Mobility Mechanisms in Service Oriented Computing*

Claudio Guidi and Roberto Lucchi

Department of Computer Science, University of Bologna, Italy
{cguidi, lucchi}@cs.unibo.it

Abstract. The usual context of service oriented computing is characterized by several services offering the same functionalities, new services that are continuosly deployed and other ones that are removed. In this case it can be useful to discover and compose services dynamically 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 where services acquire during the execution the necessary information to invoke services. In this paper we discuss the mobility mechanisms for the service oriented computing paradigm. To this end we model a service by means of the notions of interface, location, process and internal state, then we formalize a calculus supporting a specific form of mobility for each of them. We conclude by comparing mobility mechanisms of our calculus with the ones supported by the Web Services technology.

1 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 [12, 14, 9] 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 all the corresponding results, thus the state of the execution is controlled in a centralized way within the orchestrator process.

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

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

R. Gorrieri and H. Wehrheim (Eds.): FMOODS 2006, LNCS 4037, pp. 233–250, 2006.

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 discusses the mobility mechanisms in service oriented computing. We proceed as follows: i) we define a 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 supporting mobility mechanisms. In particular, we characterize a service by means of four components: the *location*, the *process*, the *interface* and the *internal state*. The location expresses where the service is deployed and then available, the process represents the program which permits to supply the service functionalities, the interface represents the acess points the service can use to interact with other ones and, finally, the internal state represents the information the service internally manages. 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 service functionality mobility, the interface mobility and the internal state mobility. Once having discussed each of them we proceed by presenting a service-based calculus we use to formally describe these mechanisms. Such a calculus, equipped of an operational semantics, is an extension of a previous work [7, 6] obtained by introducing the notion of service location. At the end we trace a comparison between the mechanisms 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. In particular, 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 2 defines a service and reasons about the meaning of the various forms of mobility that could be supported between services. Section 3 presents the service-based calculus supporting mobility mechanisms and its operational semantics. Section 4 compares the mobility mechanisms we propose with the Web Services technology. Section 5 concludes the paper with some final remarks.

2 Services Formalization and Mobility Mechanisms

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

2.1 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. The service communication mechanism is based on peer-to-peer message passing. Every information that needs to be exchanged between two services is to be communicated by means of interaction points. Each service 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 [4, 3], 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.

The set of all the operations exhibited by a service represents the *interface* of the service. In order to send a request message, a service has to explicit the output operation and the location of the receiver. In other words, the operation expresses *how* to invoke a service whereas the location specifies *where* the service can be accessed.

Let *Loc* be the set of service locations, $\mathcal{O}$ and $\mathcal{O}_R$ be two disjoint sets of operation names, $Sup = \{(o, ow) \mid o \in \mathcal{O}\} \cup \{(o_r, rr) \mid o_r \in \mathcal{O}_R\}$ be the set containing all the input operations where *ow* and *rr* indicate One-Way and Request-Response operations, respectively. Let $Inv = \{(o, n) \mid o \in \mathcal{O}\} \cup \{(o_r, sr) \mid o_r \in \mathcal{O}_R\}$ be the set containing all the output operations where *n* and *sr* denote Notification and Solicit-Response operations. Let $Interfaces = Sup \cup Inv$ be the set of all the possible operations. By definition an operation name unambiguously identifies a couple of operations: a One-Way with a Notification and a Request-Response with a Solicit-Response. This is related to the fact that an operation in *Sup* can be invoked only by the corresponding operation in *Inv* that has the same name.

Formally a service is defined by the following tuple:

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

where $I \subseteq \mathit{Interfaces}$ 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 process which expresses the service functionality encoded by exploiting the formalism f and $l \in \mathit{Loc}$ 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 following section such notions will be represented by a specific model.

2.2 Mobility Mechanisms

In this section we describe the mobility mechanisms which deal with open endedness. 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 service functionality mobility. Since the interaction mechanism is based on message passing, mobility is achieved by communicating service components by means of exchanged messages. This fact has a significant impact on 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. This means that a service can acquire at run-time the exact location of a service whose functionalities are known, as in the case of the *update* service discussed in the Introduction section which knows the *client* functionality but not their locations.
- ***Interface mobility:*** Interface mobility means that a service can acquire at run-time an operation and exhibits it in its interface. In particular, such a kind of mobility deals only with the mobility of the operation name (by definition the interaction modality can be derived by its name). Thus, the service which receives an operation can exhibit it either as an output operation and an input one. Since operations provides access points to the service functionalities, which are supplied by the service by means of its internal process, we consider that the only reasonable usage of an operation acquired at run-time is for exhibiting the related output operation and not the input one. The calculus we propose in the following section allows to exhibit acquired operations only as output operations.
- ***Functionality 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 receiving 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. In this paper we do not discuss such a problem that we consider orthogonal to the mobility mechanisms.

3 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.

3.1 The Service-Based Language

Here, we present a service-based calculus which extends OL, defined in our previous works, by means of locations. Such a language allows us to describe systems where each participant is a service[1] 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 *InternalLink* be a set of names ranged over by s, let Var be the set of variables ranged over by x, y, z, k. We denote with $\widetilde{x}$ tuples of variables, for instance, we may have $\widetilde{x} = \langle x_1, x_2, ..., x_n \rangle$. Let W be a finite ordered non-empty set of indexes, OL is defined by the following grammar:

$$
\begin{array}{l}
P ::= \mathbf{0} \mid x := e \mid \epsilon \mid \bar{s} \mid \bar{o}@l(\tilde{x}) \mid \bar{o}_r@l(\tilde{x}, \tilde{y}) \\
\quad \mid P; P \mid P \mid P \mid \sum^{+}_{i \in W} \epsilon_i; P_i \mid \sum^{\oplus}_{i \in W} \chi_i ? P_i \\
\epsilon ::= s \mid o(\tilde{x}) \mid o_r(\tilde{x}, \tilde{y}, P) \\
E ::= [P, \mathcal{S}]_l \mid E \parallel E
\end{array}
$$

where a service-based system E consists of the parallel composition of services. A service $[P, \mathcal{S}]_l$ is a process P identified by its location $l \in Loc$ whose variables state is $\mathcal{S}$. The variables state of a service is described by a function $\mathcal{S} : Var \rightarrow Val \cup \{\perp\}$ from variables to the set $Val \cup \{\perp\}$ ranged over by w. Val, ranged over by v, is a generic set of values on which is defined a total order relation[2]. $\mathcal{S}(x)$ represents the value of variable x in the state $\mathcal{S}$ ($\mathcal{S}(x) = \perp$ means that x is

[1] 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 indicating both orchestrators and simple services.

[2] We extend such an order relation on the set $Val \cup \{\perp\}$ considering $\perp < v, \ \forall v \in Val$.

not yet initialized), while $\mathcal{S}[v/x]$ denotes the state $\mathcal{S}$ where x holds value v (we use $\mathcal{S}[\tilde{v}/\tilde{x}]$ when dealing with tuples of variables), formally:

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

All the services are executed at different locations, thus they can be composed by using only the parallel operator ($\|$). 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 among input guarded processes, that represent exhibited operations, whereas the operator $\sum_{i\in W}^{\oplus} \chi_i ? P_i$ expresses a deterministic choice among processes guarded by conditions on variables state (such processes are of the form $\chi ? P$ where χ is a logic condition on the state $\mathcal{S}$ associated to P whose syntax is reported in Appendix A). $\mathbf{0}$ represents the null process whereas the processes $x := e$ deals with variable assignment. Processes s and $\bar{s}$ deal with internal service synchronizations which are exploited to coordinate the activities of processes running in parallel. In this case no message is exchanged; this is because the service variables are shared by all the processes running on that service. As far as the operations are concerned, the process $o(\tilde{x})$ represents a One-Way operation where o ranges over $\mathcal{O}$, whereas the process $o_r(\tilde{x}, \tilde{z}, P)$ represents the Request-Response one where o_r ranges over $\mathcal{O}_R$. Namely, $o(\tilde{x})$ represents a One-Way operation whose name is o and the received information are stored in the tuple of variable $\tilde{x}$, while $o_r(\tilde{x}, \tilde{y}, P)$ represents a Request-Response operation named o_r which 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. On the contrary, the processes $\bar{o}@l(\tilde{x})$ and $\bar{o}_r@l(\tilde{x}, \tilde{y})$ represent the Notification and the Solicit-Response operations respectively, where o ranges over $\mathcal{O}$ and o_r ranges over $\mathcal{O}_R$. In particular, $\bar{o}@l(\tilde{x})$ invokes the operation o of the service located at l sending the information contained in $\tilde{x}$ whereas $\bar{o}_r@l(\tilde{x}, \tilde{y})$ invokes the operation o_r of the service located at l sending the information contained in $\tilde{x}$ and waits for the response whose information will be stored in $\tilde{y}$.

The semantics of OL is defined in terms of a labelled transition system which describes the evolution of a service-based system. We define $\rightarrow$ as the least relation which satisfies the axioms and rules of Tables 1, 2 and 3. Let $Act_{OL} = \{\bar{o}, o, \bar{o}@l(\tilde{v}), o(\tilde{v}), \bar{o}_r^n(\tilde{v}), o_r^n(\tilde{v}), \bar{o}_r@l(\tilde{v}, \tilde{y})(n), o_r@l(\tilde{v}, \tilde{y})(n), \sigma, \tau\}$ be the set of actions ranged over by γ. σ is a parameterized action of the form $(l, l', op, \tilde{v}, dir)$ where l, l' are service locations, op is an operation name, $\tilde{v}$ are tuples of values and $dir \in \{\uparrow, \downarrow\}$. We exploit dir for discriminating between a request message and a response one. Table 1 deals with the axioms over P where we have introduced the processes $o_r^n(\tilde{x})$ and $\bar{o}_r^n(\tilde{x})$ in order to deal with Request-Response and Solicit-Response mechanisms. The most interesting axiom is the REQUEST one, which describes that when it is invoked, the operation behaves as the process that performs P and, once having completed such a process, performs an output that is consumed by the invoking service. On the contrary, rules SOLICIT and

Table 1. Axioms over P

$$\text{(In)}\quad (s,\mathcal{S}) \xrightarrow{s} (\mathbf{0},\mathcal{S}) \qquad \text{(Out)}\quad (\bar{s},\mathcal{S}) \xrightarrow{\bar{s}} (\mathbf{0},\mathcal{S})$$

$$\text{(Notification)}\quad (\bar{o}@l(\tilde{x}),\mathcal{S}) \xrightarrow{\bar{o}@l(\tilde{v})} (\mathbf{0},\mathcal{S}), \tilde{v} = \mathcal{S}(\tilde{x}) \qquad \text{(One-Way)}\quad (o(\tilde{x}),\mathcal{S}) \xrightarrow{o(\tilde{v})} (\mathbf{0},\mathcal{S}[\tilde{v}/\tilde{x}])$$

$$\text{(Solicit)}\quad (\bar{o}_r@l(\tilde{x},\tilde{y}),\mathcal{S}) \xrightarrow{\bar{o}_r@l(\tilde{v},\tilde{y})(n)} (o_r^n(\tilde{y}),\mathcal{S}), \tilde{v} = \mathcal{S}(\tilde{x}) \qquad \text{(Request)}\quad (o_r(\tilde{x},\tilde{y},P),\mathcal{S}) \xrightarrow{o_r@l(\tilde{v},\tilde{y})(n)} (P;\overline{o}_r^n(\tilde{y}),\mathcal{S}[\tilde{v}/\tilde{x}])$$

$$\text{(Response-Out)}\quad (\bar{o}_r^n(\tilde{x}),\mathcal{S}) \xrightarrow{\bar{o}_r^n(\tilde{v})} (\mathbf{0},\mathcal{S}), \tilde{v} = \mathcal{S}(\tilde{x}) \qquad \text{(Response-In)}\quad (o_r^n(\tilde{x}),\mathcal{S}) \xrightarrow{o_r^n(\tilde{v})} (\mathbf{0},\mathcal{S}[\tilde{v}/\tilde{x}])$$

Response-In deal with Solicit-Response behaviour where, initially, a message is sent and then the service, by means of the process $o_r^n(\tilde{x})$, waits for the response.

Table 2 deals with the rules over P where rule Assign deals with variable assignment within the services; $e \hookrightarrow_{\mathcal{S}} v$ means that the evaluation process of the expression e within state $\mathcal{S}$ reduces to v. Rule Int-Sync deals with internal synchronization 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 Choice1 and Choice2 describe the behavior of the two alternative composition operators. The former one nondeterministically selects an input guarded process among the ones listed in the choice operator, while the latter one is the deterministic choice depending on the internal state of the service where the satisfaction relation for $\vdash$ is reported in Appendix A. In Table 3 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.

Now, we remind the service formalization presented in section 2 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 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:

Table 2. Rules over P

$$\text{(Assign)}\ \frac{e \hookrightarrow_{\mathcal{S}} v}{(x := e, \mathcal{S}) \xrightarrow{\tau} (\mathbf{0}, \mathcal{S}[v/x])} \qquad \text{(Int-Sync)}\ \frac{(P,\mathcal{S}) \xrightarrow{s} (P',\mathcal{S})\ ,\ (Q,\mathcal{S}) \xrightarrow{\bar{s}} (Q',\mathcal{S})}{(P \mid Q, \mathcal{S}) \xrightarrow{\tau} (P' \mid Q', \mathcal{S})}$$

$$\text{(CongrP)}\ \frac{P \equiv_P P'\ ,\ (P',\mathcal{S}) \xrightarrow{\gamma} (Q',\mathcal{S}'),\ Q' \equiv_P Q}{(P,\mathcal{S}) \xrightarrow{\gamma} (Q,\mathcal{S}')}$$

$$\text{(Par-Int)}\ \frac{(P,\mathcal{S}) \xrightarrow{\gamma} (P',\mathcal{S}')}{(P \mid Q, \mathcal{S}) \xrightarrow{\gamma} (P' \mid Q, \mathcal{S}')} \qquad \text{(Seq)}\ \frac{(P,\mathcal{S}) \xrightarrow{\gamma} (P',\mathcal{S}')}{(P;Q, \mathcal{S}) \xrightarrow{\gamma} (P';Q, \mathcal{S}')}$$

$$\text{(Choice 1)}\ \frac{(\epsilon_i; P_i, \mathcal{S}) \xrightarrow{\gamma} (P',\mathcal{S}') \quad i \in W}{(\sum^{+}_{i\in W} \epsilon_i; P_i, \mathcal{S}) \xrightarrow{\gamma} (P',\mathcal{S}')} \qquad \text{(Choice 2)}\ \frac{\mathcal{S} \vdash \chi_i \quad \mathcal{S} \nvdash \chi_j, j \in W, j < i}{(\sum^{\oplus}_{i\in W} \chi_i ? P_i, \mathcal{S}) \xrightarrow{\tau} (P_i, \mathcal{S})}$$

(Structural Congruenge over P)

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

1. $\Theta(\mathbf{0}) = \phi$
2. $\Theta(x := e) = \phi$
3. $\Theta(s) = \phi$
4. $\Theta(\overline{s}) = \phi$
5. $\Theta(\bar{o}@l(\tilde{x})) = \{(o, n)\}$
6. $\Theta(\bar{o}_r@l(\tilde{x}, \tilde{y})) = \{(o_r, sr)\}$
7. $\Theta(o(\tilde{x})) = \{(o, ow)\}$
8. $\Theta(o_r(\tilde{x}, \tilde{y}, P)) = \{(o_r, rr)\} \cup \Theta(P)$
9. $\Theta(o^n_r(\tilde{x})) = \phi$
10. $\Theta(\overline{o}^n_r(\tilde{x})) = \phi$
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. $\Theta(\sum^{\oplus}_{i\in W} \chi_i ? P_i) = \bigcup_{i\in W} \Theta(P_i)$

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 simple example which follows where, for the sake of brevity, we abstract away from the internal states:

$$[\bar{a}(x), \mathcal{S}]_l \parallel [a(y), \mathcal{S}']_{l'} \xrightarrow{\sigma} [\mathbf{0}, \mathcal{S}]_l \parallel [\mathbf{0}, \mathcal{S}']_{l'}$$

Before the synchronization the interfaces of the two services are $I_l = \{(a, n)\}$ and $I_{l'} = \{(a, ow)\}$ respectively, whereas after the synchronization they are $I_l = \phi$ and $I_{l'=}\phi$.

3.2 Internal State Mobility

As we have noticed in section 2 the internal state mobility is strongly related to the message passing communication mechanism. Considering Table 1 and

Table 3. Rules over E

$$\text{(One-WaySync)}\quad \frac{[P,\mathcal{S}]_l \xrightarrow{\bar{o}@l'(\tilde{v})} [P',\mathcal{S}']_l \ ,\ [Q,\mathcal{T}]_{l'} \xrightarrow{o(\tilde{v})} [Q',\mathcal{T}']_{l'} \ ,\sigma=(l,l',o,\tilde{v},\uparrow)}{[P,\mathcal{S}]_l \parallel [Q,\mathcal{T}]_{l'} \xrightarrow{\sigma} [P',\mathcal{S}']_l \parallel [Q',\mathcal{T}']_{l'}}$$

$$\text{(Req-Sync)}\quad \frac{[P,\mathcal{S}]_l \xrightarrow{\bar{o}_r@l'(\tilde{v},\tilde{y})(n)} [P',\mathcal{S}']_l \ ,\ [Q,\mathcal{T}]_{l'} \xrightarrow{o_r@l(\tilde{v},\tilde{y})(n)} [Q',\mathcal{T}']_{l'} \ ,n\ fresh,\sigma=(l,l',o_r,\tilde{v},\uparrow)}{[P,\mathcal{S}]_l \parallel [Q,\mathcal{T}]_{l'} \xrightarrow{\sigma} [P',\mathcal{S}']_l \parallel [Q',\mathcal{T}']_{l'}}$$

$$\text{(Resp-Sync)}\quad \frac{[P,\mathcal{S}]_l \xrightarrow{\bar{o}_r^n(\tilde{v})} [P',\mathcal{S}']_l \ ,\ [Q,\mathcal{T}]_{l'} \xrightarrow{o_r^n(\tilde{v})} [Q',\mathcal{T}']_{l'} \ ,\sigma=(l,l',o_r,\tilde{v},\downarrow)}{[P,\mathcal{S}]_l \parallel [Q,\mathcal{T}]_{l'} \xrightarrow{\sigma} [P',\mathcal{S}']_l \parallel [Q',\mathcal{T}']_{l'}}$$

$$\text{(Par-Ext)}\quad \frac{E_1 \xrightarrow{\gamma} E_1'}{E_1 \parallel E_2 \xrightarrow{\gamma} E_1' \parallel E_2} \qquad \text{(CongrE)}\quad \frac{E_1 \equiv E_1' \ ,\ E_1' \xrightarrow{\gamma} E_2' ,\ E_2' \equiv E_2}{E_1 \xrightarrow{\gamma} E_2} \qquad \text{(Int-Ext)}\quad \frac{(P,\mathcal{S}) \xrightarrow{\gamma} (P',\mathcal{S}')}{[P,\mathcal{S}]_l \xrightarrow{\gamma} [P',\mathcal{S}']_l}$$

(Structural Congruence over E)

$$\frac{P \equiv_P Q}{[P,\mathcal{S}]_l \equiv [Q,\mathcal{S}]_l} \qquad E_1 \parallel E_2 \equiv E_2 \parallel E_1 \qquad E_1 \parallel (E_2 \parallel E_3) \equiv (E_1 \parallel E_2) \parallel E_3$$

Table 3, 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 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 3 couples the two axioms by correlating the receiver location to that explicited within the notification process and σ is a formal representation of the exchanged message. Summarizing, internal state mobility is modeled as a 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.

3.3 Location Mobility

In order to deal with location mobility here we modify the syntax of OL by replacing the processes $\bar{o}@l(\tilde{x})$ and $\bar{o}_r@l(\tilde{x},\tilde{y})$ with the new processes which follow:

$P ::= \ldots \mid \bar{o}@z(\tilde{x}) \mid \bar{o}_r@z(\tilde{x},\tilde{y}) \mid \ldots$

where z is a variable. These novelties allow us to dynamically bind the receiver location when performing the Notification and Solicit-Response operations by evaluating the content of variable z. The semantics of axioms NOTIFICATION and SOLICIT of Table 1 change as it follows:

(NOTIFICATION)
$$(\bar{o}@z(\tilde{x}), \mathcal{S}) \xrightarrow{\bar{o}@l(\tilde{v})} (\mathbf{0}, \mathcal{S}), \begin{array}{l}\tilde{v} = \mathcal{S}(\tilde{x}) \\ l = \mathcal{S}(z)\end{array}$$

(SOLICIT)
$$(\bar{o}_r@z(\tilde{x}, \tilde{y}), \mathcal{S}) \xrightarrow{\bar{o}_r@l(\tilde{v}, \tilde{y})(n)} (o_r^n(\tilde{y}), \mathcal{S}), \begin{array}{l}\tilde{v} = \mathcal{S}(\tilde{x}) \\ l = \mathcal{S}(z)\end{array}$$

Variable z is evaluated when the processes are executed. 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 that receives 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.

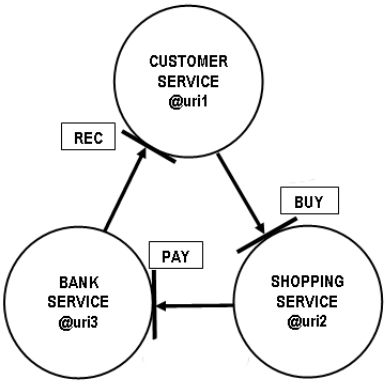

Fig. 1. Business scenario example

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

$$\begin{array}{l} System ::= [z_1 := uri2; add := uri1; inv := \bot; \overline{\mathrm{BUY}}@z_1(add); \mathrm{REC}(inv), \mathcal{S}_c]_{uri1} \\ \quad \| \; [z_2 := uri3; fwadd := \bot; \mathrm{BUY}(fwadd); \overline{\mathrm{PAY}}@z_2(fwadd), \mathcal{S}_s]_{uri2} \\ \quad \| \; [z_3 := \bot; invoice = msg; \mathrm{PAY}(z_3); \overline{\mathrm{REC}}@z_3(invoice), \mathcal{S}_b]_{uri3} \end{array}$$

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

invoice represented by the value msg. Finally, the customer stores the received invoice within the variable inv.

Location mobility is built on top of the internal state mobility because acquired locations are stored within the internal state. Such a kind of mobility allows us to design flexible services which bind their output operations at run-time.

3.4 Interface Mobility

In order to deal with interface mobility here we modify the syntax of OL by replacing the output operation processes with the new processes that follow:

$$P ::= \ldots \mid \bar{k}@z(\tilde{x}) \mid \bar{k}@z(\tilde{x},\tilde{y}) \mid \ldots$$

where z and k are variables. As far as the output operations are concerned, the operation names are evaluated at run-time by considering the value of an internal state variable (k). The new semantics of axioms NOTIFICATION and SOLICIT is as follows:

(NOTIFICATION)
$$(\bar{k}@z(\tilde{x}),\mathcal{S}) \xrightarrow{\bar{o}@l(\tilde{v})} (\mathbf{0},\mathcal{S}), \begin{array}{r} o=\mathcal{S}(k) \\ \tilde{v}=\mathcal{S}(\tilde{x}) \\ l=\mathcal{S}(z) \end{array}$$

(SOLICIT)
$$(\bar{k}@z(\tilde{x},\tilde{y}),\mathcal{S}) \xrightarrow{\bar{o}_r@l(\tilde{v},\tilde{y})(n)} (o_r^n(\tilde{y}),\mathcal{S}), \begin{array}{r} o_r=\mathcal{S}(k) \\ \tilde{v}=\mathcal{S}(\tilde{x}) \\ l=\mathcal{S}(z) \end{array}$$

Furthermore, we modify some rules for the inductive definition of Θ which allows us to extract the service interface. In particular, we modify the rules 5 and 6 which deal with the output operations:

$$5.\ \Theta(\bar{k}@z(\tilde{x}),\mathcal{S}) = \begin{cases} \{(\mathcal{S}(k),n)\}\ if\ \mathcal{S}(k)\neq\bot \wedge \mathcal{S}(k)\in\mathcal{O} \\ \phi \quad otherwise \end{cases}$$

$$6.\ \Theta(\bar{k}@z(\tilde{x},\tilde{y}),\mathcal{S}) = \begin{cases} \{(\mathcal{S}(k),sr)\}\ if\ \mathcal{S}(k)\neq\bot \wedge \mathcal{S}(k)\in\mathcal{O}_R \\ \phi \quad otherwise \end{cases}$$

It is worth noting that now the interface depends also by the internal state[3]. This is due to the fact that operation names are contained within variables. The condition $\mathcal{S}(k)\neq\bot$ guarantees that the interface contains only the known operations.

By exploiting the new output operation processes it is possible to design separately the functionalities which deal with output operations from the actual interface of the service. Let us consider the example of Fig. 1 where, now, we suppose that the bank service does not know *a priori* both the location and the one-way operation of the customer:

$$\begin{array}{l} System ::= [z_1 := uri2; add := uri1; opr_1 := \mathrm{REC}; inv := \bot \\ \quad ; \overline{\mathrm{BUY}}@z_1(\langle add, opr_1\rangle); \mathrm{REC}(inv), \mathcal{S}_c]_{uri1} \\ \quad \parallel [z_2 := uri3; fwadd := \bot; opr_2 := \bot \\ \quad ; \mathrm{BUY}(\langle fwadd, opr_2\rangle); \overline{\mathrm{PAY}}@z_2(\langle fwadd, opr_2\rangle), \mathcal{S}_s]_{uri2} \\ \quad \parallel [z_3 := \bot; k_3 := \bot; invoice = msg; \mathrm{PAY}(z_3,k_3); \overline{k_3}@z_3(invoice), \mathcal{S}_b]_{uri3} \end{array}$$

[3] Namely, the domain of Θ now considers also the internal state $\mathcal{S}$. For the sake of brevity, we do not show all the rules because they are not affected by the state.

The bank service indeed, receives from the shopping service both the location and the name of the operation of the customer and stores it in l_3 and k_3 respectively. The customer sends, by means of the variable opr_1, the operation REC on which it will wait for receiving the invoice. The example shows how is possible to design a service (in the example the bank one) with a functionality which deals with an output operation without statically knowing its interface. This fact has some implications on the service interface. By considering the new rules for Θ, the interface can also dynamically includes new operations. The interface of the bank service indeed, is $I = \{(\mathrm{PAY}, ow)\}$ before receiving a message on the PAY operation and $I = \{(\mathrm{REC}, n)\}$ after the reception of the customer operation.

3.5 Service Functionality Mobility

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

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

$\mathtt{run}(x)$ allows us to execute the code contained within the variable x. The semantics of such a primitive is expressed by a new rule that must be added to those presented in Table 2:

(Run)
$$(\mathtt{run}(x), \mathcal{S}) \xrightarrow{\tau} (\mathcal{S}(x), \mathcal{S})$$

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:

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

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. Now, 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.

$$\begin{aligned} System ::= {} & [z_1 := uri2; add := uri1; code_1 := \bot \\ & ; \overline{\mathrm{BUY}}@z_1(add, code_1); \mathtt{run}(code_1), \mathcal{S}_c]_{uri1} \\ & \parallel [z_2 := uri3; fwadd := \bot; code_2 := \text{“}inv := \bot; \mathrm{REC}(inv)\text{”} \\ & ; \mathrm{BUY}(fwadd, code_2, \mathbf{0}); \overline{\mathrm{PAY}}@z_2(fwadd), \mathcal{S}_s]_{uri2} \\ & \parallel [z_3 := \bot; invoice = msg; \mathrm{PAY}(z_3); \overline{\mathrm{REC}}@z_3(invoice), \mathcal{S}_b]_{uri3} \end{aligned}$$

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 operation has to be performed without knowing its name, on the contrary in the case of service functionality mobility the designer does not know the process which will be executed. Furthermore, by exploiting the primitive $\texttt{run}(x)$ it is possible to enrich the service interface also with both input and output operations. In the example indeed, the customer service interface is enriched with the operation (REC, ow) which is an input one.

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 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.

4 Web Services Technology

In this section we discuss the mobility mechanisms presented in the previous sections w.r.t. Web Services technology. Furthermore, we discuss a particular hidden mobility related to the Request-Response operation.

4.1 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 [16] 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, we consider WS-BPEL because it is the most credited proposal for orchestration. It supports compositional operators as parallel, sequence and choice and it has specific primitives to interact with other services which resemble the input and output operation processes of the *OL* calculus. WS-BPEL supports location mobility by managing endpoints within its internal variables. An endpoint, which is defined within WS-Addressing [15] 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 achieve it. Let us consider WSDL specification [18] that is an XML-based language which allows to specify the operations (One-Way, Request-Response, Notification and Solicit-Response) exhibited by a service[4]. 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 [2, 1, 13] that, given a WSDL document, automatically produce the corresponding classes which allow us to invoke all the operations supplied by the Web service described in that document. In this case we can guess that by exploiting such languages and libraries we can also support interface mobility.
- ***Service functionality 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, WS-CDL [17]. Such 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 WS-CDL document and extracts a piece of code which allows it to follows the protocol.

[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 can not 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.

4.2 The Hidden Mobility of the Request-Response

In this section we discuss the Request-Response interaction mechanism and in particular we compare the one we propose with the one supported by the Web Services 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 [10, 11]. 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 two processes.

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 to bind the two message exchanges instead of the service processes involved in the interactions as in our calculus. This means that if a service invokes two times a Request-Response operation at the same service location the two responses could be swapped with each other. Example 1, which follows, reveals that the interaction mechanism supported by the Web Services technology is weaker than the one previously proposed.

Table 4 reports the semantics rules governing the Request-Response interaction pattern á la Web Services. As it emerges by the semantics rules, 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 invoker and uses it when the response is to be sent. 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.

Example 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

Table 4. Modified rules for Request-Response

(Solicit)
$$(\bar{o}_r@l(\tilde{x},\tilde{y}),\mathcal{S}) \xrightarrow{\bar{o}_r@l(\tilde{v},\tilde{y})} (o_r@l(\tilde{y}),\mathcal{S}), \tilde{v} = \mathcal{S}(\tilde{x})$$

(Request)
$$(o_r(\tilde{x},\tilde{y},P),\mathcal{S}) \xrightarrow{o_r@l(\tilde{v},\tilde{y})} (P; \bar{o}_r@l(\tilde{y}),\mathcal{S}[\tilde{v}/\tilde{x}])$$

(Response-Out)
$$(\bar{o}_r@l(\tilde{x}),\mathcal{S}) \xrightarrow{\bar{o}_r@l(\tilde{v})} (\mathbf{0},\mathcal{S}), \tilde{v} = \mathcal{S}(\tilde{x})$$

(Response-In)
$$(o_r@l(\tilde{x}),\mathcal{S}) \xrightarrow{o_r@l(\tilde{v})} (\mathbf{0},\mathcal{S}[\tilde{v}/\tilde{x}])$$

(Req-Sync)
$$\frac{[P,\mathcal{S}]_l \xrightarrow{\bar{o}_r@l'(\tilde{v},\tilde{y})} [P',\mathcal{S}']_l \ ,\ [Q,\mathcal{T}]_{l'} \xrightarrow{o_r@l(\tilde{v},\tilde{y})} [Q',\mathcal{T}']_{l'} \ , \sigma = (l,l',o,\tilde{v},\uparrow)}{[P,\mathcal{S}]_l \parallel [Q,\mathcal{T}]_{l'} \xrightarrow{\sigma} [P',\mathcal{S}']_l \parallel [Q',\mathcal{T}']_{l'}}$$

(Resp-Sync)
$$\frac{[P,\mathcal{S}]_l \xrightarrow{\bar{o}_r@l'(\tilde{v})} [P',\mathcal{S}']_l \ ,\ [Q,\mathcal{T}]_{l'} \xrightarrow{o_r@l(\tilde{v})} [Q',\mathcal{T}']_{l'} \ , \sigma = (l,l',o,\tilde{v},\downarrow)}{[P,\mathcal{S}]_l \parallel [Q,\mathcal{T}]_{l'} \xrightarrow{\sigma} [P',\mathcal{S}']_l \parallel [Q',\mathcal{T}']_{l'}}$$

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):

$$A ::= OP(\langle a, b\rangle, res, P)$$
$$P ::= (\overline{\mathrm{ABS}}(a, absA)@l \mid \overline{\mathrm{ABS}}(b, absB)@l); \overline{\mathrm{SUB}}(\langle absA, absB\rangle, res)@l$$

In the case the Request-Response mechanisms is the one modeled by rules of Table 4, 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.

5 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 Services technology in order to show which kinds of mobility are actually supported. The discussion about Web Services shows that only the internal state mobility, by means of message passing communication mechanism, and the location mobility are supported by this technology. On the other hand, interface mobility and service functionality mobility raise some interesting issues from the system design point of view. In this sense our formal investigation could be a good starting point for enriching the actual technologies with these new kinds of mobility. Moreover, we have modeled the behavior of the Request-Response interactions supported by the Web Services by discussing how it seems to be weaker than 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 [10, 5] and Petri-nets [8] 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. Apache. *Axis (Java2WSDL)*. [http://ws.apache.org/axis/index.html].
2. Apache. *Axis (WSDL2Java)*. [http://ws.apache.org/axis/index.html].
3. 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)*, LNCS, pages 5–35. Springer Verlag, 2005.
4. A. Barros, M. Dumas, and A. H.M. 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.*
5. L. Bocchi, C. Laneve, and G. Zavattaro. A Calculus for Long-Running Transactions. In *FMOODS*, volume 2884 of *LNCS*, pages 124–138. Springer Verlag, 2003.
6. Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and orchestration conformance for system design. In *Proc. of 8th International conference on Coordination Models and Languages (Coordination 2006), To appear.*
7. Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and orchestration: A synergic approach for system design. In *ICSOC*, pages 228–240, 2005.
8. Remco Dijkman and Marlon Dumas. Service-oriented Design: a Multi-viewpoint Approach. *Int. J. Cooperative Inf. Syst.*, 13(4):337–368, 2004.
9. 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.
10. R. Lucchi and M. Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming.* Elsevier Press. To appear.
11. J. Misra and W. Cook. Computation orchestration. *Software and Systems modeling.* To appear.
12. OASIS. *Web Services Business Process Execution Language Version 2.0, Working Draft.* [http://www.oasis-open.org/committees/download.php/10347/wsbpel-specification-draft-120204.htm].
13. Sun microsystems. *Java Web Services Developer Pack.* [http://java.sun.com/ webservices/downloads/webservicespack.html].
14. S. Thatte. XLANG: Web Services for Business Process Design. [http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm], Microsoft Corporation, 2001.
15. W3C member submission 10 august, 2004. *Web Services Addressing.* [http://www.w3.org/submission/ws-addressing/].

16. World Wide Web Consortium. *SOAP Version 1.2 Part 1: Messaging Framework*. [http://www.w3.org/TR/soap12-part1/].
17. World Wide Web Consortium. *Web Services Choreography Description Language Version 1.0. Working draft 17 December 2004*. [http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/].
18. World Wide Web Consortium. *Web Services Description Language (WSDL) 1.1*. [http://www.w3.org/TR/wsdl].

A Syntax of χ and Satisfaction Relation for $\vdash$

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 $x = \perp$.