SOCK: a calculus for service oriented computing*

Claudio Guidi, Roberto Lucchi, Roberto Gorrieri,
Nadia Busi, and Gianluigi Zavattaro

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

Abstract. Service oriented computing is an emerging paradigm for designing distributed applications where *service* and *composition* are the main concepts it is based upon. In this paper we propose SOCK, a three-layered calculus equipped with a formal semantics, for addressing all the basic mechanisms of service communication and composition. The main contribute of our work is the development of a formal framework where the service design is decomposed into three fundamental parts: the behaviour, the declaration and the composition where each part can be designed independently of the other ones.

1 Introduction

Service oriented computing (SOC) is an emerging paradigm for designing distributed applications where *service* and *composition* are the main concepts it is based upon. A service can be seen as an application which performs a certain task when it is invoked. A composition of services can be seen as a group of services that, by means of collaborating message exchanges, fulfills a more complex task than those performed by the single services it is composed of. The key fact is that a suitable *composition of services is a service*. The most credited service oriented technology is the Web Services. A lot of industries and consortia in the world like Microsoft, IBM, W3C, OASIS (just to mention a few) have developed standards which define Web Services interfaces such as WSDL [Wor] and composition languages such as WS-BPEL [OAS]. Such a kind of languages are based on XML and are not equipped of a formal semantics.

In this paper, we propose SOCK, Service Oriented Computing Kernel, which is a process calculus equipped with a formal semantics, for addressing all the basic mechanisms of service communication and composition that takes inspiration from Web Services specifications. Our approach aims at dealing with different service features by considering them separately and in an orthogonal way. We distinguish among *service behaviour*, *service declaration*, *service engine* and *services system*. The service behaviour deals with the internal behaviour of the service and communication primitives, the service declaration is a description of the service deployment, the service engine is the execution environment where

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

the service is actually deployed and the services system is the composition of service engines. The main contribute of our work is the development of a formal framework where the service design is decomposed into three fundamental parts: the behaviour, the declaration and the composition. Each part can be designed independently of the other ones. We present a three-layered calculus which is an evolution of those proposed in our previous work [BGG+05,BGG+06,GL06]. It is formed by three calculi: the *service behaviour calculus*, the *service engine calculus* and the *services system calculus*. The former allows for the design of service behaviours by supplying computation and external communication primitives inpired to Web Services operations and workflow operators such as sequence, parallel and choice. The second calculus is built on top of the former and it allows for the specification of the service declaration where it is possible to design in an orthogonal way three main characteristics: *execution modality*, *persistent state* flag and *correlation sets*. The execution modality deals with the possibility to execute a service in a sequential order or in a concurrent one; the persistent state flag allows to declare if each session (of the service engine) has its own independent state or if the state is shared among all the sessions of the same service engine; the correlation sets is the mechanism for distinguishing sessions initiated by different dialoguers by means of the values received within some specified variables. Finally, the services system calculus allows for the definition of the whole system including all the involved services that interact each others.

The paper is organized as follows. In Section 2 we present the SOCK calculi: the service behaviour calculus, the service engine calculus and the services system one. In Section 3 conclusions and future research are reported.

2 SOCK

In this section we present calculi of SOCK for representing service behaviours, service engines and services system. They are an extension of those presented in our previous work [BGG+05,BGG+06] even if, for the sake of brevity, here we do not model asynchronous communications[1]. The semantics of the calculi are defined in terms of labelled transition systems (lts for short) [Kel76] and they are organized as follows. There are five lts layers: the *service behaviour lts layer*; the *service engine state lts layer*; the *service engine correlation lts layer*; the *service engine execution modality lts layer* and the *services system lts layer*. The first layer is the lower one. Each lts layer catches the actions raised by the underlying one and will enable or disable them. If an action is enabled by an lts layer it will be raised to the overlying one. The service behaviour lts layer describes all the possible execution paths generated by a session behaviour. The service engine state lts layer defines the rule for joining a service behaviour with a service engine local state. The service engine correlation lts layer deals with correlation set mechanism and the service engine execution modality lts layer represents rules for executing sessions concurrently or in a sequential order. Finally, the

[1] The reader interested to asynchoronous modelling in this setting may consult [BGG+06].

services system lts layer deals with a composed service engine system. A SOCK example can be found in [GLZ+].

2.1 Service behaviour calculus

Before introducing the calculus we discuss some important issues such as the external input and output actions and the service locations. The external input and output actions deal with those actions that are exploited by the service behaviour for communicating with other services. The primitives related to such a kind of actions are called *operations* and they are inspired to those of Web Services. Each operation is described by a name and an *interaction modality*. There are four kinds of peer-to-peer interaction modalities 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 input operations are published by a service behaviour in order to receive messages on them. The output operation, on the contrary, are exploited for sending messages to the input ones exhibited by the service behaviour to invoke. Here we group the operations into *single message operations* and *double message operations*. The former ones deal with the One-Way and the Notification operations whereas the latter with the Request-Response and the Solicit-Response ones. Let $\mathcal{O}$ and $\mathcal{O}_R$ be two disjoint sets of operation names where the former represents the single message operation names and the latter the double message ones. Let $Sup = \{(o, ow) \mid o \in \mathcal{O}\} \cup \{(o_r, rr) \mid o_r \in \mathcal{O}_R\}$ 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 $Op = Sup \cup Inv$ be the set of all the possible operations. In our framework, locations represent the address (let it be a logical or a physical one) where a service is located. In order to perform an output operation it is fundamental to explicit both the operation name and the location of the receiver in order to achieve a correct message delivery. In the following, locations will appear into the output operation primitives of the service behaviour calculus and, since they deal with the external communication, they will be exploited into the services system calculus for synchronizing external inputs with the corresponding output ones. Formally, let Loc be a finite set of location names ranged over by l.

The syntax. In the following we present the syntax of the calculus devoted to represent services. Let $Signals$ be a set of signal names exploited for synchronizing processes in parallel within a service behaviour. Let Var be a set of variables ranged over by x, y, z and Val, ranged over by v, be a generic set of values on which it is defined a total order relation. We exploit the notations $\boldsymbol{x} = \langle x_0, x_1, ..., x_i \rangle$ and $\boldsymbol{v} = \langle v_0, v_1, ..., v_i \rangle$ for representing tuples of variables and values respectively. Let k range over $Var \cup Loc$ where $Var \cap Loc = \emptyset$. The syntax follows:

$P, Q ::= \mathbf{0} \mid \bar{\epsilon} \mid \epsilon \mid x := e \mid \chi ? P : Q \mid P; P \mid P | P \mid \sum_{i \in W}^{+} \epsilon_i; P_i \mid \chi \rightleftharpoons P$

$\epsilon ::= s \mid o(\boldsymbol{x}) \mid o_r(\boldsymbol{x}, \boldsymbol{y}, P)$

$\bar{\epsilon} ::= \bar{s} \mid \bar{o}@k(\boldsymbol{x}) \mid \overline{o_r}@k(\boldsymbol{x}, \boldsymbol{y})$

We denote with SC the set of all possible processes ranged over by P and Q. $\mathbf{0}$ is the null process. Outputs can be a signal $\bar{s}$, a notification $\bar{o}@k(\boldsymbol{x})$ or a solicit-response $\overline{o_r}@k(\boldsymbol{x}, \boldsymbol{y})$ where $s \in Signals$, $o \in \mathcal{O}$ and $o_r \in \mathcal{O}_R$, $k \in Var \cup Loc$ represents the receiver location which can be explicit or represented by a variable[2], $\boldsymbol{x}$ is the tuple of the variables which store the information to send and $\boldsymbol{y}$ is the tuple of variables where, in the case of the solicit-response, the received information will be stored. Dually, inputs can be an input signal s, a one-way $o(\boldsymbol{x})$ or a request-response $o_r(\boldsymbol{x}, \boldsymbol{y}, P)$ where $s \in Signals$, $o \in \mathcal{O}$ and $o_r \in \mathcal{O}_R$, $\boldsymbol{x}$ is the tuple of variables where the received information are stored whereas $\boldsymbol{y}$ is the tuple of variables which contain the information to send in the case of the request-response; finally P is the process that has to be executed between the request and the response. $x := e$ assigns the result of the expression e to the variable x. For the sake of brevity, we do not present the syntax for representing expressions, we assume that they include all the arithmetic operators, values in Val and variables. $\chi ? P : Q$ is the *if-then-else* process, where χ is a logic condition on variables whose syntax is: $\chi ::= x \leq e \mid e \leq x \mid \neg\chi \mid \chi \wedge \chi$. It is worth noting that conditions such as $x = v$, $x \neq v$ and $v_1 \leq x \leq v_2$ can be defined as abbreviations; P is executed only if the condition χ is satisfied, otherwise Q is executed. $P; P$ and $P \mid P$ represent sequential and parallel composition respectively whereas $\sum_{i \in W}^{+} \epsilon_i; P_i$ is the 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 behaviour. Our calculus indeed aims at supplying a basic language for designing service behaviours where designers have a full control of the internal machinery and the only non-predictable choices are those driven by the external message reception. Finally, $\chi \rightleftharpoons P$ is the construct to model guarded iterations. As far as the semantics is concerned, here we consider the extension of SC which includes also the the terms $\bar{o_r}@l(\boldsymbol{x}), \bar{o_r}@z(\boldsymbol{x})$ and $o_r(\boldsymbol{x})$. These terms allows us to reduce the semantics rules for the Request-Response message exchange mechanism. In the semantics indeed, we will consider the response message as a One-Way message exchange as well. It is worth noting that the service behaviour calculus does

[2] It is worth noting that the receiver location is contained within the variable z in order to fullfil location mobility. The reader interested to this topic may consult [GL06].

not deal with the actual values of variables and locations but it models all the possible execution paths for all the possible variable values and locations. The semantics follows this idea by means of an infinite set of actions where external inputs, external outputs and assignment actions report all the value substitutions for both variables and locations except the actions $\bar{o}@l(\boldsymbol{v}/\boldsymbol{x}), \bar{o_r}@l(\boldsymbol{v}/\boldsymbol{x})$ and $\overline{o_r}@l(\boldsymbol{v}/\boldsymbol{x}, \boldsymbol{y})$ where locations are defined. Formally, let ω range over $\mathcal{O} \cup \mathcal{O}_R$ and let Act be the set of actions, ranged over by γ, defined as follows:
$Act = In \cup Out \cup Internal$
$In = \{\omega(\boldsymbol{v}/\boldsymbol{x}), o_r(\boldsymbol{v}/\boldsymbol{x}, \boldsymbol{y}, P)@l\}$
$Out = \{\bar{\omega}@l/z(\boldsymbol{v}/\boldsymbol{x}), \bar{\omega}@l(\boldsymbol{v}/\boldsymbol{x}), \overline{o_r}@l/z(\boldsymbol{v}/\boldsymbol{x}, \boldsymbol{y}), \overline{o_r}@l(\boldsymbol{v}/\boldsymbol{x}, \boldsymbol{y})\}$
$Internal = \{s, \bar{s}, x := v/e, \chi?, \neg\chi?\}$
We define $\rightarrow \subseteq SC \times Act \times SC$ as the least relation which satisfies the axioms and rules of Table 1 and closed w.r.t. $\equiv$, where $\equiv$ is the least congruence relation satisfying the axioms at the end of Table 1. The rules are divided into axioms and rules for defining composition operators that are quite standard. Rules One-WayOutLoc and Req-OutLoc deals with output operations where the location l is explicit whereas rules One-WayOut and Req-Out deal with output operations where the location is represented by the variable z. Rule ReqIn produces a process which executes P and then performs a notification joined with the sender location l. It is worth noting that the actual location will be joined at the level of the services system lts layer where synchronizations among service engines are defined. Rule Synchro defines the synchronization between signals which allows us to exploit them for synchronizing parallel processes of the same service behaviour. Finally, rules Iteration and Not Iteration model iteration in a way which resembles that of imperative programming.

Definition 1. ***(Well-formedness)*** *Let $P \in SC$ be a service behaviour calculus process. Let Ψ the set of all the possible external input operation terms. We say that P is a well-formed process if:*
$\exists \gamma_1, ..., \gamma_n \in \Psi, \exists Q_1, ..., Q_n \in SC, P = \gamma_1; Q_1 + ... + \gamma_n; Q_n$
We denote the set of all the well-formed processes with the symbol X_{SC}.

Definition 2. *Let $P \in X_{SC}$, a trace γ^* of P is a session iff $P \xrightarrow{\gamma^*} \boldsymbol{0}$.*

The condition of Definition 1 states that a service behaviour process is well-formed if it is formed by a set of processes, composed by means of an alternative choice, which start with an external input operation. By defintion it follows that $X_{SC} \subseteq SC$.

2.2 Service engine calculus

This section is devoted to present the service engine calculus. Before presenting its syntax, we introduce some basic concepts such as *state*, *correlation sets* and *service declaration*. In a service engine indeed, all the executed sessions of a service behaviour are joined by a state and a correlation set. Furthermore, a service engine always executes sessions by following the specifications defined

(In) $s \xrightarrow{s} \mathbf{0}$ (Out) $\bar{s} \xrightarrow{\bar{s}} \mathbf{0}$ (One-WayOut) $\bar{\omega}@z(\boldsymbol{x}) \xrightarrow{\bar{\omega}@l/z(\boldsymbol{v}/\boldsymbol{x})} \mathbf{0}$ (One-WayOutLoc) $\bar{\omega}@l(\boldsymbol{x}) \xrightarrow{\bar{\omega}@l(\boldsymbol{v}/\boldsymbol{x})} \mathbf{0}$ (One-WayIn) $\omega(\boldsymbol{x}) \xrightarrow{\omega(\boldsymbol{v}/\boldsymbol{x})} \mathbf{0}$

(Assign) $x := e \xrightarrow{x:=v/e} \mathbf{0}$ (Req-Out) $\overline{o_r}@z(\boldsymbol{x},\boldsymbol{y}) \xrightarrow{\overline{o_r}@l/z(\boldsymbol{v}/\boldsymbol{x},\boldsymbol{y})} o_r(\boldsymbol{y})$ (Req-OutLoc) $\overline{o_r}@l(\boldsymbol{x},\boldsymbol{y}) \xrightarrow{\overline{o_r}@l(\boldsymbol{v}/\boldsymbol{x},\boldsymbol{y})} o_r(\boldsymbol{y})$

(Req-In) $o_r(\boldsymbol{x},\boldsymbol{y},P) \xrightarrow{o_r(\boldsymbol{v}/\boldsymbol{x},\boldsymbol{y},P)@l} P;\bar{o_r}@l(\boldsymbol{y})$ (If then) $\chi?P:Q \xrightarrow{\chi?} P$ (Else) $\chi?P:Q \xrightarrow{\neg\chi?} Q$

(Iteration) $\chi \rightleftharpoons P \xrightarrow{\chi?} P;\chi \rightleftharpoons P$ (Not Iteration) $\chi \rightleftharpoons P \xrightarrow{\neg\chi?} \mathbf{0}$ (Synchro) $\dfrac{P \xrightarrow{s} P', Q \xrightarrow{\bar{s}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$

(Sequence) $\dfrac{P \xrightarrow{\gamma} P'}{P;Q \xrightarrow{\gamma} P';Q}$ (Parallel) $\dfrac{P \xrightarrow{\gamma} P'}{P \mid Q \xrightarrow{\gamma} P' \mid Q}$ (Choice) $\dfrac{\epsilon_i \xrightarrow{\gamma} \mathbf{0} \quad i \in I}{\sum^{+}_{i\in I} \epsilon_i; P_i \xrightarrow{\gamma} P_i}$

Structural Congruence

$$P \mid Q \equiv Q \mid P \qquad P \mid \mathbf{0} \equiv \mathbf{0} \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad \mathbf{0};P \equiv P$$

Table 1. Rules for service behaviour lts layer

within the service declaration. A state is represented 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 it is defined a total order relation[3]. We use $\mathcal{S}[v/x]$, whose definition follows, to denote the variable state update:

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

Conditions can be evaluated over states. We exploit the notation $\mathcal{S} \vdash \chi$ for denoting that the state $\mathcal{S}$ satisfies the condition χ. The satisfaction relation for $\vdash$ is defined by the following rules, where e denotes an expression and $e \hookrightarrow_{\mathcal{S}} v$ denotes that, when the state is $\mathcal{S}$, the expression e is evaluated into the value v or, when some variables within the expression are not instantiated, into the symbol $\perp$:
1. $e \hookrightarrow_{\mathcal{S}} v, \mathcal{S}(x) \leq v \Rightarrow \mathcal{S} \vdash x \leq e$ 2. $e \hookrightarrow_{\mathcal{S}} v, v \leq \mathcal{S}(x) \Rightarrow \mathcal{S} \vdash e \leq x$
3. $\mathcal{S} \vdash \chi' \wedge \mathcal{S} \vdash \chi'' \Rightarrow \mathcal{S} \vdash \chi' \wedge \chi''$ 4. $\neg(\mathcal{S} \vdash \chi) \Rightarrow \mathcal{S} \vdash \neg\chi$

Sessions often require to be distinguished and accessed only by those dialoguers which hold some specific references. In the object oriented paradigm such a ref-

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

erence is the object reference guaranteed by the object oriented framework. In service oriented computing in general, we cannot assume the existence of an underlying framework which guarantees references management. Correlation sets, which are a mechanism defined within WS-BPEL, allows us to address such an issue. A correlation set is a set of variables called *correlated variables*. Formally, let $CSet = \mathbf{P}(Var)$ be the set of all the correlation sets ranged over by c. In a service engine a session is identified by the values assigned to the correlated variables by the current state. The session identification issue is raised when a message is received on an input operation. Since there should be several sessions that are waiting on the same input operation indeed, the right session to which the message is delivered is identified by means of the correlated variables values. Given a variable x, two values v and w where the former is the value which is willing for being replaced within the variable x and the latter is the actual value of x and a correlation set c, we say that v is correlated to x coherently with c, $v/x \vdash_c w$, if: the variable x belongs to c and its actual value is $w = v$;the variable x belongs to c and its actual value is $w = \bot$; the variable x does not belong to c. Formally we exploit the following notation:

$v/x \vdash_c w \iff (x \in c \wedge (v = w \vee w = \bot)) \vee x \notin c$

We extend such a definition to vector of variables and values:

$\boldsymbol{v}/\boldsymbol{x} \vdash_c \boldsymbol{w} \iff \forall x_i, v_i/x_i \vdash_c w_i$

As far as the service declaration is concerned, it contains all the necessary information for executing sessions. In particular, it specifies: the service behaviour whose sessions are executed by the service engine; if each session has its own state or if there is a common state shared by all the sessions (in the former case the state is renewed each time the execution of a session starts and it expires when the session terminates: we say that the state is *not persistent* whereas in the latter case, the state is never renewed and the variables hold their values after the termination of the sessions: we say that the state is *persistent*); the correlation set which guards the executed sessions; if the sessions are executed in a sequential order or in a concurrent one. The syntax follows:

$U ::= P_{\times} \mid P_{\bullet} \qquad W ::= c \triangleright U \qquad D ::= !W \mid W^*$

where $P \in X_{SC}$ is a service behaviour, flag $\times$ denotes that P is equipped with a not persistent state and flag $\bullet$ denotes that P is equipped with a persistent one. c is the correlation set which guards the execution of the sessions, $!W$ denotes a concurrent execution of the sessions and W^* denotes the fact that sessions are executed in a sequential order. D is a service declaration.

The service engine calculus syntax Here we present the service engine calculus syntax:

$$Y ::= D[H] \qquad H ::= c \triangleright P_S \qquad P_S ::= (P, \mathcal{S}) \mid P_S \mid P_S$$

where D is a service declaration, P is a service behaviour process and $\mathcal{S}$ is a state. Y is a service engine and it is composed of a service declaration D and an execution environment H. H represents the actual sessions which are running on the service engine modelled as the parallel composition of service behaviour process coupled with a state $(P, \mathcal{S})$. All the couples are guarded by the same

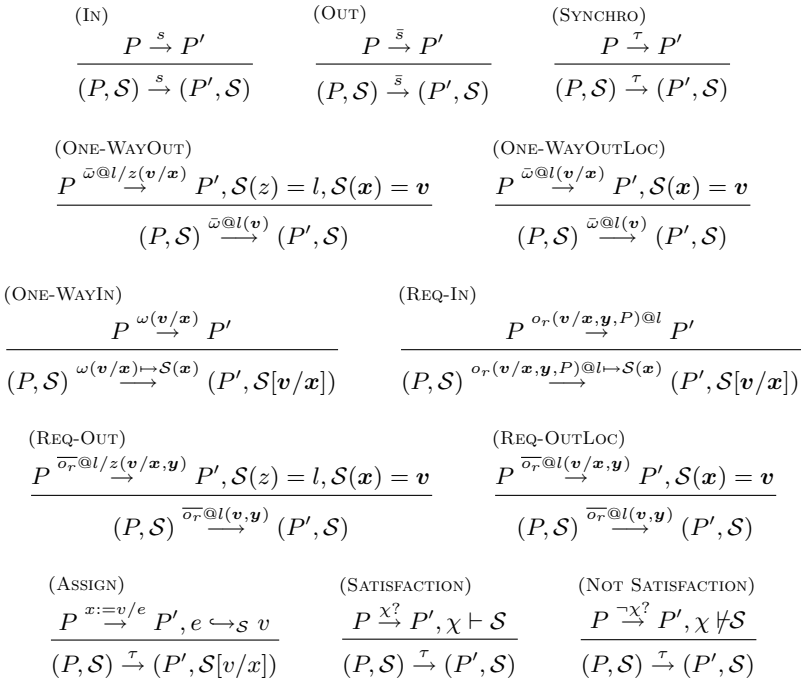

Table 2. Rules for service engine state lts layer

correlation set c; it is worth noting that a service engine is not correlated when $c = \emptyset$. We denote with HC the set of all the possible processes ranged over by Y. The semantics is defined in terms of different label transition system layers presented in Tables 2,4 and 5. Table 2 deals with the rules for the service engine state lts layer which defines the semantics for a couple of a service behaviour process and a state, Table 4 deals with the rules for service engine correlation lts layer where it is defined the semantics for managing correlation sets and Table 4 deals with the service engine execution modality lts layer which defines the semantics for executing sessions in a concurrent or in a sequential way. As far as Table 2 is concerned, an action is enabled if the current state contains variables values which correspond to those reported into the action. An action is disabled, i.e. it is not raised to the overlying lts layer, when the variables values into the state do not correspond to those reported into the action. The enabled action, when raised to the overlying lts layer, will be modified in order to forward only the needed information. Table 3 reports the one-to-one mapping from the service behaviour lts layer actions to the service engine state lts layer ones. Actions a), b) are not altered since they do not deal with the state whereas actions g) are replaced with a τ action because they do not carry any information needed by

the overlying layer. Actions c) and e) are related to the output operations and, when enabled, they resolve the values of the variables and locations. Indeed, if we consider rules One-WayOut and Req-Out they contain the conditions $\mathcal{S}(z) = l$ and $\mathcal{S}(\boldsymbol{x}) = \boldsymbol{v}$ which allows for the verification of the actual values of the variables within the current state. In particular, rules One-WayOutLoc and Req-OutLoc do not resolve locations because they are explicitly represented. Actions d) and f) deal with the input operations (rules One-WayIn, Req-In) and, when enabled, they do not resolve the values of the variables but they forward the actual values of the variables involved into the action ($\mapsto \mathcal{S}(\boldsymbol{x})$). This is due to the fact that some variables could be correlated and it will be necessary to verify if they satisfy the current correlation set. Such a control will be done in the overlying layer whose rules, closed w.r.t. the structural congruence, are reported in Table 4. Also in this case the actions, if enabled, will be modified and raised for the overlying layer[4]. In Table 4 rules CorrelatedOne-WayIn and CorrelatedReq-In deal with the actions d) and f) of Table 3 where the values of the variables are resolved only if the condition on correlation set is satisfied ($\boldsymbol{v}/\boldsymbol{x} \vdash_c \boldsymbol{w}$). As far as Table 5 is concerned, the actions are enabled at the level of the service engine (rule Execution) and session execution modalities, concurrent or sequential with a persistent or a not persistent state, are defined (rules ConcurrentNotPersistent, ConcurrentPersistent, SequentialNotPersistent and SequentialPersistent). Rule ConcurrentNotPersistent deals with a concurrent execution of the sessions and with a not persistent state. In particular, each session has its own state, that is initially fresh ($\mathcal{S}_\perp$), and it is executed concurrently with the other ones. It is worth noting that condition $\nexists \quad \mathcal{S}_i \in P_S \; c \triangleright (P, \mathcal{S}_i) \xrightarrow{\gamma} c \triangleright (P', \mathcal{S}_i')$[5] states that it is not possible to start a new session with a set of values for the correlated variables that belong to another running session. Rule ConcurrentPersistent deals with the concurrent execution of sessions which share a common state. Rule SequentialNotPersistent deals with the sequential execution of sessions which have their own state. In this case there is always no more than one executed session at a time. The state is not persistent and it is renewed each time a new session is spawned. Finally, rule SequentialPersistent deals with the sequential execution of the sessions where the state is shared and it does not expire after session termination.

2.3 Services system calculus

Here we present the services system calculus which is based on the service engine one and it allows for the composition of different engines into a system. The service engines are composed in parallel and they are equipped with a location that allows us to univocally distinguish them within the system. The calculus

[4] For the sake of brevity, we do not report the mapping table for the actions. It is easy to extract it from the rules of Table 4.

[5] We abuse of the notation $\mathcal{S}_i \in P_S$ for meaning that it exists a couple $(P, \mathcal{S}_i)$ within the term P_S.

	Service behaviour actions	Service engine state actions
a)	s	s
b)	$\overline{s}$	$\overline{s}$
c)	$\bar{\omega}@l/z(\boldsymbol{v}/\boldsymbol{x}), \bar{\omega}@l(\boldsymbol{v}/\boldsymbol{x})$	$\bar{\omega}@l(\boldsymbol{v})$
d)	$\omega(\boldsymbol{v}/\boldsymbol{x})$	$\omega(\boldsymbol{v}/\boldsymbol{x}) \mapsto \mathcal{S}(\boldsymbol{x})$
e)	$\overline{o_r}@l/z(\boldsymbol{v}/\boldsymbol{x},\boldsymbol{y}), \overline{o_r}@l(\boldsymbol{v}/\boldsymbol{x},\boldsymbol{y})$	$\overline{o_r}@l(\boldsymbol{v},\boldsymbol{y})$
f)	$o_r(\boldsymbol{v}/\boldsymbol{x},\boldsymbol{y},P)@l$	$o_r(\boldsymbol{v}/\boldsymbol{x},\boldsymbol{y},P)@l \mapsto \mathcal{S}(\boldsymbol{x})$
g)	$\chi?, \neg\chi?, x := v/e, \tau$	τ

Table 3. Enabled action mapping

(Not Correlated)
$$\frac{P_S \xrightarrow{\gamma} P'_S}{c \triangleright P_S \xrightarrow{\gamma} c \triangleright P'_S} \gamma \neq \begin{cases} o(\boldsymbol{v}/\boldsymbol{x}) \mapsto \boldsymbol{w} \\ o_r(\boldsymbol{v}/\boldsymbol{x},\boldsymbol{y},P)@l \mapsto \boldsymbol{w} \end{cases}$$

(Parallel)
$$\frac{c \triangleright P_S \xrightarrow{\gamma} c \triangleright P'_S}{c \triangleright P_S \mid Q_S \xrightarrow{\gamma} c \triangleright P'_S \mid Q_S}$$

(CorrelatedOne-WayIn)
$$\frac{P_S \xrightarrow{\omega(\boldsymbol{v}/\boldsymbol{x})\mapsto(\boldsymbol{w})} P'_S, \boldsymbol{v}/\boldsymbol{x} \vdash_c \boldsymbol{w}}{c \triangleright P_S \xrightarrow{\omega(\boldsymbol{v})} c \triangleright P'_S}$$

(CorrelatedReq-In)
$$\frac{P_S \xrightarrow{o_r(\boldsymbol{v}/\boldsymbol{x},\boldsymbol{y},P)@l\mapsto(\boldsymbol{w})} P'_S, \boldsymbol{v}/\boldsymbol{x} \vdash_c \boldsymbol{w}}{c \triangleright P_S \xrightarrow{o_r(\boldsymbol{v},\boldsymbol{y})@l} c \triangleright P'_S}$$

Structural Congruence

$$P_S \mid Q_S \equiv Q_S \mid P_S \qquad P_S \mid (Q_S \mid R_S) \equiv (P_S \mid Q_S) \mid R_S \qquad P_S \mid (\mathbf{0}, \mathcal{S}) \equiv P_S$$

Table 4. Rules for service engine correlation lts layer

(ConcurrentNotPersistent)
$$\frac{(P, \mathcal{S}_\perp) \xrightarrow{\gamma} (P', \mathcal{S}'), \nexists\ \mathcal{S}_i \in P_S\ c \triangleright (P, \mathcal{S}_i) \xrightarrow{\gamma} c \triangleright (P', \mathcal{S}'_i)}{!c \triangleright P_\times[c \triangleright P_S] \xrightarrow{\gamma} !c \triangleright P_\times[c \triangleright P_S \mid (P', \mathcal{S}')]}$$

(ConcurrentPersistent)
$$\frac{c \triangleright (P, \mathcal{S}) \xrightarrow{\gamma} c \triangleright (P', \mathcal{S}')}{!c \triangleright P_\bullet[c \triangleright (Q, \mathcal{S})] \xrightarrow{\gamma} !c \triangleright P_\bullet[c \triangleright (Q \mid P', \mathcal{S}')]}$$

(SequentialNotPersistent)
$$\frac{c \triangleright (P, \mathcal{S}_\perp) \xrightarrow{\gamma} c \triangleright (P', \mathcal{S}')}{(c \triangleright P_\times)^*[c \triangleright (\mathbf{0}, \mathcal{S}'')] \xrightarrow{\gamma} (c \triangleright P)^*_\times[c \triangleright (P', \mathcal{S}')]}$$

(SequentialPersistent)
$$\frac{c \triangleright (P, \mathcal{S}) \xrightarrow{\gamma} c \triangleright (P', \mathcal{S}')}{(c \triangleright P_\bullet)^*[c \triangleright (\mathbf{0}, \mathcal{S})] \xrightarrow{\gamma} (c \triangleright P)^*_\bullet[c \triangleright (P', \mathcal{S}')]}$$

(Execution)
$$\frac{H \xrightarrow{\gamma} H'}{D[H] \xrightarrow{\gamma} D[H']}$$

Table 5. Rules for service engine execution modality lts layer

syntax follows:

$E ::= Y_l \mid E \parallel E$

A service engine system E can be a located service engine Y_l, where l is a location, or a parallel composition of them. The semantics is defined in terms of a labelled transition system whose rules are described in Table 6 and closed w.r.t. the structural congruence. At the level of services system there are only two kinds of action label: τ and $\tilde{\tau}$ where the former represents synchronizations among service engines and the latter represents not observable internal actions of service engines (rule INTERNAL). Rules ONE-WAYSYNC and REQ-SYNC describe synchronizations among different service engines. The former models a One-Way message exchange and the latter models the request message exchange in the case of a Request-Response. It is worth noting that the response message exchange, in the case of a Request-Response, is modelled by the former rule indeed, by means of rules of Table 1, Request-Response operations can be externally seen as two One-Ways.

$$\text{(ONE-WAYSYNC)}\quad \frac{Y_l \xrightarrow{\bar{\omega}@l'(\boldsymbol{v})} Y_l' \ , \ Z_{l'} \xrightarrow{\omega(\boldsymbol{v})} Z_{l'}'}{Y_l \parallel Z_{l'} \xrightarrow{\tau} Y_l' \parallel Z_{l'}'} \qquad \text{(REQ-SYNC)}\quad \frac{Y_l \xrightarrow{\overline{o_r}@l'(\boldsymbol{v},\boldsymbol{y})} Y_l' \ , \ Z_{l'} \xrightarrow{o_r(\boldsymbol{v},\boldsymbol{y})@l} Z_{l'}'}{Y_l \parallel Z_{l'} \xrightarrow{\tau} Y_l' \parallel Z_{l'}'}$$

$$\text{(PAR-EXT)}\quad \frac{E_1 \xrightarrow{\gamma} E_1'}{E_1 \parallel E_2 \xrightarrow{\gamma} E_1' \parallel E_2} \qquad \text{(INTERNAL)}\quad \frac{Y_l \xrightarrow{\gamma} Y_l'}{Y_l \xrightarrow{\tilde{\tau}} Y_l'} \ \gamma \neq \begin{cases} \bar{o}@l(\boldsymbol{v}) \\ o(\boldsymbol{v}) \\ \overline{o_r}@l(\boldsymbol{v},\boldsymbol{y}) \\ o_r(\boldsymbol{v},\boldsymbol{y})@l \end{cases}$$

(STRUCTURAL CONGRUENCE OVER E)

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

Table 6. Rules for services system lts layer

3 Conclusion

In this paper we have proposed a set of process calculi for dealing with service design and composition. There are other works which exploit formal models for representing services and service composition. In general, they use different models for representing service behaviours and service composition and they do not deal with service deployment features. In [DD04] the authors use Petri Nets for describing service behaviours but they focus only on workflow aspects without distinguishing among the different kind of operations. In [LM] a semantics of WS-BPEL is defined in terms of pi-calculus processes but correlation sets are not considered. In [MC06] the authors present a language, called *Orc*, where services

are considered as functions and a service invocation is expressed by a function call. Finally, as far as correlation sets are concerned, in [Vir04] Viroli propose a first formalization of the mechanism specified within BPEL4WS specification.

Our work must be considered within a wider framework context we are working on where we have analyzed orchestration and choreography calculi for addressing system design issues. The relationship between the two views has been given by exploiting a notion of conformance based on bisimulation. Jointly with the formal investigation, we are developing a Java interpreter for the orchestration language called JOLIE (Java Orchestration Language Interpreter Engine) [MGLZ]. At the present, JOLIE is able to intepret the service behaviour calculus and it allows us to compose different JOLIE services over the Internet. In the future, we intend to extend it in order to interpret the service engine calculus.

References

[BGG+05] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration: A synergic approach for system design. In *ICSOC'05*, volume 3826 of *LNCS*, pages 228–240, 2005.

[BGG+06] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *Proc. of 8th International Conference on Coordination Models and Languages (COORDINATION'06)*, volume 4038 of *LNCS*, pages 63–81, 2006.

[DD04] R. Dijkman and M. Dumas. Service-oriented Design: a Multi-viewpoint Approach. *Int. J. Cooperative Inf. Syst.*, 13(4):337–368, 2004.

[GL06] C. Guidi and R. Lucchi. Mobility mechanisms in service oriented computing. In *Proc. of 8th International Conference on on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *LNCS*, pages 233–250, 2006.

[GLZ+] C. Guidi, R. Lucchi, G. Zavattaro, N. Busi, and R. Gorrieri. Technical Report UBLCS-2006-20, Dep. of Computer Science, Univ. of Bologna [http://www.cs.unibo.it/research/reports/], 2006.

[Kel76] R. M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976.

[LM] R. Lucchi and M. Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*. Elsevier Press. To appear.

[MC06] J. Misra and W. Cook. Computation orchestration, a basis for wide-area computing. *Journal of Software and Systems modeling*, 2006. To appear.

[MGLZ] F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. JOLIE: a Java Orchestration Language Interpreter Engine. In *CoOrg06*, volume to appear of *ENTCS*.

[OAS] 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].

[Vir04] M. Viroli. Towards a Formal Foundation to Orchestration Languages. In *Proc. of 1st International Workshop on Web Services and Formal Methods (WS-FM 2004)*, volume 105 of *ENTCS*. Elsevier, 2004.

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