User-Excentric Service Composition in Pervasive Environments

Noha Ibrahim
Grenoble Informatics Laboratory
Grenoble, France
Email: noha.ibrahim@imag.fr

Stéphane Frénot
Université de Lyon,INRIA
Villeurbanne, France
Email: stephane.frenot@insa-lyon.fr

Frédéric Le Mouël
Université de Lyon, INRIA
Villeurbanne, France
Email: frederic.le-mouel@insa-lyon.fr

***Abstract*—In pervasive environments, services are fastly developing and are being deployed everywhere. In this article, we introduce a Servicebook, a new social network of services, where services create and join group of service profile providing to users better access to all the services in their vicinity. We propose a novel technique to realize this Servicebook, the user-excentric service composition. This user-excentric composition relies on two service relations: the compatible relation and the composition relation. We developed and evaluated an OSGi-prototype as a proof-of-concept.**

***Keywords*-SOA; social network of services, user-excentric composition; semantic matching**

I. Introduction

The development of service-based computing technologies and communication networks opened the way for new trends, where services are the main actors of their environments. We borrow the human social behavior and define a service social behavior as everything that concerns a service in a society composed of services. A society of service is defined by the service profiles evolving in an organised group. We define the organisation of a society around interface compatibilities. One of this idea challenges is in defining the service profile upon which groups of services are formed and the criteria for services to join groups of their interest. The natural idea is to use the interface description profile. Services with the same profile, come together to form a group. A group is formed by all the services having the same interface profile but different capabilities (implementations, non functional properties, etc).

To create this service society named `Servicebook`, where services gather according to their interface profiles, we propose a user-excentric composition. The composition is excentric because not requested by users, but yet it provides composed services with new capabilities but already existing interfaces. All the service composition technologies ([8], [3], [5], [2], [10]) are user-centric as they await a user request to compose services. The composition we are after consists of (1) composing services of the environment without prior user request to extend the `Servicebook` with new capabilities and for that (2) this user-excentric composition needs to provide services with new capabilities (implementations, non functional properties, etc) but with the same interface profile as the services already existing in the `Servicebook`.

Based on the above, the contributions of this article are threefold. First, the article defines a new social network of services, the `ServiceBook`. Second, it describes using a formal language two kinds of relations between services: the compatible relation and the composition relation. Finally, it explains the user-excentric service composition based on these relations. What is new with our approach is that we are the first one to propose a user-excentric service composition approach in opposition to the user-centric one. We propose a novel application for this user-excentric composition, the `Servicebook` social network. We also define and formalize service compatible and composition relations which have not been done before.

To better appreciate the approach, we consider a running example, consisting of services and operations providing different capabilities to users. Upon this running example we illustrate our compatible and composition service relations, followed by our user-excentric service composition. The remaining of this paper is organised as follows. Section 2 is a state of the art for the service composition in pervasive environments. Section 3 defines the `Servicebook` as a social network of services. Section 4 introduces the service model and the semantic matching tools. Section 5 explains the service compatible and composition relations. Section 6 depicts the user-excentric service composition. Section 7 details the developed proof-of-concept prototype and its results. Section 8 concludes this work.

II. State of the Art

Service composition allows the combination of multiple services into a single composite service, which may be achieved at design-time (static) or at run-time (dynamic). In current middleware and systems, dynamic service composition is very often associated with the realization, on the fly, of user tasks. Indeed, service composition can be a major key for the user-centrism paradigm by enabling the user to be at the heart of the realization of his daily tasks through the combination of relevant services available in the vicinity. The three composition middleware depicted in this section (PERSE [8], SeGSeC [3], and Broker [2]) are, as all other major current service composition middleware (SeSCo [5], WebDG [6], eFlow [1], SWORD [10], and Contract-Based composition [7]), user-centric as they dynamically compose services in response to a user task.

PERSE [8] introduces the architecture of a semantic service registry for pervasive computing. This registry allows heterogeneous service capabilities to be registered and retrieved by translating their corresponding descriptions to a predefined service model. Service discovery protocol interoperability requires the translation of service advertisements into a common service description language for enabling service matching and composition to be performed independently from the specific underlying languages. Once the translation done, the services can be published, stored, compared or composed depending on what is needed in the environment.

SeGSeC [3] proposes an architecture that obtains the semantics of the requested service in an intuitive form (e.g. using a natural language), and dynamically composes the requested service based on its semantics. To compose a service based on its semantics, the proposed architecture supports semantic representation of services - through a component model named *Component Service Model with Semantics (CoSMoS)* - discovers services required for composition - through a middleware named *Component Runtime Environment (CoRE)* - and composes the requested service based on its semantics and the semantics of the discovered services - through a service composition mechanism named *Semantic Graph-Based Service Composition (SeGSeC)*

Broker [2] presents a distributed architecture and associated protocols for service composition in mobile environments that take into consideration mobility, dynamic changing service topology, and device resources. The composition protocols are based on distributed brokerage mechanisms and utilize a distributed service discovery process over ad-hoc network connectivity. The proposed architecture is based on a composition manager, a device that manages the discovery, integration, and execution of a composite request. Two broker selection-based protocols - dynamic one and distributed one - are proposed in order to distribute the integration requests to the composition managers available in the environment.

These service composition middleware are clearly user-centric as they fulfill the user requests. To our knowledge, User-excentric service composition, that composes services without the user request but still providing new functionalities to the environment, is not considered.

III. Servicebook: A Social Network of Services

Social networking has encouraged new ways to communicate and share information. A social network focuses on building online communities of people who share interests and/or activities, or who are interested in exploring the interests and activities of others. In a human social network, people get together because of certain compatibilities they define in advance, and get to create groups of interest where they can exchange. In general, social networks allow users to create a profile for themselves and groups that share common interests.

We define a social network of services, the `Servicebook`, inspired from the human social networks. `Servicebook` gets services together by grouping them according to their interfaces. Services publish their profiles which are (i) interface semantic signature, (ii) functional properties describing the service implementation and (iii) non functional properties describing the service context. Services providing the same interface semantic signature (i) come together to form a group of interest. Each service bring its functional properties (ii) and non functional properties (iii) as a specificity to the group. All the services of a same group publish the same interface (i). Groups of services are created where each group publishes a same interface but provides multiple implementations and/or non functional properties(cf. Figure 1). Each group will see it community grows or shrinks depending on what services are available in the environment.

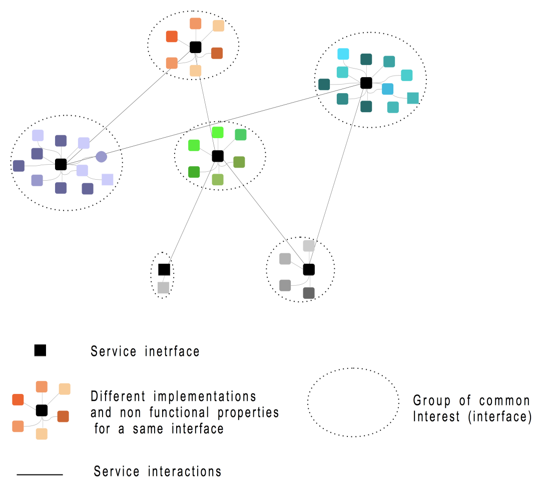

Service inetrface
Different implementations and non functional properties for a same interface
Group of common Interest (interface)
Service interactions

Figure 1. Servicebook: a social network of services

The `Servicebook` provides extended capabilities to the users. A service user searching for a specific interface will seek for the interface specific group and will have a large choice of implementations (ii) and non functional properties (iii) within this group (cf. Figure1).

The `Servicebook` gains to be web based and can provide a variety of ways for services to interact via standards such as SOAP. Services can also be hosted on ambient devices carried by users and come to meet as users encounter. This creates an ambient social network of services where users can benefit from the new capabilities of their services. In this article, important issues such as security and trust are not considered. In social networks of services these issues are essential as service may condition its participation to a particular group depending on these criteria.

Now that we defined what is a `Servicebook`, how to create a group and how to join it. We define in next sections the user-excentric composition as an efficient tool to update the `Servicebook` with new services. Indeed, finding services that offer the same interface but different capabilities is not always easy and we argue that the user-excentric composition is an efficient way to do this. We first begin by formalizing the service model and the service relations followed by the user-excentric composition definition.

IV. Service Formalization

In this section, we define our service model and the semantic matching tools for comparing different concepts belonging to a same ontology.

A. Service Model

A service has functional properties corresponding to the operations it computes and non functional properties describing the operation behaviour. Non functional properties correspond to all the properties that characterize the operation context excluding the operation implementation itself.

Consider finite sets of grammatical alphabet Σ, ontologies `O`, concepts `N` belongings to these ontologies `O`, functional properties `P`, non functional properties `Np`, quantitative and qualitative non-functional properties $\mathtt{Np}_{QN}$, $\mathtt{Np}_{QL}$.
Consider the following operators:

- * (repetition zero or more times)
- $^{0..1}$ (zero or one time)
- $|A|$ (the cardinality if applied to a set A)

We define functional properties of an operation as follows:

Definition 1 — Functional property definition

$op:\ (In^*,\ Out^{0..1}, P,\ Np^*)$
$in:\ (name, semantic),\ name\ \in\ \Sigma^*$
$out:\ (semantic)$
$p:\ (semantic, type)$
$type \in\ \{atomic, composed\}$
$semantic = (o,n),\ o\ \in\ O,\ n\ \in\ N$

■

Where:

- In is the set of the operation op inputs. $In = \{in_1\ \dots\ in_{|In|}\}$. $\forall\ in\ \in In$, in is defined as a tuple: $<\ name, semantic\ >$, where $name$ is the chosen input syntactic name and $semantic$ the input semantic description.
- out is the operation op output. out is defined as a tuple $<semantic>$ where $semantic$ its the semantic description.
- P the functional property of a service describing its implementation. This property also indicates whether the service is atomic or composed.

We define non functional properties of an operation op as follow:

Definition 2 — Non-functional property definition

$Np:\ \{Np^*_{QL}, Np^*_{QN}\}$
$Np_{QL}:\ \{np1_{QL},\ np2_{QL},\ ..\ npk_{QL}\},\ k\ =\ |NP_{QL}|$
$Np_{QN}:\ \{np1_{QN},\ np2_{QN},\ ..\ npt_{QN}\},\ t\ =\ |NP_{QN}|$
$np_{QL}:\ (name, semantic),\ name\ \in\ \Sigma^*$
$np_{QN}:\ (name, value, operator), name \in \Sigma^*, value \in \mathbb{R}$
$operator\ \in \{<,\ >,\ =\}$
$semantic:\ (o,n),\ o\ \in\ O,\ n\ \in\ N$

■

where:
Np is the set of non functional properties characterizing the operation. Np can be qualitative or quantitative.

- $np_{QL}\ \in\ Np_{QL}$ is the qualitative non functional properties defined as a tuple $(name, semantic)$, where $name$ is the syntactic name through which the property is specified and $semantic$ is its semantic description.
- $np_{QN}\ \in\ Np_{QN}$ is the quantitative non functional properties defined as a tuple, where $value\ \in\ \mathbb{R}$ and $operator\ \in\ \{>,<,=\}$. $Operator$ specifies the value range the service can offer. For '$>$' the $value$ is the smallest value the service can offer during its runtime execution. For '$<$' the $value$ is the biggest value the service can offer during its service runtime execution. For '$=$' the $value$ is the only value the service can offer during its service runtime execution.

Example 1: *Three service examples are given in Figure 2. For conciseness and clarity we suppose that each service has one operation.*

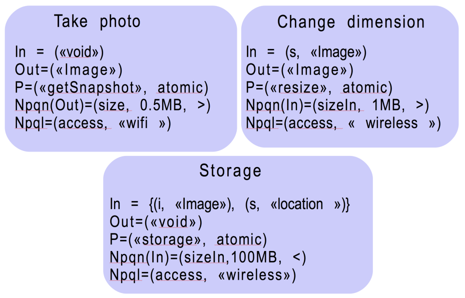

Take photo
In = («void»)
Out=(«Image»)
P=(«getSnapshot», atomic)
Npqn(Out)=(size, 0.5MB, >)
Npql=(access, «wifi »)
Change dimension
In = (s, «Image»)
Out=(«Image»)
P=(«resize», atomic)
Npqn(In)=(sizeIn, 1MB, >)
Npql=(access, « wireless »)
Storage
In = {(i, «Image»), (s, «location »)}
Out=(«void»)
P=(«storage», atomic)
Npqn(In)=(sizeIn,100MB, <)
Npql=(access, «wireless»)

Figure 2. Running example services

The chosen non functional properties are:

- *The quantitative property size describes the input and/or output parameters. It indicates how many MB the service can handle. The operator $>$ indicates the smallest value the service can offer or require. The $Takephoto$ operation produces an image with a size of at least 0.5MB. The $ChangeDimension$ operation consumes an image which size is no less than 1MB. The operator $<$ indicates the biggest value the service can offer or require. The Storage service may only stores images that are smaller than 100MB.*

- *The qualitative property access describes the overall operation. It indicates how to connect to the printer device. The concepts belong to the same ontology (cf. Figure 3).*

The functional properties describe the operation implementation and indicates that the three services are atomic.

We define two operations opi and opj to be comparable if they have the same number of inputs and the same number of outputs and if it exists a bijection f over their inputs allowing to compare the inputs parameters two by two:

$$\exists\, f : In_{opi} \rightarrow In_{opj},\ \forall\, in_l \in In_{opj},\ \exists!\, in_k \in In_{opi},\ f(in_k) = in_l$$

We define two services to be comparable if they have the same number of operations and if it exists a bijection f over their operations allowing to compare them two by two.

It may also be interesting to compare services together even if they do not have the same number of operations but at least two comparable operations. All the following definitions can be applied to this special case where two or more services are comparable over a sub-set of their operations but not all of their operations.

B. Semantic Matching Tools

The matching of two concepts belonging to the same ontology has been widely studied. We define M_{Cpt} our matching relation between concepts that belong to the same ontology.

A concept n belonging to an ontology o, can provide all its immediate sub-concepts n_1 and n_2 (Paolucci [9]) or one of its sub-concepts n_1 or n_2. We fall into the first category, stipulating that a super-concept offers what its sub-concepts offer, and hence can replace them.

Defining n and m, two concepts belonging to the same ontology o. We define the four values of concept matching M_{Cpt} inspired from Paolucci as follows:

$M_{Cpt}(n, m)$ values are:

- $Exact$ If n and m are equivalent concept
- $PlugIn$ If n is a super-concept of m
- $Subsume$ If n is a sub-concept of m
- $Fail$ If n and m do not verify the above conditions

Following Paolucci matching, we choose not to consider the siblings in our semantic matching and by that we stipulate that two concepts are not compatible even if they have the same parents.

EXAMPLE 2: *Using our ontology Figure 3, we give an example of M_{Cpt} matching.*

$$M_{Cpt} \left| \begin{array}{l} (''Wireless'',\ ''Bluetooth'') = PlugIn \\ (''Wireless'',\ ''Wifi'') = PlugIn \\ (''Wireless'',\ ''Access'') = Subsume \\ (''Wireless'',\ ''Wired'') = Fail \\ (''Wireless'',\ ''802.11a'') = PlugIn \end{array} \right.$$

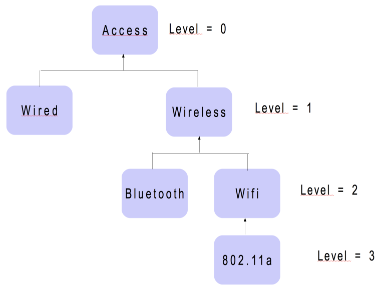


Figure 3. *access* ontology

The two values of of $Exact$ and $Fail$ are the best and worst case. For the $PlugIn$ and $Subsume$ values, we consider the $PlugIn$ to be better than the $Subsume$ as a concept can provide all its immediate sub-concepts but not necessary the other way round (Paolucci definition).

V. SERVICE RELATION FORMALIZATION

In this section, based on the above tools, we define our service relations. These latter are the cornerstone of our user-excentric service composition.

A. Service Compatible Relation

Based on the concept matching, we define the semantic matching between two comparable operations.
The operation op_i is:

- $Exact$ semantic matching to op_j if all the matching values between their respective comparable inputs and their respective output are $Exact$.

$$M_{Cpt}(In_{op_i}, In_{op_j}) = Exact$$
and
$$M_{Cpt}(Out_{op_i}, Out_{op_j}) = Exact$$

- $PlugIn$ semantic matching to op_j if they are not $Exact$ matching and all the matching between their respective comparable inputs and their respective output values are $Exact$ or $PlugIn$.

$$M_{Cpt}(In_{op_i}, In_{op_j}) \in \{Exact \vee PlugIn\}$$
and
$$M_{Cpt}(Out_{op_i}, Out_{op_j}) \in \{Exact \vee PlugIn\}$$

- $Subsume$ semantic matching to op_j if they are no $Exact$ or $PlugIn$ matching and at least one matching value between their respective comparable inputs or their respective output is $Subsume$

$$M_{Cpt}(In_{op_i}, In_{op_j}) = Subsume$$
or
$$M_{Cpt}(Out_{op_i}, Out_{op_j}) = Subsume$$

and no $Fail$ matching value is found between outputs, and the corresponding comparable inputs.

$$M_{Cpt}(In_{op_i}, In_{op_j}) \neq Fail$$
and
$$M_{Cpt}(Out_{op_i}, Out_{op_j}) \neq Fail$$

- They are $Fail$ semantic matching to op_j if at least one semantic matching value between their respective comparable inputs or their respective outputs is $Fail$.

$$M_{Cpt}(In_{op_i}, In_{op_j}) \ = \ Fail$$
or
$$M_{Cpt}(Out_{op_i}, Out_{op_j}) = \ Fail$$

We define operation op_i to be semantically compatible $\equiv_{sem}$ with operation op_j, if the semantic matching of op_i with op_j is $Exact$ or $PlugIn$ semantic matching.

DEFINITION 3 — $(\equiv_{sem} (op_i, op_j) = True)$ if

$| \ M_{Cpt}(op_i, op_j) \in \{Exact \vee PlugIn\}$

■

The compatible relation implies that op_i can easily replace op_j as it has a semantic compatible signature. Each user of the op_j operation can be satisfied with the op_i operation. He can not easily distinguish the difference as the semantic matching we adopted assures a complete compatibility.

As we did for the operations, we define the semantic matching between two comparable services S_i and S_j:

- Services S_i is $Exact$ semantic match to S_j if all the comparable operations of S_i are $Exact$ matching with the respective comparable operations of S_j.
- Services S_i is $PlugIn$ semantic match to S_j if all their comparable operations of S_i are $Exact$ and $PlugIn$ matching with the respective comparable operations of S_j. At least one $PlugIn$ matching must be found.
- Services S_i is $Subsume$ semantic match to S_j if they are not $Exact$ nor $PlugIn$ semantic match and no $Fail$ matching is found between any comparable operations of the two services.
- Services S_i is $Fail$ semantic match to S_j if at least one operation of S_i is $Fail$ semantic match to a comparable operation of S_j.

We define operation S_i to be semantically compatible $\equiv_{sem}$ to S_j if S_i is $Exact$ or $PlugIn$ semantic matching to S_j.

DEFINITION 4 — $(\equiv_{sem} (S_i, S_j) = True)$ if

$| \ M_{Cpt}(S_i, S_j) \in \{Exact \vee PlugIn\}$

■

As for the operations, the compatible relation implies that S_i can easily replace S_j as it has a semantic compatible interface. Each user of S_j can be satisfied with S_i. He can not easily distinguish the difference as the semantic matching we adopted assures a complete compatibility.

S_i and S_j can also be not semantically compatible but have semantically compatible operations. We can then say that S_i is semantically compatible to S_j over a subset of their operations. This can be very useful as it is difficult to always find comparable services having all their operations compatible.

We need to specify that we will only consider the functional properties of services for the compatible relation. We defined in [4] a function that computes the compatibility between service non functional properties. But one of the goals of our user-excentric service composition is to provide different non functional properties as we will see in section V and for that reason, no compatibility over non functional properties is required.

B. Service Composition Relation

After the service compatible relation definitions, we define the service composition relations. A service is considered to be composable with another service, if the functionality it produces can be consumed by the other service, and if the non-functional properties are compatible. Combining two services together means combining their functional interfaces, and by that at least one of their operations. The output produced by the operation of one service is used as an input for the operation of the other service. To be able to compose services, non-functional properties need to be taken into account, indeed if incompatibilities exist no composition can take place.

Two services S_i and S_j are semantically compatible for composition (O_{sem}) if they have at least two semantic composable operations.

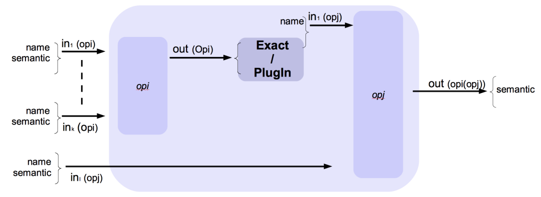


Figure 4. Operation op_i composable with operation op_j

Two operations are semantic composable if the semantic description of the output of one operation is $Exact$ or $Plugin$ matching to the semantic description of one input of the other operation (cf. Figure 4).

But even if two operations are functionally composable, they can present incompatible non-functional properties that prevent from establishing a valid composition relation between the concerned operations. The non-functional properties we are concerned with, are those that describe and specify the output and the input that are combined together. Indeed, the composition relation is built upon the condition that the output of one operation is consumed as an input for the other operation. The non-functional properties dealing with these parameters need to be compatible.

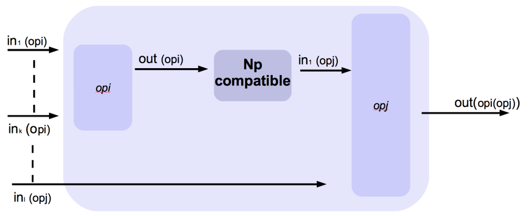


Figure 5. Non functional property compatibility

The non functional property compatibilities depend on the properties type. For qualitative properties, considering $np_{QL} = (name, semantic)$ a common property for both op_i and op_j. The $semantic$ of np_{QL} of out_{op_i} must be $Exact$ or $PlugIn$ semantic match with the $semantic$ of np_{QL} of in_{op_j}. By that way the out_{op_i} satisfies the functional and non functional requirement of in_{op_j}. If the $semantic$ of np_{QL} of out_{op_i} is $Subsume$ semantic match with the $semantic$ of np_{QL} of in_{op_j} there is no guarantee that the composition works as the $subsume$ concept is not capable of providing all the required concept of its super-concept.

For the quantitative property, considering $np_{QN} = (name, value, operator)$ a common property for both op_i and op_j. To be able to compose the two operations, the intersection of their respective values must not be equal to zero, otherwise, no composition can take place.

All these relations are illustrated via our running example in the next section.

VI. User-Excentric Service Composition

We first begin by defining the user-excentric service composition and illustrating this composition via our running example. Then, we return to our `Servicebook` and show how the user-excentric composition is used for building and updating such a service network.

A. *Definitions and Techniques*

Using the previous definition on service relations, we define in this section our user-excentric service composition. This composition is not requested by users and hence need to be as transparent as possible. To define the composition over services we begin by defining the composition over operations. Considering three operations op_i, op_j and op_k. we define a user-excentric composition as follows:

Definition 5 — User-excentric operation composition

$O_{sem}(op_i, op_j) = True$
and
$\exists l \in \{i, j, k\}, \equiv_{sem} (op_{comp(op_i, op_j)}, op_l) = True$

■

The new resulting composed operation $op_{comp(op_i, op_j)}$ is compatible with one of the three operations (op_i, op_j, op_k) which means that it is invisible to the users (it has the same signature as already existing operations) but hence propose new functionalities (which correspond to the combination of the two operations op_i and op_j).

Example 3: *Considering our running example, two user-excentric service composition are possible as shown in Figure 6.*

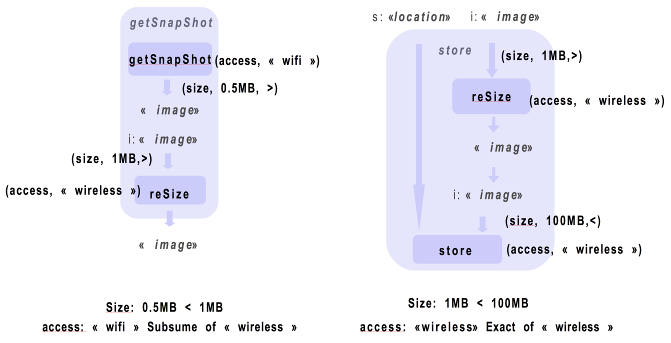

getSnapShot
getSnapShot (access, « wifi »)
(size, 0.5MB, >)
« image»
i: « image»
(size, 1MB,>)
(access, « wireless ») reSize
« image»
s: «location» i: « image»
store
(size, 1MB,>)
reSize (access, « wireless »)
« image»
i: « image»
(size, 100MB,<)
store (access, « wireless »)
Size: 0.5MB < 1MB
access: « wifi » Subsume of « wireless »
Size: 1MB < 100MB
access: «wireless» Exact of « wireless »

Figure 6. User-excentric operation composition

As shown Figure 6, the new composed getSnapshot operation has the same signature as the atomic getSnapshot but with extended functionalities. It can resize the picture image it produces. The new composed store operation has the same signature as the atomic store with extended implementation allowing to resize images before storing them.

Concerning the non functional property compatibility, the store composition is always possible as the qualitative property of the reSize output is Exact semantic matching with the store input qualitative property, and the intersection of their quantitative property exists. For the getSnapshot the composition is not possible all the time due to the Subsume semantic match between the respective qualitative property.

Based on DEFINITION 5 we define the user-excentric service composition as follows:

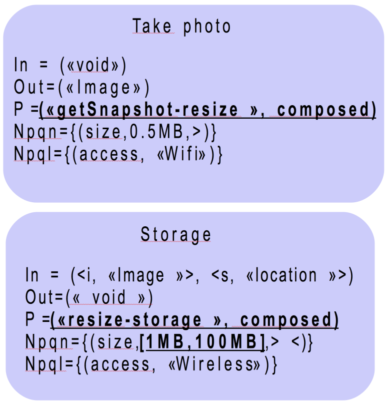
Definition 6 — User-excentric service composition

$O_{sem}(S_i, S_j) = True$
and
$\exists l \in \{i, j, k\}, \equiv_{sem} (S_{comp(S_i, S_j)}, S_l) = True$

■

The new resulting service is compatible to an already existing service as it publishes the same interface.

EXAMPLE 4: *Figure 7 describes the functional and non functional properties of the two services* Take photo *and* Storage. *The functional properties reflect the service composition and the non functional properties are the intersection between those of the composed services. These new composite services extend the environment with new implementations (functional property) and new non functional properties. This result reflects our choice to restrict the semantic compatibility between services and operations only to their interface signature. By this way we extend the environment not only with new implementations but also new non functional properties.*



Take photo

```
In = («void»)
Out=(«Image»)
P =(«getSnapshot-resize », composed)
Npqn={(size,0.5MB,>)}
Npql={(access, «Wifi»)}
```

Storage

```
In = (<i, «Image »>, <s, «location »>)
Out=(« void »)
P =(«resize-storage », composed)
Npqn={(size,[1MB,100MB],> <)}
Npql={(access, «Wireless»)}
```

Figure 7. New composed service descriptions

B. `Servicebook` Application

Coming back to the `Servicebook` application, we argue that the user-excentric composition is an efficient tool for updating this social network with all the capabilities the environment can offer. When a service appears in the `Servicebook` environment, it first checks whether a group of its interface description already exists. If so, the new service joins this group and brings with it its functional implementation and non functional properties. If no group already exists, it creates a new group publishing this new interface.

We suppose that three atomic different services are available in the environment S_i, S_j and S_k. For each service, a group is created hosting the interface of the service and the corresponding implementation (cf. Figure 8).
Supposing that:

$O_{sem}(S_i, S_j) = True$
and
$\exists l \in \{i, j, k\}, \equiv_{sem} (S_{comp(S_i,S_j)}, S_l) = True$

The new composed service $S_{comp(S_i,S_j)}$ is added to the S_l group $\forall\ l \in \{i, j, k\}$. Each group is extended with new user-excentric composite service appearing (in Figure 8 the S_k group is extended as we suppose $l = k$).

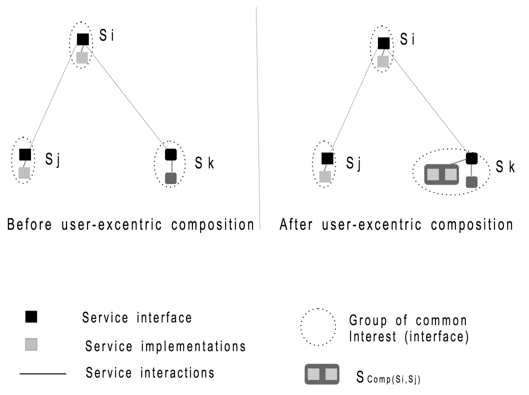

S i
S j
S k
Before user-excentric composition
S i
S j
S k
After user-excentric composition
Service interface
Service implementations
Service interactions
Group of common Interest (interface)
S Comp(Si,Sj)

Figure 8. User-excentric composition in `Servicebook`

When the service disappears (disappearance of one of the service involved in the composition) the $S_{comp(S_i,S_j)}$ is removed from the group. When all the implementations of a given interface are gone the group is destroyed.

VII. EVALUATION AND RESULTS

We implemented, as a proof of concept, all the major functionalities of the user-excentric service composition under an OSGi service platform implementation, the Apache Felix. For the evaluations we developed a simulation use case composed of 100 OSGi services in a small environment deployed on three laptops. We evaluated the time and memory consumption of the semantic tools, service relation computing and the user-exentric service composition.

- The prototype matches all the inputs and outputs of these services together in order to find all the compatible services from these 100 services. We suppose that each service has one operation. Table I gives the time and memory consumption values for matching two operations and 100 operations.

Table I
TIME EXECUTION AND MEMORY CONSUMPTION FOR SEMANTIC MATCHING

OSGi *Op*	Time (ms)	Memory (Ko)
2	400	2000
100	1000	4000

- The time to find composable services from 100 services is the same as above but we add to it the non functional property matching. For qualitative properties as it is based on the same semantic tool table I gives the time to compare semantic properties together. Table II gives the values for quantitative property.

Table II
TIME EXECUTION AND MEMORY CONSUMPTION FOR QUANTITATIVE PROPERTY COMPUTING

NP_{QN}	Time (ms)	Memory (Ko)
2	15	7
100	47	47

We notice that semantic matching is much slower and consume more than simple quantitative computing. Indeed, accessing an ontology to find the needed concept can be tedious depending on the ontology length and depth.

- Table III gives the time and memory values to compose two services together and 100 services two by two. The composition technique we adopted is in creating for each newly composed service a new unit of deployment independent from the services taking part in the composition.

Table III
TIME EXECUTION AND MEMORY CONSUMPTION FOR USER-EXCENTRIC SERVICE COMPOSITION

OSGi services	Time (ms)	Memory (Ko)
2	200	100
100	5000	7000

We can notice that the more the number of composed service is, the slower and heavier is the composition process. This is due to our composition technique that creates for each newly composed service a new unit of deployment. Another technique would be to only redirect the call to the services involved in the composition allowing by that our user-excentric service composition to scale to large environments.

VIII. CONCLUSION

In this article, we proposed a new way to compose services in a user-excentric way. This novel composition is ensured by simply combining two very well known service relations: the compatible relation and the composition one. We formalized the service compatible and composition relations and proposed an easy combination of them in order to propose this new way of composing services. This user-excentric service composition extends the environment with new capabilities even if not required at that moment by users with a highly appreciable condition, keeping the service and operation signatures unchanged. We proposed, the `Servicebook` as an application use case for the user-excentric composition. This new type of service social network provides to users all the capabilities an environment can offer in terms of services. We implemented a prototype under Java OSGi framework as a proof of concept and evaluated the efficiency of our proposal. The next step would be to test our prototype in large and highly dynamic environments, such as university campus, were thousands of services may meet and where a real end user experience could be tested to evaluate the interest, efficiency and scalability of our user-excentric service composition approach in these highly dynamic environments. Later on, we aim to develop the `Servicebook` on the web to evaluate the utility and attractiveness of such networks for Web applications.

REFERENCES

[1] Fabio Casati, Ski Ilnicki, Li jie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in eflow. In *CAiSE '00: Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, pages 13–31, London, UK, 2000. Springer-Verlag.

[2] Dipanjan Chakraborty, Anupam Joshi, Tim Finin, and Yelena Yesha. Service Composition for Mobile Environments. *Journal on Mobile Networking and Applications, Special Issue on Mobile Services*, 10(4):435–451, 2005.

[3] Keita Fujii and Tatsuya Suda. Semantics-based dynamic service composition. *IEEE Journal on Selected Areas in Communications*, 23(12):2361– 2372, December 2005.

[4] Noha Ibrahim, Frédéric Le Mouël, and Stéphane Frénot. MySIM: a Spontaneous Service Integration Middleware for Pervasive Environments. In *ACM International Conference on Pervasive Services (ICPS)*, volume 4836/2008. ACM, July 2009.

[5] Swaroop Kalasapur, Mohan Kumar, and Behrooz Shirazi. Dynamic Service Composition in Pervasive Computing. *IEEE Trans. Parallel Distrib. Syst.*, 18(7):907–918, 2007.

[6] Brahim Medjahed, Athman Bouguettaya, and Ahmed K. Elmagarmid. Composing Web services on the Semantic Web. *The VLDB Journal*, 12(4):333–351, 2003.

[7] Nikola Milanovic. *Contract-based Web Service Composition*. PhD thesis, University of Berlin, 2006.

[8] Sonia Ben Mokhtar. *Semantic Middleware for Service-Oriented Pervasive Computing*. PhD thesis, University of Paris 6, December 2007.

[9] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Semantic matching of web services capabilities. *The semantic Web - ISWC*, 2342/2002, 2002.

[10] Shankar R. Ponnekanti and Armando Fox. SWORD: A developer toolkit for web service composition. In *11th World Wide Web Conference*, 2002. Honolulu, USA.

[11] Mahadev Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communication*, August 2001.