Service Oriented Computing from a Process Algebraic Perspective ⋆

Mario Bravetti Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università degli Studi di Bologna, Mura Anteo Zamboni 7, I-40127 Bologna, Italy.
e-mail:{bravetti,zavattar}@cs.unibo.it

Abstract

Service Oriented Computing is emerging as a reference model for a new class of distributed computing technologies such as Web Services and the Grid. We discuss three main aspects of Service Oriented Computing (loose coupling, communication latency, and open endedness), and we relate them with traditional process algebra operators. We also indicate some new issues, raising from the combination of these three aspects, that require the investigation of suitable new process algebra operators.

1 Introduction

Service Oriented Computing is an emerging paradigm for distributed computing based on services as the basic computational entities. Services are autonomous, platform-independent, heterogeneous elements that interact via basic patterns of service invocation. The main novelty of service oriented computing, with respect to traditional distributed computing models, is that services are stateless and all information they need is usually passed within the exchanged messages. This technique is called contextualization because the messages contain additional context information, such as cookies or session identifiers, used to describe the state of the overall computation. Due to the statelessness assumption, the service oriented paradigm is particularly suited to program systems based on a minimal shared knowledge and understanding among the interacting parts. These systems are usually referred to as loosely coupled systems.

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

The most prominent service oriented technologies are Web Services and the Grid. These technologies are based on standardized mechanisms used to describe the interface of the services, to advertise and locate new services, and to invoke the available services via one of the basic interaction pattern. Complex service interactions, which cannot be trivially encoded in the basic patterns, require a so-called service orchestration. Service orchestration is usually achieved adding new components (called the orchestrators) that do not actually perform computation, but simply manage the flow of invocation of the services involved in the collaboration. A plethora of languages (comprising e.g. XLANG, BizTalk, WSFL, WS-BPEL) has been recently defined to specify and program orchestrators. All these languages combine workflow constructs and communication primitives. The workflow constructs are used to decribe the flow of execution of the orchestration activities, while the communication primitives correspond to the basic service interaction patterns. Most of these languages have explicitly taken inspiration from process algebras such as CSP or the π-calculus. Nevertheless, due to peculiarities of service oriented computing, some constructs and primitives differ from the traditional operators of process algebras. Three of these peculiarities are:

Loose coupling. Orchestrators have a minimal control on the orchestrated services, for instance, a service can autonomously exit the orchestration without any previous notice.

Communication latency. The transport layer responsible for the exchange of messages, among the orchestrator and the services, does not give guarantees about the reliability and timing of remote message delivery.

Open-endedness. An orchestrator can dynamically, i.e. at run time, retrieve new services to be involved in the orchestration; for instance, this could be useful to replace services that autonomously leave the orchestration.

The remainder of the paper is organized as follows: in Sections 2, 3, and 4 we focus separately on the three above aspects, while in Section 5 we conclude discussing interesting issues raised by their combination.

2 Loose coupling

In order to cope with loose coupling, orchestration languages usually provide linguistic constructs to program the so-called loosely coupled transactions. Traditional database transactions guarantees the ACID properties: Atomicity, Consistency, Isolation, and Durability. When the activities involved in a transaction are loosely coupled the ACID properties adapt badly. In particular, Isolation usually requires to lock resources. In Web Services applications, for instance, resources may belong to different companies and there is no chance for an orchestrator to lock resources of other companies. Additionally, trans-

actions may last long periods of time, and it is not feasible to block resources so long.

The loosely coupled transactions weaken the notion of rollback: a service might decide that rollback will not cancel all the activities carried out. The cancellation of an airplane booking, for instance, may lead to the payment of a fee. Services that do not support an "absolute" mechanism of rollback, make failures extremely complicated, to be dealt with ad-hoc rollbacks. These ad-hoc rollback processes are called compensations.

The notion of compensation is the key aspect of several recent processs algebras defined on purpose to formalize the semantics of compensation execution, and to reason about properties of compensation policies. The first proposal in this direction is StAC, a calculus with an explicit compensation operator whose operational semantics has been formalized in [8]. StAC has recently inspired also a new CSP dialect, called cCSP [9], whose semantics is defined denotationally in terms of traces. An alternative proposal is represented by the SAGAS calculi [7] that defines a concurrent big-step semantics for sequential, parallel, and nested compensatable transactions. Recently, in [6] cCSP and the SAGAS calculi have been thoroughly compared discussing how to encode (fragments of) the former in (some of) the latter calculi, and vice versa. Compensations have been formalized and investigated also in the context of π-calculus in [4], where a calculus inspired by the compensation policy of BizTalk is presented.

In this Section we focus on the basic mechanisms required to run compensations. Compensations are usually activated in case an unexpected event occurs such as the reception of a negative response or the unavailability of a service. Usually, when one of these events occurs, some activity must be interrupted (because it has failed) and some other should be activated instead (responsible for executing the ad-hoc rollback procedure). In order to investigate these aspects formally, we define a process calculus comprising a new operator that combines the possibility to interrupt a process with the possibility to activate an alternative compensating process. A similar operator has been already investigated in [13] in the context of the asynchronous π-calculus. However, there are significant differences with that paper. First of all, in this paper interrupt signals can be produced only internally from the activity to be interrupted. Moreover, in case an interruptable activity contains other nested interruptable activities, the inner ones are blocked in case of interruption of the outer one; this was not the case in [13] where inner transactions and messages are not involved in transaction abort operations.

In this paper we include the process interruption operator in a process calculus based on asynchronous shared dataspace communication: processes interact by producing tuples (i.e. ordered sequences of data) that are stored in a shared

repository called the tuple space where they can be subsequently retrieved (either read or consumed) by means of read or input coordination primitives. Shared dataspace communication has revealed a natural choice for modeling asynchronously interacting services. In particular, as will be made more clear in the next Section, we will be able to model in a rather simple manner networks of remotely interacting services exploiting the notion of distributed tuple spaces.

2.1 A Basic Calculus with Interruptable Processes

The coordination primitives that we consider to access the shared tuple space are: $out(e)$, $in(t)$ and $rd(t)$. The output operation $out(e)$ inserts a tuple e in the tuple space (TS for short). Primitive $in(t)$ is the blocking input operation: when an occurrence of a tuple e matching with t (denoting a template) is found in the TS, it is removed from the TS and the primitive returns the tuple. The read primitive $rd(t)$ is the blocking read operation that, differently from $in(t)$, returns the matching tuple e without removing it from the TS.

In languages based on shared tuple space communication, tuples are ordered and finite sequences of typed fields, while template are ordered and finite sequences of fields that can be either *actual* or *formal* (see [11]): a field is actual if it specifies a type and a value, whilst it is formal if the type only is given. For the sake of simplicity, in the formalization we are going to present, fields are not typed.

Formally, let $Mess$, ranged over by m, m', ..., be a denumerable set of messages and Var, ranged over by x, y, ..., be the set of data variables. In the following, we use $\vec{x}$, $\vec{y}$, ..., to denote finite sequences $x_1; x_2; \ldots; x_n$ of variables. We consider also expressions taken from a generic set Exp (ranged over by exp, exp', ...); expressions may contain variables and are equipped with an evaluation function $Eval : Exp \rightarrow Mess$.

Tuples, denoted by e, e', ..., are finite and ordered sequences of data fields (we use $arity(e)$ to denote the number of fields of e), whilst templates, denoted by t, t', ..., are finite and ordered sequences of fields that can be either data or wildcards (used to match with any message).

Formally, tuples are defined as follows:

$$e = \langle \vec{d} \rangle$$

where $\vec{d}$ is a term of the following grammar:

$\vec{d} ::= d \mid d; \vec{d}$
$d ::= m \mid x \mid exp.$

We overload the evaluation function and apply it also to tuples with the expected meaning; $Eval(e)$ returns the tuple obtained by evaluation of the fields of e that contain an expression.

The definition of template follows:

$$t = \langle \vec{dt} \rangle$$

where $\vec{dt}$ is a term of the following grammar:

$\vec{dt} ::= dt \mid dt; \vec{dt}$
$dt ::= d \mid null.$

A *data field* d can be a message or a variable or an expression. The additional value *null* denotes the wildcard, whose meaning is the same of formal fields, i.e. it matches with any field value. With abuse of notation we apply the *Eval* function also to templates. In the following, the set $Tuple$ (resp. $Template$) denotes the set of tuples (resp. templates) containing no variable and no expressions.

The matching rule between tuples and templates we consider is as follows.

Definition 2.1 ***Matching rule -*** *Let $e = \langle d_1; d_2; \ldots; d_n \rangle \in Tuple$ be a tuple, $t = \langle dt_1; dt_2; \ldots; dt_m \rangle \in Template$ be a template; we say that e matches t (denoted by $e \triangleright t$) if the following conditions hold:*

(1) $m = n$.
(2) $dt_i = d_i$ *or* $dt_i = null$, $1 \leq i \leq n$.

Condition 1. checks if e and t have the same arity, whilst 2. tests if each non-wildcard field of t is equal to the corresponding field of e.

Processes, denoted by P, Q, ..., are defined as follows:

$P, Q, \ldots ::=$

$commit$	commit command
$\mid$ $abort$	abort command
$\mid$ $out\ (e).P$	output
$\mid$ $rd\ t(\vec{x}).P$	read
$\mid$ $in\ t(\vec{x}).P$	input
$\mid$ $P \mid P$	parallel composition
$\mid$ $!P$	replication
$\mid$ $P \hookleftarrow P$	interruption

A process can complete its computation entering either in a commit or abort state; these two states are denoted by the *commit* and *abort* states, respectively. The other processes are prefix forms $\mu.P$, the parallel composition of two programs, the replication of a program, or an interruptable process $P \hookleftarrow Q$. The prefix μ can be one of the following coordination primitives: i) *out* (e), that writes the tuple e in the TS; ii) *rd* $t(\vec{x})$, that given a template t reads a matching tuple e in the TS and stores the return value in $\vec{x}$; iii) *in* $t(\vec{x})$, that given a template t consumes a matching tuple e in the TS and stores the return value in $\vec{x}$. In both the *rd* $t(\vec{x})$ and *in* $t(\vec{x})$ operations $(\vec{x})$ is a binder for the variables in $\vec{x}$. The parallel composition $P \mid Q$ of two processes P and Q behaves as two processes running in parallel. Infinite behaviours can be expressed using the replication operator $!P$. Replication is a typical operator used in process calculi to denote the parallel composition of an unbounded amount of instances of the same process. The last operator is used to program interruptable activities. In the term $P \hookleftarrow Q$ the process P executes its operation until a *abort* command is executed. After execution of the *abort* command the process Q is activated as interrupt handler.

To shorten the notation we usually omit trailing *commit*, e.g., we write the process $out(e).commit$ simply as $out(e)$.

In the following, $P[d/x]$ denotes the process that behaves as P in which all free occurrences of x (also inside expressions) are replaced with d. We also use $P[\vec{d}/\vec{x}]$ to denote the process obtained by replacing in P all occurrences of variables in $\vec{x}$ with the corresponding value in $\vec{d}$, i.e. $P[d_1; d_2; \ldots; d_n/x_1; x_2; \ldots; x_n] = P[d_1/x_1][d_2/x_2] \ldots [d_n/x_n]$.

We say that a process is *well formed* if each prefix of kind $rd/in\ \langle \vec{dt} \rangle (\vec{x})$ is such that the variables $\vec{x}$ and the data $\vec{dt}$ have the same arity. Notice that in the *rd* $t(\vec{x})$ and *in* $t(\vec{x})$ operations we explicitly indicate in $(\vec{x})$ the variables that will be bound to the actual fields of the matching tuple. Moreover, a well formed process is also *closed*: given an occurrence of an output primitive *out* $(e).P$, we assume that all the variables in e (also those occurring inside expressions) are included in the scope of a binder. In the following, we consider only processes that are well formed; $Process$ denotes the set of such processes.

Let $DSpace$, ranged over by DS, DS', ..., be the set of possible configurations of the TS, that is $DSpace = \mathcal{M}_{fin}(Tuple)$, where $\mathcal{M}_{fin}(S)$ denotes the set of all the possible finite multisets on S. In the following, we use $DS(e)$ to denote the number of occurrences of e within $DS \in DSpace$.The set $System = \{\langle P, DS \rangle \mid P \in Process, DS \in DSpace\}$, ranged over by s, s', ..., denotes the possible configurations of systems.

The semantics we use to describe processes interacting via coordination primitives is defined by means of a structural congruence relation $\equiv$ that equates

processes that we do not want to distiguish. $\equiv$ is defined to be the minimal congruence relation over processes such that

$$P|Q \equiv Q|P \qquad P|(Q|R) \equiv (P|Q)|R \qquad !P \equiv !P|P$$
$$P|commit \equiv P \qquad commit \hookleftarrow P \equiv commit$$

The equivalence relation $\equiv$ is extended to system configurations by means of the rule

$$\text{if } P \equiv Q \text{ then } \langle P, DS\rangle \equiv \langle Q, DS\rangle$$

Semantics of processes is defined in terms of a transition system over equivalence classes of system configurations, i.e. $[s]_{\equiv}$ for some $s \in System$. In the rest of this section we will use $[s]$ as a shorthand for $[s]_{\equiv}$ and $[P, DS]$ as a shorthand for $[\langle P, DS\rangle]_{\equiv}$. More precisely the operational semantics is defined to be $(System/\equiv, \longrightarrow)$, where: $System/\equiv$ is the set of states and $\longrightarrow \subseteq System/\equiv \times System/\equiv$ is the minimal relation satisfying the axioms and rules of Table 1. $([s], [s']) \in \longrightarrow$ (also denoted by $[s] \longrightarrow [s']$) means that a system (configuration) s can evolve (performing a single action) in the system (configuration) s'. When evaluating the semantics of a process P, we consider $[P, \emptyset] \in System/\equiv$ to be the initial state.

(1) $$\frac{Eval(e) = e'}{[out\ (e).P, DS] \longrightarrow [P, DS \oplus e']}$$

(2) $$\frac{Eval(t) = t' \quad \exists e \in DS : e \triangleright t'}{[in\ t(\vec{x}).P, DS] \longrightarrow [P[e/\vec{x}], DS - e]}$$

(3) $$\frac{Eval(t) = t' \quad \exists e \in DS : e \triangleright t'}{[rd\ t(\vec{x}).P, DS] \longrightarrow [P[e/\vec{x}], DS]}$$

(4) $$\frac{[P, DS] \longrightarrow [P', DS']}{[P \mid Q, DS] \longrightarrow [P' \mid Q, DS']}$$

(5) $$[abort|P \hookleftarrow Q, DS] \longrightarrow [Q, DS]$$

(6) $$\frac{[P, DS] \longrightarrow [P', DS']}{[P \hookleftarrow Q, DS] \longrightarrow [P' \hookleftarrow Q, DS']}$$

Table 1
Semantics of the basic calculus with interruptable processes.

Rule (1) describes the output operation that produces a new occurrence of the tuple e in the shared space DS ($DS \oplus e$ denotes the multiset obtained by DS increasing by 1 the number of occurrences of e). Rules (2) and (3) describe the *in* and the *rd* operations, respectively: if a matching e tuple is currently available in the space, it is returned at the process invoking the operation and, in the case of *in*, it is also removed from the space ($DS - e$ denotes the removal of an occurrence of e from the multiset DS). Rule (4) represents

a local computation of processes. Rule (5) defines how the *abort* command can be used to interrupt a process and activate the interrupt handler instead. Finally, rule (6) is used to close the transition system w.r.t. the interrutable process composition operator.

3 Communication Latency

Orchestration languages support a time aware programming style. For instance, in the visual orchestration language BizTalk [14], timed activities can be programmed which are interrupted in case they do not complete within a predefined period of time. Similarly, in WS-BPEL [15], it is possible to program signals that are raised at specific time instant, and to install handlers that are triggered by these signals.

Timed process algebras are an extremely powerful tool for modeling and analysing timed systems. There exists numerous models of time inspired by different intuitions and abstractions, see e.g. [1] for a comprehensive overview. According to the traditional taxonomy of timed process algebras, the model of time that we adopt can be classified as *discret* and *asynchronous*. Basic time intervals are considered, and the unique time aware operator is $P \hookleftarrow_n Q$, a timed version of the interrupt operator that permits to interrupt the process P in the case it does not complete within a predefined number of time intervals. This number of intervals is quantified by the timeout n which is a strictly positive natural number (or infinity). Standard operations (output, read and input) take no time. An additional timed relation is defined in order to model the effect of the elapse of one time interval that require to decrease by one unit the timeouts. If one timeout cannot be decreased because it is already equal to 1, the corresponding activity P is interrupted and the interrupt handler Q is activated instead.

Besides timed interruptions, we consider also process and dataspace distribution. Time and distribution are strongly related concepts due to communication latency. In fact, messages exchanged among remote services are delivered after an unpredictaly time delay, and services usually do not wait indefinitely for such these messages. For instance, an orchestrator that sends requests to two services, e.g. an hotel and an airplane reservation service, cannot indefinitely wait for the two answers; in case one of the two reservations does not complete in due time, the whole orchestration is aborted.

Distrubution is achieved simply adding the notion of location $\langle P, DS \rangle_l$ which is a triple composed of a process P, a dataspace DS and a location identifier l. A tuple e can be sent towards a remote location l simply by performing the *out* (e@l) operation.

We consider the new set of location *Loc*, ranged over by l, l', ..., to denote location names. Location names can be communicated, thus a tuple e can contain also locations besides variables and messages.

Processes are the same except that the parameter of the output operation may be also of the kind $e@l$; such a message is inserted in the local data space but will be subsequently sent towards its destination injecting it in the network. Moreover, the transaction operator has an additional timeout $P \hookleftarrow_n Q$ where n is a strictly positive natural number or ∞ (for which $\infty + 1 = \infty$).

$M, N, \ldots ::=$	machines	
$\langle P, DS \rangle_l$		location
$\mid \; e@l$		message
$\mid \; M \| M$		network

We define a predicate $P \downarrow$ that verifies whether a process has committed

$$commit \downarrow$$

$$\text{if } P \downarrow \text{ then } P \hookleftarrow_n Q \downarrow$$

$$\text{if } P \downarrow \text{ and } Q \downarrow \text{ then } P|Q \downarrow$$

We need to extend the notion of structural congruence to machines also. $\equiv$ is now defined as the minimal congruence relation over machines that includes equivalence over locations defined in the previous section and that is such that

$$M|N \equiv N|M \qquad M|(N|L) \equiv (M|N)|L$$

We are now ready to define the function on processes $\phi(\cdot)$ that models the effect of the passing of one time unit

$$\text{if } P \not\downarrow \text{ then } \phi(P \hookleftarrow_{n+1} Q\rangle) = \phi(P) \hookleftarrow_n Q$$

$$\text{if } P \not\downarrow \text{ then } \phi(P \hookleftarrow_1 Q) = Q$$

$$\phi(P|Q) = \phi(P)|\phi(Q)$$

$$\phi(P) = P \text{ in all other cases}$$

Operational semantics is now defined in terms of two transition systems, one describing the execution of operations, another one describing the effect of

the passing of one time unit. More precisely semantics of machines is defined in terms of a timed transition system $(Machine/\equiv, \longrightarrow, \xrightarrow{\surd})$, where: $Machine/\equiv$ is the set of states; $\longrightarrow \subseteq Machine/\equiv \times Machine/\equiv$ is the minimal relation satisfying the set of axioms and rules in Table 1 (where l is added as pedix of systems, i.e., $[P, DS]$ becomes $[\langle P, DS\rangle_l]$, and timeout n is added as index of the interruptable processes, i.e., $P \hookleftarrow Q$ becomes $P \hookleftarrow_n Q$) and in Table 2; and $\xrightarrow{\surd} \subseteq Machine/\equiv \times Machine/\equiv$ is the minimal relation satisfying the set of axioms and rules in Table 3. $([M], [M']) \in \xrightarrow{\surd}$ (also denoted by $[M] \xrightarrow{\surd} [M']$) means that a machine M after one time tick evolves into a machine $M' \in Machine$. When evaluating the semantics of a machine M, we consider $[M] \in Machine/\equiv$ to be the initial state of the transition system.

(7) $$\frac{[M] \longrightarrow [M']}{[M \| N] \longrightarrow [M' \| N]}$$

(8) $$[\langle P, DS \oplus e@l\rangle_{l'}] \longrightarrow [\langle P, DS\rangle_{l'} \mid e@l]$$

(9) $$[\langle P, DS\rangle_l \mid e@l] \longrightarrow [\langle P, DS \oplus e\rangle_l]$$

Table 2
Semantics of the calculus with distribution and time: reduction relation.

(10) $$[\langle P, DS\rangle_l] \xrightarrow{\surd} [\langle \phi(P), DS\rangle_l]$$

(11) $$[e@l] \xrightarrow{\surd} [e@l]$$

(12) $$\frac{[M] \xrightarrow{\surd} [M'] \quad [N] \xrightarrow{\surd} [N']}{[M|N] \xrightarrow{\surd} [M'|N']}$$

Table 3
Semantics of the calculus with distribution and time: timed relation.

The most interesting new rules of the reduction relation are (8) and (9). The former models the injection on the network of a message to be sent to a remote location, while the latter models the delivery of the message. Rule (7) simply lifts the reductions for machines to an entire network of machines. As far as the rules of Table 3 are concerned, rule (10) indicates that the elapsing of one time unit requires the application on processes of the function $\phi(\cdot)$ defined above, rule (11) shows that messages injected in the network are not affected by time passing, and rule (12) states that the effect of the elapsing of one time unit on an entire network is given by the effect of time passing on each of its nodes.

4 Open-endedness

Open-endedness is an inherent characteristics in orchestration of services retrieved from the internet: new services may appear and disappear at run-time, available services (or their efficiency) may depend on their current location or on the current location of the orchestrator (if we deal with mobile entities), requests towards services offering the same service (where "same" is established in terms of some semantical definition of its behavior) may be distributed so to have a balanced workload. Assuming that we know available services and we bind them when the orchestrator is created (i.e. at "compile-time") is not realistic in this context.

Expressing open-endedness in process algebra requires evolved mechanism for channel retrieval to access services. In particular the retrieval should be based on requirements on the desired service, e.g. on some abstraction of its behavior. This can be done in several ways: by using matching rules on tuples of data (formed e.g. by one element representing the channel and others describing the service and its behavior) as in Linda [11] or by using direct subtyping on channels themselves [10]. Note that in this context process algebra may be involved even in the description of the desired behavior of services itself. For example [12] uses abstract process algebraic descriptions as types of systems (services in our case) which are expressed in a more complex process algebra.

In this Section we extend the process calculus previously defined in order to model dynamic retrieval of services. The basic idea is to exploit tuples to describe available services, that can be retrieved using the read and/or input operations. Inspired by the standard UDDI protocol [17], we model a service registry as a node in the network that can be used to publish and discovery new services. One of the main limitation of tuple space used as service description repository is that it has no structure: all tuples in the tuple space have the same relevance, thus it is rather complex to cope with contexts in which there are services that are more important than other because, for instance, provides more powerful resources or connection with a larger bandwidth. To address this limitation, we follow the approach initiated in [5] that consists of associating weights to tuples in order to quantify the relevance of the tuple. The higher is the weight of a tuple, the higher is the probability for that tuple to be retrieved. We first extend the calculus with weights and probabilities, then we formally define how to model a service registry in the new calculus.

4.1 A Calculus with Weighted Tuples

Let $Weight$, ranged over by w, w', ..., be the set of the possible weights. We assume to use positive (non-zero) real numbers as weights, thus $Weight$ coincides with $\Re_+ \setminus \{0\}$. Tuples are now defined as follows:

$$e = \langle \vec{d} \rangle\, [w]$$

where $w \in Weight$ and $\vec{d}$ is a sequence of data fields d that are defined by the following grammar:

$d ::= m \mid w \mid x \mid exp.$

A data field d now can be a message, a weight, a variable or an expression (possibly containing also weights). We also define $\tilde{\cdot}$ as the function that, given a tuple e, returns its sequence of data fields (e.g. if $e = \langle \vec{d} \rangle\, [w]$ then $\tilde{e} = \vec{d}$). In the following, we denote with W the function that, given a tuple, returns its weight (e.g., if $e = \langle \vec{d} \rangle\, [w]$ then $W(e) = w$). Weights are not considered in the matching rule whose definition is unchanged.

4.1.1 The Semantics

The semantics replaces the standard non-deterministic choice of a tuple among the matching ones in the TS, with a probabilistic choice exploiting weights.

We consider probability distributions taken from the set $Prob = \{\rho \mid \rho : Machine/\equiv \longrightarrow [0,1] \wedge supp(\rho) \text{ is finite } \wedge \sum_{[M] \in Machine/\equiv} \rho([M]) = 1\}$, where $supp(\rho) = \{[M] \mid \rho([M]) > 0\}$.

The operational semantics is defined in terms of a probabilistic and timed transition system $(Machine/\equiv, \longrightarrow, \xrightarrow{\surd})$, where: $Machine/\equiv$ is the set of states; $\longrightarrow \subseteq Machine/\equiv \times Prob$ is the minimal relation satisfying the set of axioms and rules that are obtained from those included in Tables 1 and 2 by updating some rules as described in Table 4; and $\xrightarrow{\surd}$ is the minimal relation satisfying the set of axioms and rules in Table 3. $([M], \rho) \in \longrightarrow$ (also denoted by $[M] \longrightarrow \rho$) means that a machine M can evolve (performing a single action) into a probability distribution ρ over machines, such that the machine $M' \in Machine$ is reached with a probability equal to $\rho([M'])$. We use $[M] \longrightarrow [M']$ to denote $[M] \longrightarrow \rho$, with ρ the trivial distribution which gives probability 1 to $[M']$ and probability 0 to all other states. When evaluating the semantics of a machine M, we consider $[M] \in Machine/\equiv$ to be the initial state.

Note that, a machine M can evolve into several probability distributions, i.e. it may be that $[M] \longrightarrow \rho$ for several different ρ. This means that (like in the simple model of [16]) whenever the system is in state $[M]$, first a non-deterministic

$$(2') \quad \frac{Eval(t) = t' \quad \exists e \in DS : e \triangleright t'}{[\langle in\ t(\vec{x}).P, DS\rangle_l] \longrightarrow \rho_{\langle in\ t'(\vec{x}).P, DS\rangle_l}}$$

$$(3') \quad \frac{Eval(t) = t' \quad \exists e \in DS : e \triangleright t'}{[\langle rd\ t(\vec{x}).P, DS\rangle_l] \longrightarrow \rho_{\langle rd\ t'(\vec{x}).P, DS\rangle_l}}$$

$$(4') \quad \frac{[\langle P, DS\rangle_l] \longrightarrow \rho}{[\langle P \mid Q, DS\rangle_l] \longrightarrow \rho|Q}$$

$$(6') \quad \frac{[\langle P, DS\rangle_l] \longrightarrow \rho}{[\langle P \hookleftarrow Q, DS\rangle_l] \longrightarrow \rho \hookleftarrow Q}$$

$$(7') \quad \frac{[M] \longrightarrow \rho}{[M \| N] \longrightarrow \rho \| N}$$

Table 4
Semantics of the calculus with weighted tuples.

choice is performed which decides which of the several probability distributions ρ must be considered, then the next state is probabilistically determined by the chosen distribution ρ. Note that the non-deterministic choice may, e.g., arise from several concurrent rd operations which probabilistically retrieve data from the tuple-space.

Table 5 defines: (i) the probability distributions $\rho_{\langle in\ t(\vec{x}).P, DS\rangle_l}$ and $\rho_{\langle rd\ t(\vec{x}).P, DS\rangle_l}$ used for in and rd operations, respectively; (ii) the operators $\rho|Q$, $\rho \hookleftarrow Q$ and $\rho\|N$, that, given ρ, compute new probability distributions that accounts for parallel composition with process "Q", composition with interrupting process "Q" and parallel composition with machine N, respectively. It is worth noting that $\rho_{\langle in\ t(\vec{x}).P, DS\rangle_l}$ and $\rho_{\langle rd\ t(\vec{x}).P, DS\rangle_l}$ are defined only if $t \in Template$ and $DS \in DSpace$ are such that there exists $e \in DS : e \triangleright t$ (that is the condition reported in axioms (2) and (3)). The meaning of such probability distributions, that are used in axioms and rules of Table 4, is commented in the description of the operational semantics that follows.

Axiom (2') describes the behaviour of in operations; if a tuple e matching with template t is available in the DS, the in execution produces the removal from the space of e and then the process behaves as $P[\tilde{e}/\vec{x}]$. The probability

$$\rho_{\langle in\ t(\vec{x}).P,DS\rangle_l}([M]) =$$

$$\begin{cases} \dfrac{W(e)\cdot DS(e)}{\sum_{e'\in DS:e'\rhd t} W(e')\cdot DS(e')} & \text{if } M \equiv \langle P[\tilde{e}/\vec{x}], DS - e\rangle_l \\ & \quad \text{with } e \in DS \wedge e \rhd t \\ 0 & \text{o.w.} \end{cases}$$

$$\rho_{\langle rd\ t(\vec{x}).P,DS\rangle_l}([M]) =$$

$$\begin{cases} \dfrac{\sum_{e'\in DS:e'\rhd t\wedge P[\tilde{e'}/\vec{x}]\equiv P[\tilde{e}/\vec{x}]} W(e')\cdot DS(e')}{\sum_{e'\in DS:e'\rhd t} W(e')\cdot DS(e')} & \text{if } M \equiv \langle P[\tilde{e}/\vec{x}], DS\rangle_l \\ & \quad \text{with } e \in DS \wedge e \rhd t \\ 0 & \text{o.w.} \end{cases}$$

$$(\rho)f\ ([M]) = \sum_{[M']\in dom(f):\ (\,[M']\,)f\,=\,[M]}\ \rho([M'])$$

where $f : Machine/\equiv \multimap Machine/\equiv$
can be "$|Q$", "$\hookleftarrow Q$" or "$\|N$", defined by:

$$\begin{aligned} (\ [\langle P, DS\rangle_l]\ )|Q &= [\langle P|Q, DS\rangle_l] \\ (\ [\langle P, DS\rangle_l]\ )\hookleftarrow Q &= [\langle P\hookleftarrow Q, DS\rangle_l] \\ (\quad [M] \quad)\|N &= [M\|N] \end{aligned}$$

Table 5
Probability distributions.

of reaching a configuration where a matching tuple e contained in the DS is removed is the ratio of the total weight of the several instances of e in the DS, to the sum of the total weights of the several instances of the matching tuples currently available in the DS. In this way, the probability to reach a system configuration takes into account the multiple ways of removing e due to the several occurrences of e in the DS. The axiom (3') describes rd operations; if a tuple e matching with template t is available in the DS, then the process behaves as $P[\tilde{e}/\vec{x}]$. Differently from the previous axiom, rd operations do not modify the tuple space, i.e. reached states do not change the configuration of DS, therefore they are simply differentiated by the continuation $P[\tilde{e}/\vec{x}]$ of the reading process. For example, let us consider two different tuples $e = \langle d; d_c\rangle[w]$ and $e' = \langle d'; d_c\rangle[w']$. Let $P = rd\ \langle null; null\rangle(x_1; x_2).out(\langle x_2\rangle[w])$ be the process performing the read; it is not possible to discriminate the selection of the two different tuples. Therefore, the probability of reaching a configuration s that

is obtained by reading a tuple e matching with t in the DS (yielding value e) is the ratio of the sum of the total weights associated to the several instances of tuples e' matching with t in the DS such that the continuation of the reading process obtained by reading tuple e' is the same as the one obtained by reading e, to the sum of the total weights of the several instances of the matching tuples currently available in the DS. Rule (4') defines the behaviour of the parallel composition of processes: if states reachable from $[\langle P, DS\rangle_l]$ are described by the probability distribution ρ, and P performs an action in the system $[\langle P \mid Q, DS\rangle_l]$ (the process that proceeds between P and Q is non-deterministically selected), then the reachable states are of the form $[\langle P' \mid Q, DS'\rangle_l]$, for some $P' \in Process$ and $DS' \in DSpace$. The probability values of one such state $[\langle P' \mid Q, DS'\rangle_l]$ does not depend on Q (that is "inactive") and is equal to the summation of the probability values $\rho([\langle P'', DS'\rangle_l])$ for all P'' (among which there is P') such that $[\langle P'' \mid Q, DS'\rangle_l] = [\langle P' \mid Q, DS'\rangle_l]$. The rules (6') and (7') define the behaviour of process interruption and parallel composition of machines in a similar way.

Example 4.1 Service registry - *A service description is a pair composed of two information: task and bind. The former described the kind of task provided and executed by the service. The latter provides the information needed to connect and exploit the service finctionality. A service registry can be seen as a machine whose tuple space is a repository of service descriptions. Two processes run on that machine that are responsible to provides to the client of the registry the publish and discovery operations. The former is used to add a service description in the repository, the latter is used to retrieve and available service description.*

Following this approach, new services can be published sending a request to the registry. We assume that messages containing a publish request have the format $\langle \text{PUBREQ}; mitt; task; bind\rangle$ *where* PUBREQ *is a value that indicates that the tuple is a publish request message, mitt is a location where an acknowledgement of the registration will be sent, and task and bind describes the service to be publishes. On the other hand, a message containing a discovery request must have the format* $\langle \text{DISREQ}; mitt; task\rangle$ *where* DISREQ *is the value characterizing the tuples representing discovery requests, mitt is the location where the description of the retrieved service will be sent, and task described the functionality expected from the retrieved service.*

Namely, the initial state of the service registry is modeled by the machine

$$\langle !Publish \mid !Discovery, \emptyset\rangle_{uddi}$$

where the processes Publish and Discovery are defined as follows:

$$\begin{aligned} Publish &= in\ \langle \text{PUBREQ}; null; null; null\rangle(x; mitt; task; bind).\\ &\quad out(\langle task; bind\rangle[w(mitt, task, bind)]).\ out(\langle \text{PUBACK}; uddi\rangle@mitt)\\ Discovery &= in\ \langle \text{DISREQ}; null; null\rangle(x; mitt; task).\\ &\quad rd\ \langle task; null\rangle(x; bind).\ out(\langle \text{DISRES}; uddi; task; bind\rangle@mitt) \end{aligned}$$

Note that the weight of the tuple that describes the service is computed according to an application dependent expression $w(mitt, task, bind)$. Moreover, two extra special values PUBACK *and* DISRES *are used to qualify messages representing publish acknowledgements and discovery responses, respectively.*

Example 4.2 Service publication and discovery - *A process willing to publish a service on a registry defined as in the Example 4.1 can be modeled by the following machine*

$$\langle DoPub_{task,bind,uddi}, \emptyset\rangle_{user}$$

where the process $DoPub_{task,bind,uddi}$ is as follows:

$$\begin{aligned} DoPub_{task,bind,uddi} &= out(\langle \text{PUBREQ}; user; task; bind\rangle@uddi).\\ &\quad (in\ \langle \text{PUBACK}; uddi\rangle(x_1; x_2).Normal \hookleftarrow_n Timeout) \end{aligned}$$

Note that the process is not willing to wait indefinitely for the acknowledgement. The timed interruption operator is used to activate an alternativa process Timeout responsible to manage situations in which the acknowledgement is not received in due time.

On the contrary, a process willing to discover a new service can be modeled by the following machine

$$\langle DoDis_{task,uddi}, \emptyset\rangle_{user}$$

where the process $DoDis_{task,uddi}$ is as follows:

$$\begin{aligned} DoDis_{task,uddi} &= out(\langle \text{DISREQ}; user; task\rangle@uddi).\\ &\quad (in\ \langle \text{DISRES}; uddi; task; null\rangle(x_1; x_2; x_3; bind).Normal \hookleftarrow_n Timeout) \end{aligned}$$

Also in this case the process is not willing to wait indefinitely for the service binding.

Example 4.3 Orchestrated discovery - *As a final example we present a process which needs to contact two different services in order to compose them. Consider, e.g. the organization of a travel that requires to book a flight as well as the hotel. Two different discoveries are needed to retrieve two specialized*

services, one for flight reservation, and another one for hotel reservation. In case one of the two discoveries fail, it is necessary to interrupt the whole orchestration as it is no more possible to complete it successfully.

In order to program such an orchestrator, we first consider two processes, similar to $DoDis_{task,uddi}$ defined in the prvious example, that retrieve the flight reservation and hotel reservation services, respectively.

$$
\begin{aligned}
FlightDis \;=\; & out(\langle \text{DISREQ}; user; \text{FLIGHT}\rangle @ uddi).\\
& (in\ \langle \text{DISRES}; uddi; \text{FLIGHT}; null\rangle(x_1; x_2; x_3; bind).out(\langle \text{FLIGHT}; bind\rangle)\\
& \hookleftarrow_n abort)\\
HotelDis \;=\; & out(\langle \text{DISREQ}; user; \text{HOTEL}\rangle @ uddi).\\
& (in\ \langle \text{DISRES}; uddi; \text{HOTEL}; null\rangle(x_1; x_2; x_3; bind).out(\langle \text{HOTEL}; bind\rangle)\\
& \hookleftarrow_n abort)
\end{aligned}
$$

We are now ready to model the machine performing the orchestrated discovery

$$
\begin{aligned}
& \langle\ (FlightDis \mid HotelDis \mid\\
& in(\langle \text{FLIGHT}; null\rangle)(x; bind1).in(\langle \text{HOTEL};\text{NULL}\rangle)(y; bind2).Normal)\\
& \hookleftarrow_\infty Failure, \emptyset\ \rangle_{user}
\end{aligned}
$$

where $Normal$ denotes the behaviour in case of successful discovery, while $Failure$ manages the case of failed discovery. The overall discovery fails when one of the two discoveries does not complete in due time. Indeed, in this case the abort process is activated as timed failure handler. The effect of the abort process is to interrupt the overall orchestrated discovery and to activate the $Failure$ process.

5 Conclusion

In this paper we have discussed some of the interesting issues raised by the combination of three relevant characteristics of service oriented computing: loose coupling, communication latency and open-endedness. In particular, we have modeled basic mechanisms for programming timed loosely coupled transactions and for managing dynamic service retrieval according to quantitative information (the weights) that permits to model run time features of services that are not known at design time.

Other interesting aspects raise, for instance, from the combination of loosely coupled transactions and open endedness. In service oriented computing there

is a great interest in negotiation and contract protocols. These are used to select the partners involved in a transaction according to some minimal service requirements. Process algebras have been already used to model specific negotiation protocols used in the context of distributed commitment. The two phase commitment protocol guaranteeing atomicity is analysed in [2], while the BTP protocol guaranteeing a relaxed notion of partial atomicity –called cohesion– was investigated in [3]. These are only specific cases of negotiation protocols; a formal investigation of other protocols such as those based on quality of services, or supporting the dynamic redefinition of the involved partners during the execution of the transaction, is still lacking.

References

[1] J.C.M. Baeten and C.A. Middelburg. Process Algebra with Timing. EATCS Monograph, Springer Verlag 2002.

[2] M. Berger and K. Honda. The Two-Phase Commit Protocol in an Extended π-Calculus. In *EXPRESS'00, Proc. of the 7th International Workshop on Expressiveness in Concurrency*, volume 39 of *ENTCS*, 2000.

[3] L. Bocchi. Compositional Nested Long Running Transactions. In *FASE'04, Proc. of the Fundamental Approaches to Software Engineering*, volume 2984 of *LNCS*, pages 194–208. 2004.

[4] L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long running transactions. In *FMOODS'03, Proc. of the 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, volume 2884 of *LNCS*, pages 124–138. Springer-Verlag, 2003.

[5] M. Bravetti, R. Gorrieri, R. Lucchi, G. Zavattaro. *"Quantitative Information in the Tuple Space Coordination Model"*, *Theoretical Computer Science*, 346:1, pages 28-57, Elsevier, 2005.

[6] R. Bruni, M. Butler, C. Ferreira, T. Hoare, H. Melgratti, and U. Montanari. Reconciling two approaches to compensable flow composition. In *CONCUR'05, Proc. of the 16th International Conference on Concurrency Theory*, volume to appear of *LNCS*, 2005.

[7] R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL'05, Proc. of the 32nd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, 2005

[8] M. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *COORDINATION'04, Proc. of the 6th International Conference on Coordination Models and Languages*, volume 2949 of *LNCS*, pages 87–104. Springer-Verlag, 2004.

[9] M. Butler, T. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In Proceedings of 25 Years of CSP, London, 2004.

[10] G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the pi-calculus. In *LICS'05, Proc. of Logic in Computer Science*, june 2005.

[11] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[12] A. Igarashi, N. Kobayashi. A Generic Type System for the Pi-Calculus. In Theoretical Computer Science, 311(1-3):121-163, 2004.

[13] C. Laneve, and G. Zavattaro. Foundations of Web Transactions. In *FOSSACS'05, Proc. of the 8th Foundations of Software Science and Computational Structures*, volume 3441 of *LNCS*, pages 282–298. Springer-Verlag, 2005.

[14] Microsoft BizTalk Server. [http://www.microsoft.com/biztalk/default.asp], Microsoft Corporation.

[15] 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].

[16] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1995.

[17] Universal Description, Discovery and Integration for Web Services (UDDI) V3 Specification.