice.

B. Web Service mobility mechanisms

- ***Internal state mobility***: Since Web Services are a message passing technology, they fully support the internal state mobility as we have formalized it in Section 3. In particular, an information exchange between two services is an XML document whose schema is defined within the SOAP specification.
- ***Location mobility***: As we have shown in Section 3 location mobility is strictly related to the communication mechanisms of the internal process that we have formalized by exploiting OL. Although that Web Services are platform independent and there is not a standard formalism for describing the internal process, here we consider orchestration languages as a class of languages which can be used for expressing it. Indeed, they deal with service coordination aspects which are fundamental to the end of location mobility. In particular, WS-BPEL supports location mobility by managing endpoints within its internal variables. An endpoint, which is defined within WS-Addressing [13] specification, is a data structure which contains all the information required for invoking a service, that is the operation and the location.
- ***Interface mobility***: The interface mobility that we have formalized in Section 3 is strictly related to the communication mechanisms of the internal process. Following the same approach of location mobility we consider WS-BPEL. As previously mentioned, WS-BPEL is able to manage endpoints which contain the information related to the operations. However it does not support interface mobility because the operations it exploits for invoking and receiving messages are defined statically at design time and they cannot be bound at run-time. To the best of our knowledge interface mobility is not supported by the Web Services technology even if it is possible to consider other solutions that indirectly allows us to partially achieve it. Several programming languages, at a low-level w.r.t. the orchestration ones, are equipped of libraries which permit to simplify the service composition. In particular, there exist libraries in Java [14]–[16] that, given a WSDL document[4], automatically produce the corresponding classes which allow for the invocation of all the operations supplied by the Web service described in that document. Such a kind of libraries allows us to

[4]A WSDL interface could be modeled by exploiting the service interface I defined in section 2 but there are some relevant issues to take into account: a WSDL document is statically defined and cannot change dynamically during the evolution of the service by adding or removing some of the exhibited operations and, generally, Notification and Solicit-Response operations are unused

© 2007 ACADEMY PUBLISHER

partially achieve interface mobility. The interfaces indeed are not communicated as information but extracted from a WSDL document. Furthermore, they cannot be joint automatically with the service internal variables but, at the state of the art, they require to be joint by considering the presence of a human designer.

- ***Internal process mobility***: To the best of our knowledge Web Services technology does not explicitly support such a kind of mobility. Nevertheless we trace a comparison between service functionality mobility and some languages for describing conversational behaviours of service-based systems as, for instance, choreography languages. As we have said, such a kind of languages are exploited for describing the communication protocols services have to follow in order to participate to a given service-based system. We can imagine that a service which is willing to access that system could download the related choreography document and extracts a piece of code which allows it to follows the protocol.

(SOLICIT)
$$\frac{\vec{t} \vdash \mathcal{S}(\tilde{x})}{(\bar{o}_{\vec{t},\vec{t'}}@l(\tilde{x},\tilde{y}),\mathcal{S}) \stackrel{\bar{o}_{\vec{t},\vec{t'}}@l(\mathcal{S}(\tilde{x}),\tilde{y})}{\longrightarrow} (o_{\vec{t},\vec{t'}}@l(\tilde{y}),\mathcal{S})}$$

(REQUEST)
$$\frac{\vec{t} \vdash \tilde{v}}{(o_{\vec{t},\vec{t'}}(\tilde{x},\tilde{y},P),\mathcal{S}) \stackrel{o_{\vec{t},\vec{t'}}@l(\tilde{v},\tilde{y})}{\rightarrow} (P;\bar{o}_{\vec{t},\vec{t'}}@l(\tilde{y}),\mathcal{S}[\tilde{v}/\tilde{x}])}$$

(RESPONSE-OUT)
$$\frac{\vec{t'} \vdash \mathcal{S}(\tilde{x})}{(\bar{o}_{\vec{t},\vec{t'}}@l(\tilde{x}),\mathcal{S}) \stackrel{\bar{o}_{\vec{t},\vec{t'}}@l(\mathcal{S}(\tilde{x}))}{\longrightarrow} (\mathbf{0},\mathcal{S})}$$

(RESPONSE-IN)
$$\frac{\vec{t'} \vdash \tilde{v}}{(o_{\vec{t},\vec{t'}}@l(\tilde{x}),\mathcal{S}) \stackrel{o_{\vec{t},\vec{t'}}@l(\tilde{v})}{\rightarrow} (\mathbf{0},\mathcal{S}[\tilde{v}/\tilde{x}])}$$

(REQ-SYNC)
$$\frac{[P_S]_l \stackrel{\sigma}{\longrightarrow} [P'_S]_l \ , \ [Q_S]_{l'} \stackrel{\sigma'}{\longrightarrow} [Q'_S]_{l'}}{[P_S]_l \parallel [Q_S]_{l'} \stackrel{\tau}{\longrightarrow} [P'_S]_l \parallel [Q'_S]_{l'}} \quad \begin{cases} \sigma = \bar{o}_{\vec{t},\vec{t'}}@l'(\tilde{v},\tilde{y}) \\ \sigma' = o'_{\vec{t''},\vec{t'''}}@l(\tilde{v},\tilde{y}) \\ \bar{o}_{\vec{t},\vec{t'}} \bowtie o'_{\vec{t''},\vec{t'''}} \end{cases}$$

(RESP-SYNC)
$$\frac{[P_S]_l \stackrel{\bar{o}_{\vec{t},\vec{t'}}@l'(\tilde{v})}{\rightarrow} [P'_S]_l \ , \ [Q_S]_{l'} \stackrel{o_{\vec{t},\vec{t'}}@l(\tilde{v})}{\rightarrow} [Q'_S]_{l'}}{[P_S]_l \parallel [Q_S]_{l'} \stackrel{\tau}{\rightarrow} [P'_S]_l \parallel [Q'_S]_{l'}}$$

TABLE IV.
REQUEST-RESPONSE PATTERN RULES Á LA WEB SERVICE

C. The hidden mobility of the Request-Response

In this section we discuss the Request-Response interaction pattern and in particular we compare the one we propose with the one supported by the Web Service technology. Usually the Request-Response interaction pattern has been intended as a powerful mechanism which is able to relate the two message exchanges involved within a Request-Response as modeled in our calculus and in [17], [18]. In particular, these proposals formalize the Request-Response behaviour by joining the output operation process with the input one. As far as our proposal is concerned, in Table 3 we have exploited a fresh label n in order to couple the sender operation processes and the receiver one.

In the Web Services technology the Request-Response interaction is not supported at the service application level but, as specified by the WSDL recommendation, it has to be supplied by the communication infrastructure (e.g. HTTP) which exploits the service locations and the operation names to bind the two message exchanges instead of a reference of the service processes involved in the interactions as in our calculus. Table IV reports the semantics rules governing the Request-Response interaction pattern á la Web Services. Such a semantics, that we consider faithful w.r.t. the Web Services technology, represents a meaningful contribute towards the formal reasoning of the current technology features and lacks. As it emerges by rules REQUEST and REQ-SYNC, there exists a hidden form of location mobility that is used by the infrastructure to support the response phase. Indeed, the infrastrure keeps the location of the invoker and exploits it for sending the response. It is worth noting that, in this case, the only references for coupling the sender and the receiver during the response phase are the service location and the operation name. This means that if a service invokes two times a Request-Response operation at the same service location the two responses could be swapped each other. Example 4.1, which follows, reveals that the Request-Response pattern supported by the Web Service technology is weaker than the one previously proposed.

Example 4.1: Let us consider the following example where a service, say A, provides a functionality which computes, given two numbers a and b, $|a| - |b|$. Such a service exploits another service, located at l, which supplies the absolute value and the subtraction functionality supplied by means of the Request-Response operations ABS and SUB, respectively. Let OP be the Request-Response operation A uses to supply its functionality, the service could be programmed as it follows (we do not describe the variables state since its initial configuration does not alterate the behaviour):

$$\vec{t} = \langle inf\ inf \rangle \qquad \vec{t'} = \langle inf \rangle$$

$$A ::= OP_{\vec{t},\vec{t'}}(\langle a,b \rangle, res, P)$$

$$P ::= (\overline{\mathrm{ABS}}_{\vec{t'},\vec{t'}}@l(a,absA) \mid \overline{\mathrm{ABS}}_{\vec{t'},\vec{t'}}@l(b,absB)); \\ \overline{\mathrm{SUB}}_{\vec{t},\vec{t'}}@l(\langle absA, absB \rangle, res)$$

In the case the Request-Response mechanisms is the one

© 2007 ACADEMY PUBLISHER

modeled by rules of Table IV, there exists an execution path where the responses of the two ABS invocations can be swapped and then, in this case, the OP response is $|b| - |a|$ instead of the expected value $|a| - |b|$. On the contrary, in the case the Request-Response mechanism is modeled as in section 3 such a behavior is not allowed.

V. CONCLUSION

In this work we have discussed the mobility aspects of service-oriented computing. We have caught the essence of a service by modeling it as a tuple of four basic components (state, location, interface, process) and we have discussed a specific form of mobility for each of them. Namely, we have modeled such a tuple by extending a formal language defined in our previous works that has been exploited as a formal workbench for highlighting the peculiarities of each kind of mobility. Finally, we have analyzed the Web Service technology in order to show which kinds of mobility are actually supported. The discussion about Web Service shows that only the internal state and the location mobility are supported by this technology. On the other hand, interface mobility and internal process mobility raise some interesting issues from the system design point of view. In this sense our formal framework could help on the one hand designers to investigate about the issues related with these kinds of mobility that, as shown by the examples we discuss, provide a mean to design real business applications and, on the other hand, to enrich current tecnologies with new mobility mechanisms. Moreover, we have modeled the behavior of the Request-Response interactions supported by the Web Service by discussing how it seems to be weaker than the common acceptation, supported also by ORC [19], that is the one we propose in our model.

The contribute of this paper is twofold, on the one hand we have formalized the mobility aspects of service oriented computing and on the other hand we have discussed them by analyzing the current technology state of the art. To the best of our knowledge this is the first attempt to strictly formalize mobility aspects of the service oriented computing paradigm. There are several works which exploit other formalisms like pi-calculus [17], [20] and Petri-nets [21] for dealing with service-based composition but a comprehensive investigation on mobility does not exist.

In our previous work we have defined a formal framework devoted to represent the peculiarities of choreography and orchestration languages and their interdependencies. It emerges that orchestration is a further developement step w.r.t. the choreography which defines the conversation rules among participants. A conformance notion captures such a relationship and permits to verify whether an orchestrated system behaves accordingly with a given choreography. In this paper we have enriched the orchestration language (here called service-based language) with mobility aspects and, as a future work, we plan on the one hand to rephrase the choreography language and the conformance notion by considering the issues raised by mobility mechanisms and, on the other hand, we intend to enrich our formal framework by introducing other fundamental aspects like sessions.

REFERENCES

[1] *Web Services Business Process Execution Language Version 2.0, Working Draft*, OASIS, [http://www.oasis-open.org/committees/download.php/10347/wsbpel-specification-draft-120204.htm].

[2] S. Thatte, "XLANG: Web Services for Business Process Design," [http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm], Microsoft Corporation, 2001.

[3] F. Leymann, "Web Services Flow Language (WSFL 1.0)," [http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf], Member IBM Academy of Technology, IBM Software Group, 2001.

[4] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro, "Choreography and orchestration: A synergic approach for system design." in *ICSOC*, ser. LNCS, vol. 3826, 2005, pp. 228–240.

[5] ——, "Choreography and orchestration conformance for system design," in *Proc. of 8th International Conference on Coordination Models and Languages (COORDINATION'06)*, ser. LNCS, vol. 4038, 2006, pp. 63–81.

[6] A. Barros, M. Dumas, and A. H. ter Hofstede, "Service interaction patterns: Towards a reference framework for service-based business process interconnection," *Tech. Report FIT-TR-2005-02,Faculty of information Technology, Queensland University of technology, Brisbane, Australia, March 2005.*

[7] A. Barros and E. Borger, "A compositional framework for service interaction patterns and interaction flows," in *Proc. of International conference on formal engineering methods (ICFM 2005)*, ser. LNCS. Springer Verlag, 2005, pp. 5–35.

[8] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo, "Formalizing web services choreographies," in *Proc. of 1st International Workshop on Web Services and Formal Methods (WS-FM 2004)*, ser. ENTCS, M. Bravetti and G. Zavattaro, Eds., vol. 105. Elsevier, 2004.

[9] *Web Services Description Language (WSDL) 1.1*, World Wide Web Consortium, [http://www.w3.org/TR/wsdl].

[10] *SOAP Version 1.2 Part 1: Messaging Framework*, World Wide Web Consortium, [http://www.w3.org/TR/soap12-part1/].

[11] *UDDI - Universal Description, Discovery and Integration of Web Services*, Oasis, [http://www.uddi.org/specification.html].

[12] *Web Services Choreography Description Language Version 1.0. Working draft 17 December 2004*, World Wide Web Consortium, [http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/].

[13] *Web Services Addressing*, W3C member submission 10 august, 2004, [http://www.w3.org/submission/ws-addressing/].

[14] *Axis (WSDL2Java)*, Apache, [http://ws.apache.org/axis/index.html].

[15] *Axis (Java2WSDL)*, Apache, [http://ws.apache.org/axis/index.html].

[16] *Java Web Services Developer Pack*, Sun microsystems, [http://java.sun.com/webservices/downloads/webservicespack.html].

[17] R. Lucchi and M. Mazzara, "A pi-calculus based semantics for WS-BPEL," *Journal of Logic and Algebraic Programming*, vol. 70 issue 1 Web services and formal methods, pp. 96–118, January 2007.

© 2007 ACADEMY PUBLISHER

[18] J. Misra and W. Cook, "Computation orchestration, a basis for wide-area computing," *Journal of Software and Systems modeling*, to appear.
[19] J. Misra and W. R. Cook, "Computation orchestration: A basis for wide-area computing," *Journal of Software and Systems Modeling*, 2006, to appear. A preliminary version of this paper appeared in the Lecture Notes for NATO summer school, held at Marktoberdorf in August 2004.
[20] L. Bocchi, C. Laneve, and G. Zavattaro, "A Calculus for Long-Running Transactions," in *FMOODS*, ser. LNCS, vol. 2884. Springer Verlag, 2003, pp. 124–138.
[21] R. Dijkman and M. Dumas, "Service-oriented Design: a Multi-viewpoint Approach," *Int. J. Cooperative Inf. Syst.*, vol. 13, no. 4, pp. 337–368, 2004.

Claudio Guidi is currently a Ph.D student at the Department of Computer Science of the University of Bologna. His Ph.D. thesis title is "Formalizing languages for Service Oriented Computing".

Roberto Lucchi received his Ph.D. degree in Computer Science in 2004 at the University of Bologna with a thesis on "Security, Probability and Priority in the tuple-space Coordination Model". Currently he is working at the European Commission, DG Joint Research Centre, Institute for Environment and Sustainability, Spatial Data Infrastructures Unit. Before joining JRC he was employed in the SENSORIA (Software Engineering for Service-Oriented Overlays Computers) EU Integrated Project. His research interests include formal methods, security, coordination models and languages for the service oriented computing paradigm.

APPENDIX

The syntax of χ is

$$\chi ::= x \leq e \mid e \leq x \mid \neg\chi \mid \chi \wedge \chi$$

where e denotes an expression which can contain variables references and which can be evaluated into a value v or, when some variables within the expression are not instantiated, into the symbol $\perp$.

The satisfaction relation for $\vdash$ is defined by the following rules:

1) $\mathcal{S}(x) = \perp \Rightarrow \mathcal{S} \vdash (x \leq \perp \wedge \perp \leq x)$
2) $e \hookrightarrow_{\mathcal{S}} v, \mathcal{S}(x) \leq v \Rightarrow \mathcal{S} \vdash x \leq e$
3) $e \hookrightarrow_{\mathcal{S}} v, v \leq \mathcal{S}(x) \Rightarrow \mathcal{S} \vdash e \leq x$
4) $\mathcal{S} \vdash \chi' \wedge \mathcal{S} \vdash \chi'' \Rightarrow \mathcal{S} \vdash \chi' \wedge \chi''$
5) $\neg(\mathcal{S} \vdash \chi) \Rightarrow \mathcal{S} \vdash \neg\chi$

We highlight the fact that rule 1 states that when a variable x is defined with value $\perp$ the only condition which can be satisfied on such a state is $x = \perp$.

© 2007 ACADEMY PUBLISHER